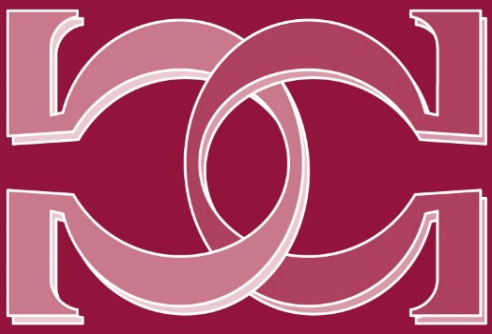
Cristian S. Calude
Michael J. Dinneen
Gheorghe Păun
Grzegorz Rozenberg
Susan Stepney (Eds.)

LNCS 4135

Unconventional Computation

5th International Conference, UC 2006
York, UK, September 2006
Proceedings



Springer

Lecture Notes in Computer Science 4135

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Cristian S. Calude Michael J. Dinneen
Gheorghe Păun Grzegorz Rozenberg
Susan Stepney (Eds.)

Unconventional Computation

5th International Conference, UC 2006
York, UK, September 4-8, 2006
Proceedings

Springer

Volume Editors

Cristian S. Calude
Michael J. Dinneen
University of Auckland, Department of Computer Science
Private Bag 92019, Auckland, New Zealand
E-mail: {cristian,mjd}@cs.auckland.ac.nz

Gheorghe Păun
Institute of Mathematics of Romanian Academy
P.O. Box 1-764, 014700 Bucureşti, Romania
and
Sevilla University, Department of Computer Science and AI
Avda Reina Mercedes s/n, 41012 Sevilla, Spain
E-mail: gpaun@us.es

Grzegorz Rozenberg
Leiden University, Leiden Center of Advanced Computer Science (LIACS)
Niels Bohrweg 1, 2333 CA Leiden, The Netherlands
E-mail: rozenber@liacs.nl

Susan Stepney
University of York, Department of Computer Science
Heslington, York, YO10 5DD, UK
E-mail: susan@cs.york.ac.uk

Library of Congress Control Number: 2006931474

CR Subject Classification (1998): F.1, F.2

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN 0302-9743
ISBN-10 3-540-38593-2 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-38593-6 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11839132 06/3142 5 4 3 2 1 0

Preface

The 5th International Conference on Unconventional Computation, UC 2006, organized under the auspices of the EATCS by the Centre for Discrete Mathematics and Theoretical Computer Science of the University of Auckland, and the Department of Computer Science of the University of York, was held in York, UK, September 4–8, 2006.

York combines evidence of a history going back to Roman times with a bustling modern city center. The Minster, built on the foundations of the Roman city and an earlier Norman cathedral, is among the finest Gothic cathedrals, and dominates the city. Romans, Vikings, and more recent history are commemorated in a number of top-class museums, as well as being apparent in the architecture of the city.

The series of International Conferences on Unconventional Computation (UC), `https://www.cs.auckland.ac.nz/CDMTCS/conferences/uc/`, is devoted to all aspects of unconventional computation, theory as well as experiments and applications. Typical, but not exclusive, topics are: natural computing including quantum, cellular, molecular, neural and evolutionary computing; chaos and dynamical systems-based computing; and various proposals for computations that go beyond the Turing model.

The first venue of the Unconventional Computation Conference (formerly called Unconventional Models of Computation) was Auckland, New Zealand in 1998; subsequent sites of the conference were Brussels, Belgium in 2000, Kobe, Japan in 2002, and Seville, Spain in 2005.

The titles of volumes of the past UC conferences are the following:

1. C.S. Calude, J. Casti, M.J. Dinneen (eds.). *Unconventional Models of Computation*, Springer, Singapore, 1998, viii + 426 pp. ISBN: 981-3083-69-7.
2. I. Antoniou, C.S. Calude, M.J. Dinneen (eds.). *Unconventional Models of Computation, UMC'2K*, Springer, London, December 2000, xi + 301 pp. ISBN 1-85233-417-0.
3. C.S. Calude, M.J. Dinneen, F. Peper (eds.). *Third International Conference, UMC 2002*, Proceedings Lecture Notes in Computer Science, Vol. 2509, Springer, Heidelberg, 2002, vii + 331 pp. ISBN: 3-540-44311-8.
4. C.S. Calude, M.J. Dinneen, M.J. Pérez-Jiménez, Gh. Păun, G. Rozenberg (eds.). *Proc. 4th International Conference Unconventional Computation*, Lecture Notes in Computer Science, Vol. 3699, Springer, Heidelberg, 2005, xi + 267 pp. ISBN: 3-540-29100-8.

The Steering Committee of the series of International Conferences on Unconventional Computation includes T. Bäck (Leiden, The Netherlands), C.S. Calude (Auckland, New Zealand, Co-chair), L.K. Grover (Murray Hill, NJ, USA), J. van Leeuwen (Utrecht, The Netherlands), S. Lloyd (Cambridge, MA, USA), Gh. Păun

(Bucharest, Romania, and Seville, Spain), T. Toffoli (Boston, MA, USA), C. Torras (Barcelona, Spain), G. Rozenberg (Leiden, The Netherlands, and Boulder, Colorado, USA, Co-chair), A. Salomaa (Turku, Finland).

The five key-note speakers of the conference were:

1. Gerard Dreyfus (ESPCI, Paris, France): *Graph Machines and Their Applications to Computer-Aided Drug Design: A New Approach to Learning from Structured Data*
2. Michael C. Mozer (Department of Computer Science, and Institute of Cogntive Science, University of Colorado, USA): *Rational Models of Cognitive Control*
3. Reidun Twarock (University of York, UK): *Self-Assembly in Viruses*
4. Erik Winfree (Computer Science and Computation & Neural Systems, California Institute of Technology): *Fault-Tolerance in Biochemical Systems*
5. Damien Woods (University College Cork, Ireland): *Optical Computing and Computational Complexity*

UC 2006 included the following tutorials:

1. Andrew Adamatzky, Benjamin De Lacy Costello, Tetsuya Asai (Computing, Engineering and Mathematical Sciences, University of the West of England, Bristol, UK): *Reaction-Diffusion Computers*
2. Cristian S. Calude (University of Auckland, New Zealand): *Computing with Randomness*
3. Nataša Jonoska (University of South Florida, USA), and Darko Stefanovic (University of New Mexico, USA): *Biomolecular Automata*
4. Viv Kendon (University of Leeds, UK): *Quantum Computing*
5. Joseé del R. Millán (Institute for Systems, Informatics and Safety Joint Research Centre, Ispra, Italy): *Brain Signal Analysis*
6. Christof Teuscher (LANL, USA): *To Compute, or not to Compute*

The workshop "From Utopian to Genuine Unconventional Computers" was part of this year's conference.

The Programme Committee thanks the much appreciated work done by the paper reviewers for the conference. These experts were: Nevil Brownlee, Sam Braunstein, Douglas S. Bridges, Matteo Cavaliere, Cristian S. Calude, S. Barry Cooper, Jack Copeland, David Corne, Gabor Csardi, Erzsebet Csuhaj-Varjú, Michael J. Dinneen, Peter Erdi, Marian Gheorghe, Georgy Gimel'farb, James Goodman, Jozef Gruska, Oscar H. Ibarra, Mario de Jesus Pérez-Jiménez, Nataša Jonoska, Jarko Kari, Jan van Leeuwen, Chang Li, Rossella Lupacchini, Joseé del R. Millán, Pablo Moscato, Andrei Păun, Gheorghe Păun, Ion Petre, Vladimir Rogojin, Ulrich Speidel, Susan Stepney, Karl Svozil, Carme Torras, Christof Teuscher, Hiroshi Umeo.

The Programme Committee consisting of J.-P. Banâtre (Rennes, France), S. Braunstein (York, UK), C.S. Calude (Auckland, New Zealand, Co-chair), B. Cooper (Leeds, UK), D. Corne (Exeter, UK), M.J. Dinneen (Auckland, New Zealand, Secretary), P. Érdi (Kalamazoo, MI, USA), E. Goles (Santiago, Chile),

N. Jonoska (Tampa, FL, USA), J. Kari (Turku, Finland), J. van Leeuwen (Utrecht, Netherlands), R. Lupacchini (Bologna, Italy), J. del R. Millan (Ispra, Italy), Gh. Păun (Bucharest, Romania, and Seville, Spain, Co-chair), M.J. Pérez-Jiménez (Seville, Spain), I. Petre (Turku, Finland), P. Prusinkiewicz (Calgary, Canada), C. Teuscher (LANL, Los Alamos, USA), C. Torras (Barcelona, Spain), H. Umeo (Osaka, Japan), S. Stepney (York, UK), K. Svozil (Vienna, Austria), selected 17 papers (out of 36) to be presented as regular contributions.

We extend our thanks to all members of the Conference Committee, particularly to L. Caves, E. Clark, K. Clegg, G. Danks, O. Leyser (Co-chair), F. Polack, S. Stepney (Co-chair), J. Timmis, H. Turner, A. Weeks, J. Wright, for their invaluable organizational work.

We thank the University of York and the Centre for Discrete Mathematics of the University of Auckland for their technical support. The hospitality of our hosts, the Department of Computer Science of the University of York, is much appreciated.

The conference was partially supported by the Department of Biology of the University of York, the Enterprise and Innovation office of of the University of York, Microsoft Research, EPSRC, and the University consortium "White Rose"; we extend to all our gratitude.

It is a great pleasure to acknowledge the fine cooperation with the *Lecture Notes in Computer Science* team of Springer for producing this volume in time for the conference.

June 2006

C.S. Calude
M.J. Dinneen
Gh. Păun
G. Rozenberg
S. Stepney

Table of Contents

Invited Papers

Graph Machines and Their Applications to Computer-Aided Drug Design: A New Approach to Learning from Structured Data
Aurélie Goulon, Arthur Duprat, Gérard Dreyfus 1

Rational Models of Cognitive Control
Michael C. Mozer 20

Fault-Tolerance in Biochemical Systems
Erik Winfree 26

Optical Computing and Computational Complexity
Damien Woods 27

Regular Papers

If a Tree Casts a Shadow Is It Telling the Time?
Russ Abbott 41

Peptide Computing – Universality and Theoretical Model
M. Sakthi Balan, Helmut Jürgensen 57

Handling Markov Chains with Membrane Computing
Mónica Cardona, M. Angels Colomer, Mario J. Pérez-Jiménez, Alba Zaragoza 72

Approximation Classes for Real Number Optimization Problems
Uffe Flarup, Klaus Meer 86

Physical Systems as Constructive Logics
Peter Hines 101

On Spiking Neural P Systems and Partially Blind Counter Machines
Oscar H. Ibarra, Sara Woodworth, Fang Yu, Andrei Păun 113

Chemical Information Processing Devices Constructed Using a Nonlinear Medium with Controlled Excitability
Yasuhiro Igarashi, Jerzy Gorecki, Joanna Natalia Gorecka 130

Flexible Versus Rigid Tile Assembly
Nataša Jonoska, Gregory L. McColm 139

On Pure Catalytic P Systems
Shankara Narayanan Krishna 152

Mapping Non-conventional Extensions of Genetic Programming
W.B. Langdon 166

The Number of Orbits of Periodic Box-Ball Systems
Akihiro Mikoda, Shuichi Inokuchi, Yoshihiro Mizoguchi, Mitsuhiko Fujio 181

The Euclid Abstract Machine: Trisection of the Angle and the Halting Problem
Jerzy Mycka, Francisco Coelho, José Félix Costa 195

$1/f$ Noise in Elementary Cellular Automaton Rule 110
Shigeru Ninagawa 207

A Light-Based Device for Solving the Hamiltonian Path Problem
Mihai Oltean 217

Optimizing Potential Information Transfer with Self-referential Memory
Mikhail Prokopenko, Daniel Polani, Peter Wang 228

On the Power of Bio-Turing Machines
H. Ramesh, Shankara Narayanan Krishna, Raghavan Rama 243

Ergodic Dynamics for Large-Scale Distributed Robot Systems
Dylan A. Shell, Maja J. Matarić 254

Author Index 267

Graph Machines and Their Applications to Computer-Aided Drug Design: A New Approach to Learning from Structured Data

Aurélie Goulon[1], Arthur Duprat[1, 2], and Gérard Dreyfus[1]

[1] Laboratoire d'Électronique,
[2] Laboratoire de Chimie Organique, (CNRS UMR 7084)
École Supérieure de Physique et de Chimie Industrielles de la Ville de Paris
(ESPCI-ParisTech)
10 rue Vauquelin, 75005 PARIS, France
Aurelie.Goulon@espci.fr, Arthur.Duprat@espci.fr,
Gerard.Dreyfus@espci.fr

Abstract. The recent developments of statistical learning focused on *vector machines,* which learn from examples that are described by vectors of features. However, there are many fields where structured data must be handled; therefore, it would be desirable to learn from examples described by *graphs. Graph machines* learn real numbers from graphs. Basically, for each graph, a separate learning machine is built, whose algebraic structure contains the same information as the graph. We describe the training of such machines, and show that virtual leave-one-out, a powerful method for assessing the generalization capabilities of conventional vector machines, can be extended to graph machines. Academic examples are described, together with applications to the prediction of pharmaceutical activities of molecules and to the classification of properties; the potential of graph machines for computer-aided drug design are highlighted.

1 Introduction

Whether neural networks still fall in the category of "unconventional" computational methods is a debatable question, since that technique is well understood and widely used at present; its advantages over conventional regression methods are well documented and mathematically proven. Neural networks are indeed conventional in that they learn from *vector* data: typically, the variables of the neural model are in the form of a vector of numbers. Therefore, before applying learning techniques to neural networks, or any other conventional learning machine (Support Vector Machine, polynomial, multilinear model, etc.), the available data must be turned into a vector of variables; the learning machine then performs a mapping of a set of input vectors to a set of output vectors. In most cases, the output is actually a scalar, so that the mapping is from $\mathbb{R}^n$ to $\mathbb{R}$, where n is the dimension of the input vectors. When modeling a physical process for instance, the factors that have an influence on the quantity to be modeled are known from prior analysis, so that the construction of the vector of

C.S. Calude et al. (Eds.): UC 2006, LNCS 4135, pp. 1 – 19, 2006.

variables is straightforward, requiring simply normalization, and possibly variable selection by statistical methods.

In many cases of interest, however, encoding the data into a vector cannot be performed without information loss. Such is the case whenever the information to be learnt from is structured, i.e. is naturally encoded into a graph. In scene analysis for instance, a scene can be encoded into a graph that describes the relationships between the different parts of the scene. In computer-aided drug design, the purpose of learning is a mapping of the space of molecules to the space of pharmaceutical activities; in most cases, the structure of the molecule explains, to a large extent, its activity. Since molecular structures are readily described by graphs, QSAR (Quantitative Structure-Activity Relationships) aims at mapping the space of the graphs of molecular structures to the space of molecular activities or properties.

In the present paper, we describe an approach to learning that can be termed unconventional insofar as its purpose is a mapping of graphs to real numbers (or vectors) instead of a mapping of vectors to real numbers. The latter quantities may be either real-valued (graph regression) or binary (graph classification). The idea of learning from graphs (and generally structured data) can be traced back to the early days of machine learning, when Recursive Auto-Associative Memories (RAAMs) were designed for providing compact representations of trees [1]. It evolved subsequently to Labeled RAAMs [2], recursive networks [3], and graph machines (for a review of the development of numerical machine-learning from structured data, see [4]).

The first part of the paper is devoted to a description of graph machines and of some didactic, toy problems. It will also be shown that model selection methods that are proved to be efficient for conventional machine learning can be extended to graph machines. The second part of the paper will describe novel applications of graph machines to the prediction and classification of the properties or activities of molecules, a research area known as QSAR/QSPR (Quantitative Structure-Activity/Structure-Property Relationships). We show that graph machines are particularly powerful in that area, because they avoid a major problem of that field: the design, computation and selection of molecular descriptors.

2 Graph Machines

We first provide the definitions and notations for handling acyclic graphs, and the construction of graph machines from general graphs (possibly cyclic). Academic problems are described as illustrations. It is shown that the training and model selection methods developed for vector machines can be extended to graph machines.

2.1 Handling Directed Acyclic Graphs

Definitions: we consider the mapping from a set of acyclic graphs G to a set of real-valued numbers.

For each acyclic graph G_i of G, a parameterized function $g^i \ \mathbb{R}^n \to \mathbb{R}$ is constructed, which is intended (i) to encode the structure of the graph [5], and (ii) to provide a prediction of the quantity of interest, e.g. a property or an activity of the molecule, from G_i. It is constructed as follows. A parameterized function $f_{\boldsymbol{\theta}}$ ("node function") is

associated to each node. $\boldsymbol{\theta}$ denotes the vector of parameters of the node function. All nodes, except the root node, have the same node function $f_{\boldsymbol{\theta}}$; those functions are combined in such a way that g^i has the same structure as graph G_i: if an edge from node k to node l exists in the graph, then the value of the node function associated to node k is a variable of the node function associated to node l. The root node may be assigned a different function, denoted by $F_{\boldsymbol{\Theta}}$, where $\boldsymbol{\Theta}$ is the vector of parameters of $F_{\boldsymbol{\Theta}}$. If the node functions are neural networks, the g^i's are termed recursive neural networks [3].

Notations: the following notations are used throughout the paper.
We denote by $\mathbf{x}_j$ the (optional) vector of labels that provide information about node j of graph G_i. The size of the label vector is denoted by X_i; it is the same for all nodes of a given graph. Therefore, the parameterized function associated to G_i will be denoted as $g^i_{\boldsymbol{\theta},\boldsymbol{\Theta}}\left(\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_{v_i}\right)$, where v_i is the number of nodes of graph G_i. If no specific information about the node is necessary, $g^i_{\boldsymbol{\theta},\boldsymbol{\Theta}}$ has no variable: its value depends only on the structure of graph G_i.

We denote by $\mathbf{z}_j$ the vector of variables of the node function $f_{\boldsymbol{\theta}}(\mathbf{z}_j)$ of the non-root node j of graph G_i. Denoting by d_j the in-degree of non-root node j, and defining $M_i = \arg\max_j d_j$, the size of vector $\mathbf{z}_j$ is equal to $D_i = M_i + X_i + 1$. The vectors of variables of the node functions $f_{\boldsymbol{\theta}}(\mathbf{z}_j)$ are constructed as follows: for all j, the first component z_j^0 is equal to 1 (the "bias" if $f_{\boldsymbol{\theta}}(\mathbf{z}_j)$ is a neural network, the constant term if $f_{\boldsymbol{\theta}}(\mathbf{z}_j)$ is an affine function); for node j, of in-degree d_j, components z_j^1 to $z_j^{d_j}$ are the values of the node functions assigned to the parent nodes of node j; if $d_j < M_i$, components $z_j^{d_j+1}$ to $z_j^{M_i}$ are equal to zero; if $X_i \neq 0$, components $z_j^{M_i+1}$ to $z_j^{M_i+X_i}$ are the labels of node j.

We denote by $\mathbf{y}_i$ the vector of variables of the node function $F_{\boldsymbol{\Theta}}\left(\mathbf{y}_i\right)$ of the root node of graph G_i. The size of $\mathbf{y}_i$ is $\Delta_i = d_r + X_r + 1$, where d_r denotes the in-degree of the root node and X_r the size of its vector of labels. y_i^0 is equal to 1 (bias), y_i^1 to $y_i^{d_r}$ are the values of the node functions assigned to the parent nodes of the root node, $y_i^{d_r+1}$ to $y_i^{d_r+X_r}$ are the labels assigned to the root node.

As an example, Fig. 1 shows an acyclic graph G_1 with maximum in-degree $M_1 = 2$; the corresponding graph machine is:

$$g^1_{\boldsymbol{\theta},\boldsymbol{\Theta}}\left(\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_8\right) = F_{\boldsymbol{\Theta}}\left(\mathbf{y}_1\right) = F_{\boldsymbol{\Theta}}\left(\mathbf{x}_8, f_{\boldsymbol{\theta}}\left(\mathbf{x}_7, f_{\boldsymbol{\theta}}\left(\mathbf{z}_6\right), 0\right), f_{\boldsymbol{\theta}}\left(\mathbf{x}_5, f_{\boldsymbol{\theta}}\left(\mathbf{z}_4\right), f_{\boldsymbol{\theta}}\left(\mathbf{x}_3, f_{\boldsymbol{\theta}}\left(\mathbf{z}_2\right), f_{\boldsymbol{\theta}}\left(\mathbf{z}_1\right)\right)\right)\right)\right) \quad (1)$$

If no information about the nodes is required by the problem at hand ($X_1 = 0$), one has $D = 3$, and:

$$\mathbf{z}_1 = \mathbf{z}_2 = \mathbf{z}_4 = \mathbf{z}_6 = \begin{pmatrix} 1 & 0 & 0 \end{pmatrix}^{\mathrm{T}},\ \mathbf{z}_3 = \begin{pmatrix} 1 & f_{\boldsymbol{\theta}}\left(\mathbf{z}_1\right) & f_{\boldsymbol{\theta}}\left(\mathbf{z}_2\right) \end{pmatrix}^{\mathrm{T}},$$

$$\mathbf{z}_5 = \begin{pmatrix} 1 & f_{\boldsymbol{\theta}}\left(\mathbf{z}_3\right) & f_{\boldsymbol{\theta}}\left(\mathbf{z}_4\right) \end{pmatrix}^{\mathrm{T}},\ \mathbf{z}_7 = \begin{pmatrix} 1 & f_{\boldsymbol{\theta}}\left(\mathbf{z}_6\right) & 0 \end{pmatrix}^{\mathrm{T}},\ \mathbf{y}_1 = \begin{pmatrix} 1 & f_{\boldsymbol{\theta}}\left(\mathbf{z}_5\right) & f_{\boldsymbol{\theta}}\left(\mathbf{z}_7\right) \end{pmatrix}^{\mathrm{T}}.$$

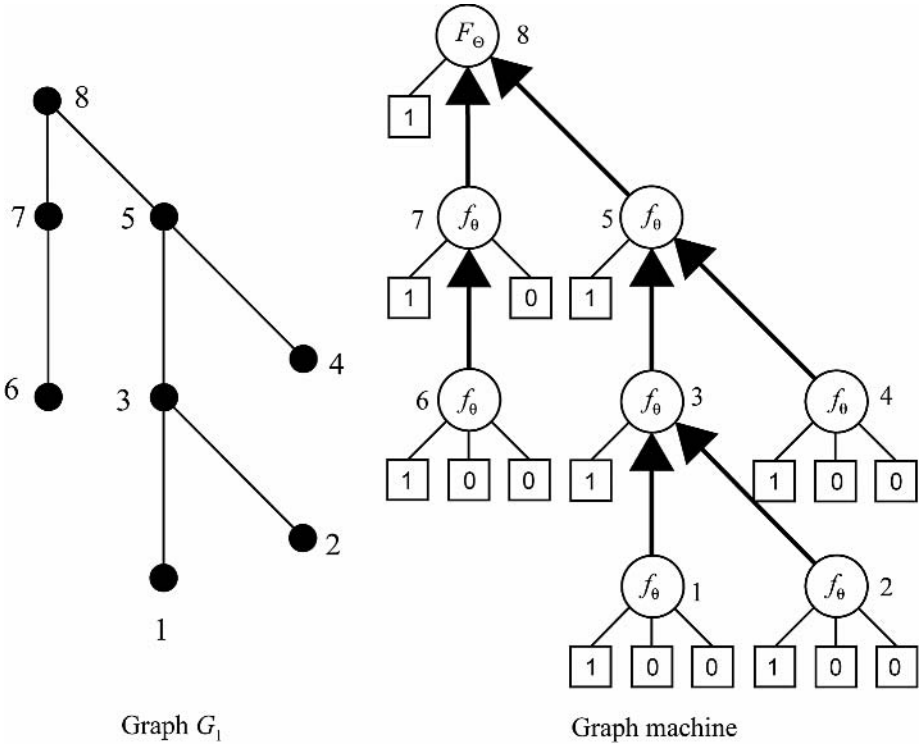


Fig. 1. An acyclic graph and its graph machine

2.2 Cyclic Graphs

Graph machines handle cyclic graphs and parallel edges. To that effect, the initial graph is preprocessed by deleting a number of edges equal to the number of cycles, and all parallel edges but one; moreover, a label is assigned to each node: it is equal to the degree of the node, thereby retaining the information about the original graph structure. Finally, a root node is chosen and the edges are assigned orientations, according to an algorithm described in [6].

2.3 The Training of Graph Machines

Graph machines are trained in the usual framework of empirical risk minimization. A cost function $J(\boldsymbol{\Theta},\boldsymbol{\theta})$ is defined, and its minimum with respect to the parameters is sought, given the available training data. The cost function takes into account the discrepancy between the predictions of the models and the observations present in the training set, and may include regularization terms, e.g.:

$$J(\boldsymbol{\theta},\boldsymbol{\Theta}) = \sum_{i=1}^{N} \left(y^i - g^i_{\boldsymbol{\theta},\boldsymbol{\Theta}} \right)^2 + \lambda_1 \left\| \boldsymbol{\theta} \right\| + \lambda_2 \left\| \boldsymbol{\Theta} \right\|, \tag{2}$$

where N is the size of the training set, y^i is the value of the i-th observation of the quantity to be modeled, and λ_1 and λ_2 are suitably chosen regularization constants.

Since the parameter vectors $\boldsymbol{\theta}$ and $\boldsymbol{\Theta}$ must be identical within each function g^i and across all those functions, one must resort to the so-called *shared weight* trick; the k-th component of the gradient of the cost function can be written as

$$\frac{\partial J(\boldsymbol{\theta},\boldsymbol{\Theta})}{\partial \theta_k} = \sum_{i=1}^{N} \frac{\partial J^i}{\partial \theta_k}, \tag{3}$$

where J^i is the contribution of example i to the cost function. We denote by $n^i_{\theta_k}$ the number of occurrences of parameter θ_k in acyclic graph G_i; if the root is assigned the same parameterized function as the other nodes, then $n^i_{\theta_k}$ is equal to the number of nodes in graph G_i. The shared weight trick consists in setting

$$\frac{\partial J^i}{\partial \theta_k} = \sum_{j=1}^{n^i_{\theta_k}} \frac{\partial J^i}{\partial \theta_{k_j}}, \tag{4}$$

so that one has finally:

$$\frac{\partial J(\boldsymbol{\theta},\boldsymbol{\Theta})}{\partial \theta_k} = \sum_{i=1}^{N} \sum_{j=1}^{n^i_{\theta_k}} \frac{\partial J^i}{\partial \theta_{k_j}}. \tag{5}$$

Relation (5) is subsequently used for minimizing cost function (2) by any suitable gradient descent algorithm (Levenberg-Marquardt, BFGS, conjugate gradient, …).

If functions $f_{\boldsymbol{\theta}}$ and $F_{\boldsymbol{\Theta}}$ are neural networks, the usual backpropagation algorithm may be conveniently used for computing the gradient; otherwise, one resorts to numerical estimations thereof.

2.4 Didactic Examples: Learning the Number of Nodes and the Number of Cycles of a Graph

In the present section, two simple examples are provided, whose solutions can be worked out analytically because they are linear. In both cases, we consider the training set made of three graphs, shown on Fig. 2.

Learning the number of nodes of a graph: first, assume that it is desired to learn, from examples, the number of nodes of a graph. Then the desired mapping is: $G_1 \rightarrow 4$; $G_2 \rightarrow 8$; $G_3 \rightarrow 9$. Moreover, generalization should be performed by using the node functions thus obtained in any other graph machine, i.e. to compute the number of nodes of any graph.

The first step consists in constructing directed acyclic graphs (DAGs) from the initial graphs. The construction of the DAGs is obvious for G_1 and G_2. Since graph G_3 has four cycles, four edges must be deleted. Fig. 3 shows the directed acyclic graphs on which the graph machines will be based.

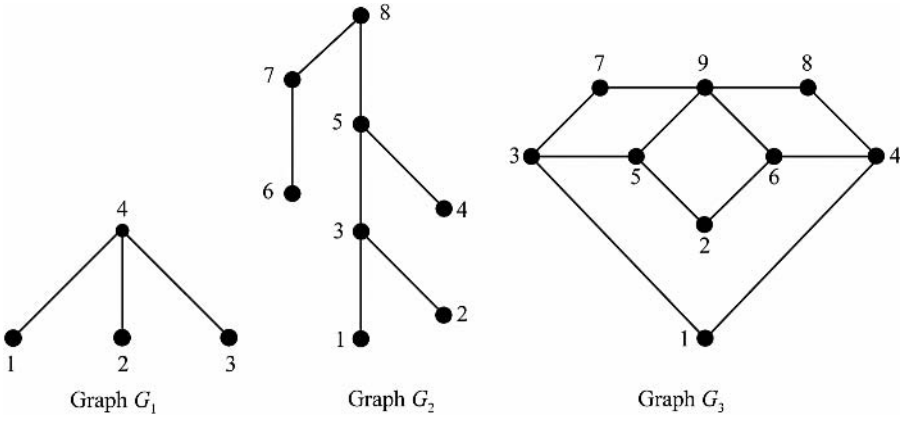


Fig. 2. A training set

The node function $f_{\boldsymbol{\theta}}$ is sought in the family of affine functions $f_{\boldsymbol{\theta}}(\mathbf{z}) = \sum_{j=0}^{D-1} \theta_j z_j$, and $F_{\boldsymbol{\Theta}}$ is taken identical to $f_{\boldsymbol{\theta}}$. Since the presence or absence of an edge is irrelevant for the computation of the number of nodes, no label is necessary: $X_1 = X_2 = X_3 = 0$. The node functions being the same for all graphs of the training set, we take $D = \max_i M_i + 1 = 5$. Since all edges are equivalent, one has $\theta_1 = \theta_2 = \theta_3 = \theta_4 = \theta$. Therefore, there are actually two independent parameters only.

The obvious solution is $\theta_0 = \theta = 1$. For graph G_1 for instance, one has:

$$g^1_{\boldsymbol{\theta},\boldsymbol{\Theta}}(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4) = f_{\boldsymbol{\theta}}(1, f_{\boldsymbol{\theta}}(\mathbf{z}_1), f_{\boldsymbol{\theta}}(\mathbf{z}_2), f_{\boldsymbol{\theta}}(\mathbf{z}_3), 0) = \theta_0 + 3\theta\theta_0 = 4,$$

where $\mathbf{z}_1 = \mathbf{z}_2 = \mathbf{z}_3 = (1 \quad 0 \quad 0 \quad 0 \quad 0)^{\mathrm{T}}$.

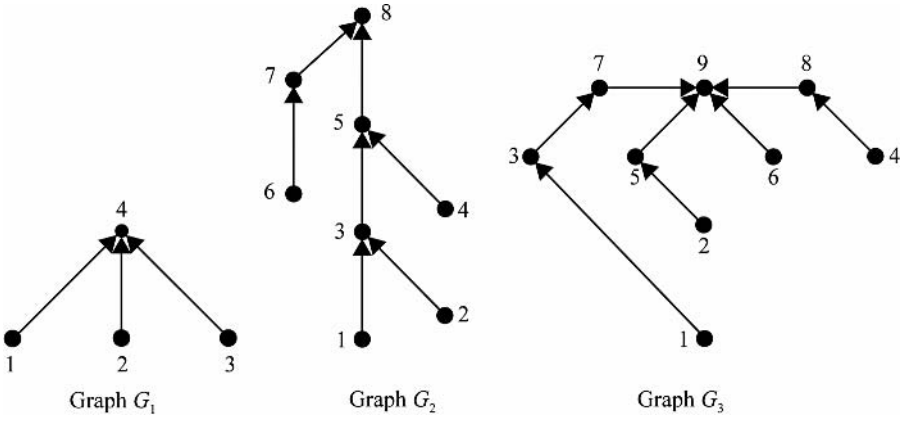


Fig. 3. The acyclic graphs derived from the training set shown on Fig. 2

Similarly, one has, for graph G_2: $g^2_{\boldsymbol{\theta},\boldsymbol{\Theta}}\left(\mathbf{x}_1,\mathbf{x}_2,...,\mathbf{x}_8\right)=2\theta_0\left(1+\theta+\theta^2+\theta^3\right)=8$, and, for graph G_3: $g^3_{\boldsymbol{\theta},\boldsymbol{\Theta}}\left(\mathbf{x}_1,\mathbf{x}_2,...,\mathbf{x}_9\right)=\theta_0\left(1+4\theta+3\theta^2+\theta^3\right)=9$.

Learning the number of cycles in a graph: similarly, consider learning, from examples, the number of cycles in a graph.

By contrast to the previous example, the presence or absence of edges is highly relevant, so that each node must be labeled by its degree: $X_1 = X_2 = X_3 = 1$. Therefore, one has $D=\max_i M_i+2=5$.

$f_{\boldsymbol{\theta}}$ is sought in the family of affine functions $f_{\boldsymbol{\theta}}\left(\mathbf{z}\right)=\sum_{j=0}^{D-1}\theta_j z_j$, and the root node is assigned a different affine function $F_{\boldsymbol{\Theta}}\left(\mathbf{y}_i\right)$. Therefore, the size of vectors $\mathbf{y}_i$ is $\max_i \Delta_i+1=7$, because the present problem requires an additional label, equal to 1, for the root nodes.

An obvious solution to the problem is the following: θ_0 = -1, $\theta_1 = \theta_2 = \theta_3 = 1$, $\theta_4 = 1/2$, Θ_0 = -1, $\Theta_1 = \Theta_2 = \Theta_3 = \Theta_4 = 1$, $\Theta_5 = 1/2$, $\Theta_6 = 1$; the additional label assigned to the root node is equal to 1.

Consider graph G_1: $f_{\boldsymbol{\theta}}(\mathbf{z}_1) = f_{\boldsymbol{\theta}}(\mathbf{z}_2) = f_{\boldsymbol{\theta}}(\mathbf{z}_3)$ = -1/2, $\mathbf{y}_1=\begin{pmatrix}1 & -1/2 & -1/2 & -1/2 & 0 & 3 & 1\end{pmatrix}^{\mathrm{T}}$, so that: $F_{\boldsymbol{\Theta}}(\mathbf{y}_1)$ = –1 – 3/2 + 3/2 + 1 = 0, as expected.

Similarly, for graph G_3, one has: $f_{\boldsymbol{\theta}}(\mathbf{z}_1) = f_{\boldsymbol{\theta}}(\mathbf{z}_2) = 0$, $f_{\boldsymbol{\theta}}(\mathbf{z}_4) = f_{\boldsymbol{\theta}}(\mathbf{z}_6) = 1/2$, $f_{\boldsymbol{\theta}}(\mathbf{z}_3) = f_{\boldsymbol{\theta}}(\mathbf{z}_5) = 1/2$, $f_{\boldsymbol{\theta}}(\mathbf{z}_7) = f_{\boldsymbol{\theta}}(\mathbf{z}_8) = 1/2$, $\mathbf{y}_3=\begin{pmatrix}1 & 1/2 & 1/2 & 1/2 & 1/2 & 4 & 1\end{pmatrix}^{\mathrm{T}}$, hence $F_{\boldsymbol{\Theta}}(\mathbf{y}_3) = 4$.

2.5 Some Nonlinear Learning Tasks

The above two problems have linear solutions that can be obtained by inspection. In general, graph regression or classification problems cannot be solved in the framework of linear models, so that one has to resort to training, as described above. The two examples described below are examples of graph machines being trained to learn graph properties as in the previous section, but the solutions are not linear. A database of 150 randomly generated graphs, featuring 2 to 15 nodes and 0 to 9 cycles, was created. Model selection was performed by cross-validation.

Learning the diameter of a graph: the diameter of a graph is the length of the shortest path between its most distant nodes:

$$D=\max_{u,v} d(u,v), \tag{6}$$

where $d(u, v)$ is the distance (the length of the shortest path) between nodes u and v. In the database under investigation, the index ranges from 1 to 9. That is clearly a non-linear property; therefore, the node function was a neural network; model selection resulted in a neural network with four hidden neurons. The root mean square

error on the training set is 0.36, and the root mean square validation error (10-fold cross-validation) is 0.53. Since the index is an integer ranging from 1 to 9, the prediction is excellent given the complexity of the graphs.

Learning the Wiener index of a graph: the Wiener index of a graph G is the sum of the distances between its vertices. That index was first defined by H. Wiener [7], in order to investigate the relationships between the structure of chemicals and their properties. It is defined as:

$$W(G) = \frac{1}{2}\sum_{u,v} d(u,v)\,. \tag{7}$$

In our database, that index ranges from 1 to 426.

Model selection by 10-fold cross-validation resulted in a 4-hidden neuron node function, leading to a RMS validation error of 7.9. The scatter plot is shown on Fig. 4, illustrating the accuracy of the results obtained by training *without having to compute any feature for describing the graph structure*. The problem of feature design and selection, which is central in conventional machine learning with vector machines, is irrelevant for graph machines. This is very important for the applications described below.

2.6 Model Selection

Similarly to vector machines, usual model selection techniques such as hold-out, K-fold cross-validation, leave-one-out, can be applied to recursive networks and to graph machines. In the present section, we show how virtual leave-one-out, a powerful method for estimating the generalization capability of a vector machine, can be extended to graph machines.

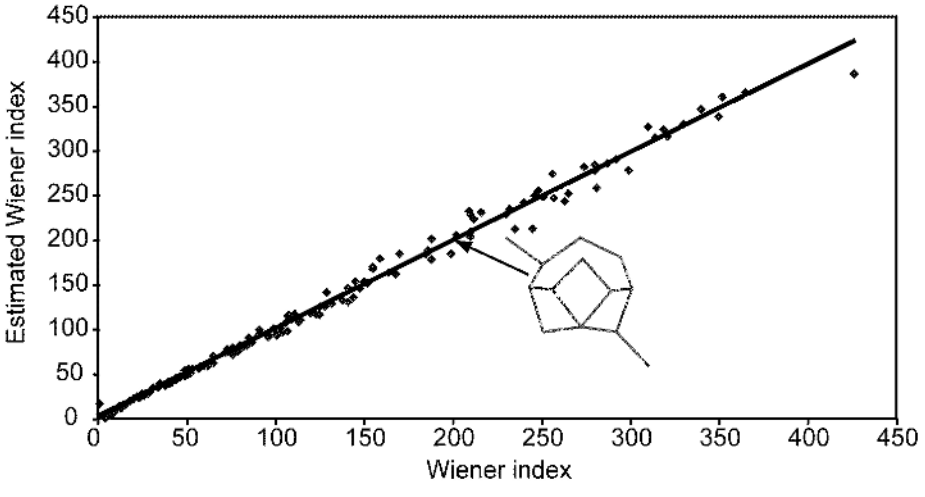


Fig. 4. Learning the Wiener index of graphs

Virtual leave-one-out for vector machines: leave-one-out is known to provide an unbiased estimation of the generalization error of a model [8]. However, it is very demanding in terms of computation time: it involves training N different models, where N is the number of examples; each model is trained from $N - 1$ examples, and the modeling error on the left-out example is computed; the estimator of the generalization error is

$$\sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(R_i^{-i}\right)^2}\,, \tag{8}$$

where R_i^{-i} is the modeling error on example i when the latter is left out of the training set.

Virtual leave-one-out is a very effective method for obtaining an approximation of the above estimate [9], [10]. It consists in training the candidate model with all examples, and computing the virtual leave-one-out score as:

$$\sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(\frac{R_i}{1-h_{ii}}\right)^2}\,, \tag{9}$$

where R_i is the modeling error on example i when the latter is in the training set. h_{ii} is the *tangent-plane leverage* of example i*:* it is the i-th diagonal element of the *hat matrix*:

$$\mathbf{H} = \mathbf{Z}\left(\mathbf{Z}^{\mathrm{T}}\mathbf{Z}\right)^{-1}\mathbf{Z}^{\mathrm{T}}\,. \tag{10}$$

$\mathbf{Z}$ is the Jacobian matrix of the model, i.e. the matrix whose columns are the values of the partial derivative of the model $g_{\boldsymbol{\theta}}(\mathbf{x})$ with respect to its parameters, for the examples of the training set:

$$z_{ij} = \left(\frac{\partial g_{\boldsymbol{\theta}}\left(\mathbf{x}\right)}{\partial \theta_j}\right)_{\mathbf{x}=\mathbf{x}_i}\,. \tag{11}$$

For models that are linear in their parameters, relation (9) is exact: it is known as the PRESS (Predicted REsidual Sum of Squares) statistics. For models that are not linear in their parameters, such as neural networks, it is a first-order approximation of the estimator.

Leverages have the following properties:

$$\begin{aligned} &0 < h_{ii} < 1 \\ &\sum_{i=1}^{N} h_{ii} = q \end{aligned} \tag{12}$$

where q is the number of parameters of the model. Therefore, the leverage of example i can be viewed as the proportion of the parameters of the model that is devoted to

fitting example *i*. If h_{ii} is on the order of 1, the model has devoted a large fraction of its parameters to fitting example *i*, so that the model is probably strongly overfitted to that example. Therefore, virtual leave-one-out is a powerful tool for overfitting detection and model selection.

Virtual leave-one-out for graph machines: in the present section, we show how virtual leave-one-out can be extended to graph machines. We give a simplified proof of the result, which provides a flavor of the full derivation. For simplicity, consider a model with a single parameter θ; moreover, assume that, for all graph machines, the node function of the root node is identical to the node function of non-root nodes $f_\theta(\mathrm{x})$. We denote by y_p^j the measured value of the property of interest for example *j*: the modeling error on example *j* is $R_j = y_p^j - g_\theta^j$; we denote by R_j^{-i} the modeling error on example *j* when example *i* has been withdrawn from the training set: $R_j^{-i} = y_p^j - g_{\theta^{-i}}^j$, where θ^{-i} is the parameter computed from the training set from which example *i* has been withdrawn. Therefore, one has:

$$R_j^{-i} = R_j + g_\theta^j - g_{\theta^{-i}}^j . \tag{13}$$

Assuming that the withdrawal of example *i* from the training set does not affect the parameters of the model to a large extent, a first-order Taylor expansion of the model, in parameter space, can be performed:

$$g_{\theta^{-i}}^j = g_\theta^j + \frac{\partial g_\theta^j}{\partial \theta}\left(\theta^{-i} - \theta\right). \tag{14}$$

Therefore, one has:

$$R_j^{-i} = R_j - \frac{\partial g_\theta^j}{\partial \theta}\left(\theta^{-i} - \theta\right). \tag{15}$$

The first derivative of the model can similarly be expanded to:

$$\frac{\partial g_{\theta^{-i}}^j}{\partial \theta} = \frac{\partial g_\theta^j}{\partial \theta} + \frac{\partial^2 g_\theta^j}{\partial \theta^2}\left(\theta^{-i} - \theta\right). \tag{16}$$

As defined above, the least squares cost function (without regularization terms), is

$$J\left(\theta\right) = \sum_{i=1}^{N}\left(y^i - g_\theta^i\right)^2 , \tag{17}$$

which is minimum for θ. Therefore, one has

$$0 = \frac{\partial J\left(\theta\right)}{\partial \theta} = \sum_j R_j \frac{\partial g_\theta^j}{\partial \theta} , \tag{18}$$

and, similarly

$$0 = \frac{\partial J^{-i}\left(\theta\right)}{\partial\theta} = \sum_{j\neq i} R_j^{-i} \frac{\partial g^j_{\theta^{-i}}}{\partial\theta}\,, \tag{19}$$

where J^i is the cost function after training with the dataset from which i was withdrawn. Substituting relations (15) and (16) into relation (19) gives:

$$0 = \sum_j R_j \frac{\partial g^j_\theta}{\partial\theta} - R_i \frac{\partial g^i_\theta}{\partial\theta} - \left[\sum_{j\neq i} \left(\frac{\partial g^j_\theta}{\partial\theta}\right)^2 - \sum_{j\neq i} R_j \frac{\partial^2 g^j_\theta}{\partial\theta^2}\right]\left(\theta^{-i} - \theta\right). \tag{20}$$

The first term on the right-hand side is equal to zero. Neglecting the second derivatives with respect to the squared first derivatives (the usual Levenberg-Marquardt approximation), one gets, to first order:

$$\left(\theta^{-i} - \theta\right) = -\frac{R_i \dfrac{\partial g^i_\theta}{\partial\theta}}{\left.\sum_{j\neq i} \dfrac{\partial g^j_\theta}{\partial\theta}\right)^2}\,. \tag{21}$$

Substituting into (15) with $j = i$, one obtains:

$$R_i^{-i} = \frac{R_i}{1 - h_{ii}}\,, \tag{22}$$

with:

$$h_{ii} = \frac{\left(\dfrac{\partial g^i_\theta}{\partial\theta}\right)^2}{\sum_j \left(\dfrac{\partial g^j_\theta}{\partial\theta}\right)^2}\,. \tag{23}$$

The above relation is similar to relation (10), which, for a single-parameter model, reduces to:

$$h_{ii} = \frac{\left(\dfrac{\partial g_\theta\left(\mathbf{x}\right)}{\partial\theta}\right)^2_{\mathbf{x}=\mathbf{x}_i}}{\sum_j \left(\dfrac{\partial g_\theta\left(\mathbf{x}\right)}{\partial\theta}\right)^2_{\mathbf{x}=\mathbf{x}_j}}\,. \tag{24}$$

Thus, virtual leave-one-out can provide an estimate of the generalization error of graph machines, in much the same way as for conventional vector machines: the

matrix whose general term is $z_{ij} = \frac{\partial g_{\boldsymbol{\theta}}^i}{\partial \theta_j}$ plays exactly the same role as the Jacobian matrix $\mathbf{Z}$ (of general term $z_{ij} = \left(\frac{\partial g_{\boldsymbol{\theta}}(\mathbf{x})}{\partial \theta_j} \right)_{\mathbf{x}=\mathbf{x}_i}$) for conventional vector machines. In the case of neural networks, it can easily be computed by backpropagating an error equal to ½, after the completion of training.

3 Graph Machines for the Prediction of Properties and/or Activities of Molecules

The prediction of the physico-chemical properties and pharmaceutical activities of molecules is a critical task in the drug industry for shortening the development times and costs. Typically, one synthesized molecule out of 10,000 becomes a commercial drug, and the development time of a new drug is approximately 10 years. Therefore, predicting the activity of a hitherto non-existent molecule may lead to tremendous savings in terms of development time and cost. Hence, over the past few years, QSPR/QSAR has become a major field of research in the chemical industry.

In a typical QSAR/QSPR scenario, a database of measured properties or activities of molecules is available, and it is desired to infer, from those data, the property/activity of molecules that have not yet been synthesized. Therefore, machine learning is a natural context for solving such problems. Linear, polynomial, neural, and SVM regression have been used extensively. For all such techniques, the design and the selection of relevant features, for the prediction of a given activity, are a major problem.

In the following, we show that graph machines solve that problem by exempting the model designer from finding and computing elaborate features, because the structure of the molecule is embodied into the learning machine itself. We show that, for the problems described here as well as for other problems, graph machines provide predictions that are at least as good as (and generally better than) predictions made by conventional machine learning, without the need for designing, computing and selecting features.

3.1 Encoding the Molecules

Molecules are usually described in databases in a representation called SMILES (Simplified Molecular Input Line Entry System), which provides a description of the graph structure of the molecule as a character string. In the applications described here, the functions $g_{\boldsymbol{\theta}}^i$ were generated from the SMILES files of the molecules by the following procedure: the molecules, described by these files, were converted into labeled graphs by the association of each non-hydrogen atom to a node, and each bond to an edge. The nodes were also assigned labels describing the atoms they were related to (e.g. their nature, their degree or stereoisomery …). Then, the adjacency matrices associated to those labeled graphs were generated. In the subsequent step,

the matrices were cast into a canonical form, by an algorithm that ranks the nodes according to criteria such as their degree, the fact that they belong to a cycle… [6]. This canonical form allowed the choice of the root nodes, and the conversion of the graphs into directed acyclic graphs. Fig. 5 illustrates the processing of a molecule from its SMILES representation into a directed acyclic graph.

Graph machines were then built for each graph of the data set; node functions were feedforward neural networks with a single layer of hidden neurons whose complexity (i.e. the number of neurons in the hidden layer) was controlled by cross-validation. The graph machines were then trained, with the shared weight condition, using the software package NeuroOne™[1], which computes the gradient of the cost function by backpropagation and minimizes the cost function by the Levenberg-Marquardt algorithm.

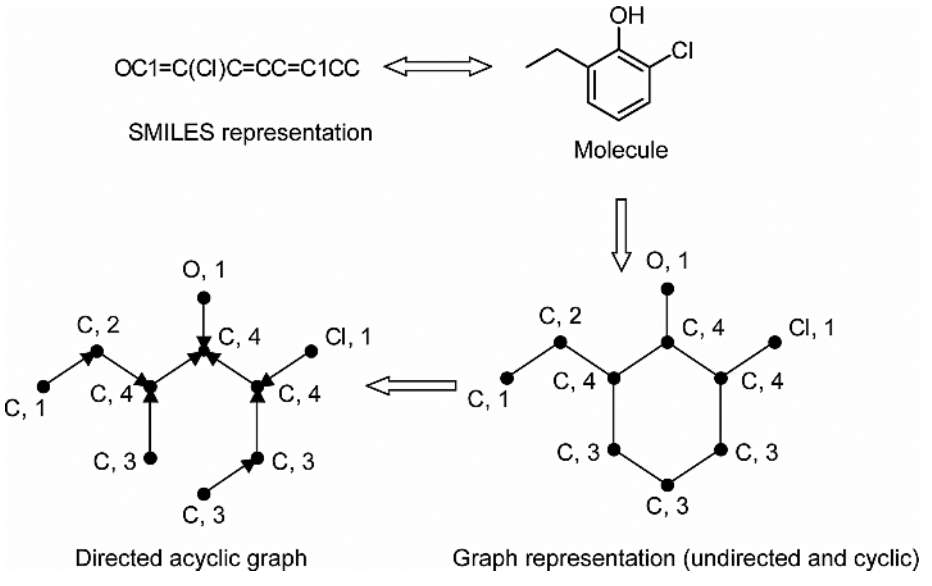


Fig. 5. Encoding a molecule into a graph machine

3.2 Predicting the Boiling Points of Halogenated Hydrocarbons

The volatility of halogenated hydrocarbons is an important property, because those compounds are widely used in the industry, for example as solvents, anaesthetics, blowing agents, and end up in the environment, where they can damage the ozone layer or be greenhouse gases. The volatility of a molecule can be assessed by its boiling point, a property measured only for a small proportion of possible halogenated hydrocarbons.

We studied a data set of 543 haloalkanes, whose boiling points were previously predicted by Multi Linear Regression (MLR) [11],[12]. This regression required the computation of numerous molecular descriptors, including arithmetic descriptors,

[1] NeuroOne is a product of Netral S.A. (http://www.netral.com)

topological indices, geometrical indices, and counts of substructures and fragments. The best feature subset was then selected, and generally included 6 or 7 descriptors. To provide a comparison with the results obtained by this method, we used the same training and test sets as Rücker et al. [12]. They feature 507 and 36 haloalkanes respectively, whose boiling points range from -128 °C to 249 °C.

In order to select the number of neurons required by the complexity of the problem, we first performed 10-fold cross-validation on the 507 examples of the training set. The results suggested the use of neural networks with 4 hidden neurons.

We then trained the selected graph machines, and predicted the boiling points of the test set molecules. The results of this study are shown in Table 1, where they are compared to the results obtained by Rücker et al. [12] on the same sets, using a 7-regressor MLR model. The predictions of the model on the test set are also displayed on Fig. 6.

Table 1. Prediction of the boiling points of haloalkanes by multi-linear regression and graph machines

	MLR (7-descriptor)		**4N-GM**	
	RMSE (°C)	R^2	RMSE (°C)	R^2
Training	6.607	0.9888	3.92	0.9960
10-fold CV	-	-	4.70	0.9942
Test	7.44	0.9859	5.07	0.9921

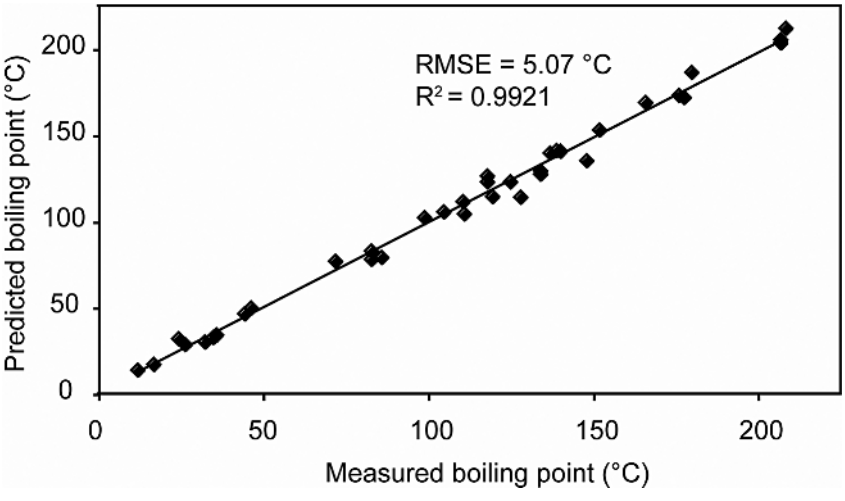


Fig. 6. Scatter plot for the prediction of the boiling point of 36 haloalkanes

The above results show that graph machines are able to model the boiling points of haloalkanes very well, without requiring the computation of any descriptor. Furthermore, this modeling task illustrates the fact that the design of the learning machine from the structure of the molecules avoids the loss of information caused by the selection of descriptors. Actually, Rücker et al. [12] stress the fact that the prediction of the boiling points of fluoroalkanes with their model is not satisfactory, which is presumably due to the lack of a descriptor taking into account the strong dipole interactions. The removal of 7 of these compounds decreased the training error of MLR regression by 0.56 °C (from 6.607 to 6.049 °C). By contrast, in the case of graph machines, the errors on those examples are not particularly high, and their removal from the database decreased the training error by 0.08 °C only (from 3.92 to 3.84 °C).

3.3 Predicting the Anti-HIV Activity of TIBO Derivatives

TIBO (Tetrahydroimidazobenzodiazepinone) derivatives are a family of chemicals with a potential anti-HIV activity. They belong to the category of non-nucleoside inhibitors, which block the reverse transcriptase of the retrovirus and prevent its duplication. We studied a data set of 73 of those compounds, whose activity was previously modeled with several QSAR methods, including conventional neural networks [13], multi-linear regression [14], comparative molecular field analysis (CoMFA) [15], and the substructural molecular fragments (SMF) method [16]. The latter approach is based on the representation of the molecules with graphs, which are split into fragments, whose contribution to the modeled activity is then computed by linear or non-linear regression. Those fragments are either atom-bond sequences, or "augmented atoms", defined as atoms with their nearest neighbours.

In order to compare the prediction abilities of graph machines to the performances of the SMF method [16], the data set was split into a training and a test set of 66 and 7 examples respectively, exactly as in [16]. The activity is expressed as $\log(1/IC_{50})$, where IC_{50} is the concentration leading to the inhibition of 50% of the HIV-1 reverse transcriptase enzyme. Since some compounds of the set are stereoisomers, a label that described the enantiomery (*R* or *S*) of the atoms was added when necessary.

We first performed 6-fold cross-validation on the training set with node functions having up to five hidden neurons to select the complexity of the model. Three hidden neurons provided the best cross-validation estimate of the generalization. The results obtained with this model are reported in Table 2 and on Fig. 7.

Table 2. Prediction of the activity of TIBO derivatives by different methods

	SMF		**3N GM**	
	RMSE	R^2	RMSE	R^2
Training set	0.89	0.854	0.28	0.9901
Test set	0.51	0.943	0.45	0.9255

Since the accuracies of the experimental values are not known, the prediction errors cannot be compared to the measurement errors. However, this study demonstrates again that graph machines compare favourably with other methods, without the requirement of computing descriptors. This task also illustrates another advantage of graph machines on some other methods: Solov'ev et al. [16] had to remove several compounds from their original set because they contained "rare" fragments, whereas this problem does not occur with graph machines, insofar as the molecules of the test set do not require labels (atom types or degrees for example) that are not present in the training set.

Additional results on the prediction of the toxicity of phenols, the anti-HIV activity of HEPT analogues, and the carcinogenicity of molecules, are described in [17].

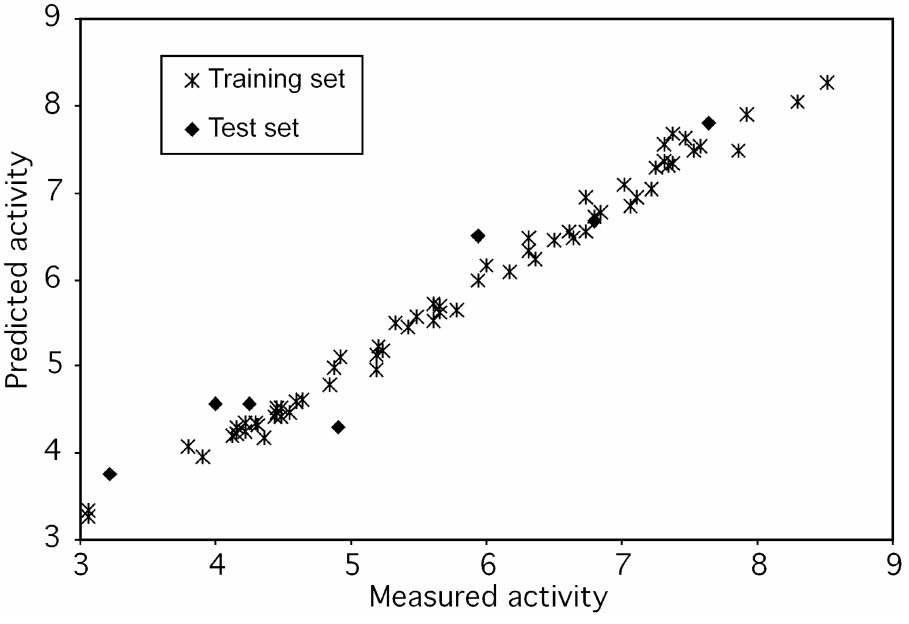


Fig. 7. Scatter plot of the prediction of the activity of TIBO derivatives

4 Graph Machines for Classification: Discriminating Aromatic From Non-aromatic Molecules

All the above examples are regression problems, in which a mapping is performed from graphs to real numbers. Graph machines can also perform classification, i.e. mappings from graphs to {-1, +1}. As an illustration, we show that discrimination between aromatic molecules (i.e. molecules that contain an aromatic cycle) and non-aromatic molecules can be performed. A cycle is aromatic when it fulfils several criteria: it must be planar, and have a set of conjugated π orbitals, thereby creating a delocalized π molecular orbital system. Furthermore, there must be $4n + 2$ electrons in this system, where n is an integer.

A set of 321 molecules was investigated, with various functional groups taken from [18]; it was divided into a training and a test set of 274 and 47 examples respectively. The proportion of molecules containing at least one aromatic cycle is shown on Fig. 8.

To select the number of hidden neurons required by this problem, 10-fold cross-validation was performed on the set of 273 examples. Table 3 reports the percentage of correct classification obtained with three and four hidden neurons.

The cross-validation error with the graph machines with 4 hidden neurons is due to a single misclassified example: the pipamperone, shown on Fig. 9. That error can be traced to the fact that the main part of the molecule is non-aromatic.

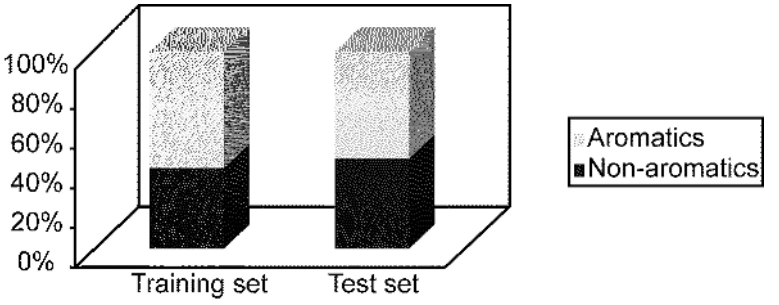


Fig. 8. Distribution of the molecules including at least one aromatic cycle in the data sets

Table 3. 10-fold cross-validation results for the prediction of the aromaticity

	Correct classification (training)	**Correct classification** (10-fold CV)
GM 3N	100%	100%
GM 4N	100%	99.7%

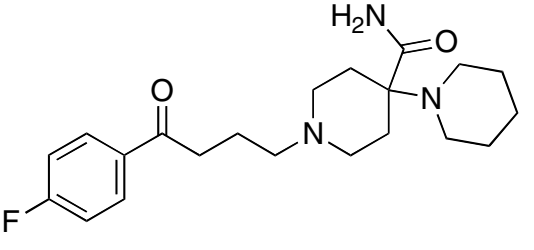


Fig. 9. Structure of the pipamperone, misclassified with 10-fold cross-validation

No example from the test set was misclassified by the graph machines with 3 hidden neurons, which illustrates the ability of graph machines to efficiently encode the structures of the graphs, thus to retain structure-related properties such as aromaticity.

5 Conclusions

A computational method that allows regression and classification on graphs has been described. Interestingly, it illustrates two principles that turn out to be very useful for solving real-life machine-learning tasks: (i) if a representation of the data for a given problem cannot be found "by hand", it may be advantageous to learn it; (ii) always embed as much prior information as possible into the structure of the learning machine. In agreement with statement (i), a representation of the graph evolves during training, as described in [5]; in agreement with statement (ii), the information about the structure of the data is embedded in the structure of the graph machine itself.

Model selection for recursive networks and graph machines used to be performed by conventional cross-validation, hold-out or leave-one-out; we have shown, in the present paper, that the powerful technique of virtual leave-one-out extends to recursive networks or graph machines in a relatively straightforward fashion; the full derivation of the results, and illustrations, will be provided in a forthcoming paper.

Applications of graph machines to computer-aided drug design have been described. The major asset of graph machines is their ability to make efficient predictions while exempting the model designer from the design, computation and selection of descriptors, which is recognized to be a major burden in QSAR/QSPR tasks. It should be noted, however, that, if the graph structure of the molecule is not sufficient for accurate prediction, descriptors could indeed be implemented as inputs to graph machines, in the form of labels for the nodes.

Scalability is an issue that should be investigated in a principled way. If the method is to be used for scene or text analysis for instance, very large corpuses must be handled. Experimental planning methods, allowing the model designer to use only the most informative data, have recently become available for nonlinear models in conventional machine learning. The extension of those techniques to graph machines should be investigated.

References

1. Pollack, J.B.: Recursive Distributed Representations. Artificial Intell. **46** (1990) 77-106.
2. Sperduti, A.: Encoding Labeled Graphs by Labeling RAAM. Connection Science **6** (1994) 429-459.
3. Frasconi, P., Gori, M, Sperduti, A: A General Framework for Adaptive Processing of Data Structures. IEEE Trans. on Neural Networks **9** (1998) 768-786.
4. Goulon-Sigwalt-Abram, A., Duprat, A., Dreyfus, G.: From Hopfield Nets to Recursive Networks to Graph Machines: Numerical Machine Learning for Structured Data. Th. Comp. Sci. **344** (2005) 298-334.
5. Hammer, B.: Recurrent Networks for Structured Data - a Unifying Approach and its Properties. Cognitive Systems Res. **3** (2002) 145-165.
6. Jochum, C., Gasteiger, J.: Canonical Numbering and Constitutional Symmetry. J. Chem. Inf. Comput. Sci. **17** (1977) 113-117.
7. Wiener, H.: Structural Determination of Paraffin Boiling Point. J. Amer. Chem. Soc. **69** (1947) 17-20.
8. Vapnik, V.N.: The Nature of Statistical Learning Theory. Springer-Verlag (2000).
9. Monari, G., Dreyfus, G.: Local Overfitting Control via Leverages. Neural Computation **14** (2002) 1481-1506.

10. Oussar, Y., Monari, G., Dreyfus G.: Reply to the Comments on "Local Overfitting Control via Leverages" in "Jacobian Conditioning Analysis for Model Validation" by I. Rivals and L. Personnaz. Neural Computation **16** (2004) 419-443.
11. Balaban, A.T., Basak S.C., Colburn, T., Grunwald, G.D.: Correlation between Structure and Normal Boiling Points of Haloalkanes C1-C4 Using Neural Networks. J. Chem. Inf. Comput. Sci. **34** (1994) 1118-1121.
12. Rücker, C., Meringer, M., Kerber, A.: QSPR Using MOLGEN-QSPR: The Example of Haloalkane Boiling Points. J. Chem. Inf. Comput. Sci. **44** (2004) 2070-2076.
13. Douali, L., Villemin, D., Cherqaoui, D.: Exploring QSAR of Non-Nucleoside Reverse Transcriptase Inhibitors by Neural Networks: TIBO Derivatives. Int. J. Mol. Sci. **5** (2004) 48-55.
14. Huuskonen, J.: QSAR modeling with the electrotopological state: TIBO derivatives. J. Chem. Inf. Comput. Sci. **41** (2001) 425-429.
15. Zhou, Z., Madura, J.D.: CoMFA 3D-QSAR Analysis of HIV-1 RT Nonnucleoside Inhibitors, TIBO Derivatives Based on Docking Conformation and Alignment. J. Chem. Inf. Comput. Sci. **44** (2004) 2167-2178.
16. Solov'ev, V. P., Varnek, A.: Anti-HIV Activity of HEPT, TIBO, and Cyclic Urea Derivatives: Structure-Property Studies, Focused Combinatorial Library Generation, and Hits Selection Using Substructural Molecular Fragments Method. J. Chem. Inf. Comput. Sci. **43** (2003) 1703-1719.
17. Goulon, A., Picot, T., Duprat, A., Dreyfus, G.: Predicting Activities without Computing Descriptors: Graph Machines for QSAR. Submitted to SAR and QSAR in Environmental Research.
18. Duprat, A.F., Huynh, T., Dreyfus, G.: Toward a Principled Methodology for Neural Network Design and Performance Evaluation in QSAR. Application to the Prediction of LogP. J. Chem. Inf. Comput. Sci. **38** (1998) 586-594.

Rational Models of Cognitive Control

Michael C. Mozer

Department of Computer Science and
Institute of Cognitive Science
University of Colorado
Boulder, CO 80309 USA
mozer@colorado.edu
http://www.cs.colorado.edu/~mozer

1 Cognitive Control

Human behavior is remarkably flexible. An individual who drives the same route to work each day easily adjusts for a traffic jam or to pick up lunch. Any theory of human cognition must explain not only routine behavior, but how behavior is flexibly modulated by the current environment and goals. In this extended abstract, we discuss this ability, often referred to as *cognitive control*.

In a psychological laboratory, a task that has been used to study cognitive control is the *Stroop phenomenon*. Individuals are asked to name the color in which a list of words is printed. This task is straightforward, unless the to-be-ignored words are color names, such as the word *yellow* printed in red ink. The correct response is "red," but individuals are inclined to respond "yellow." The explanation for this phenomenon is straightforward: individuals have much practice reading words but have little practice in naming colors. As a result, word reading is more automatic than color naming. What is particularly interesting about this phenomenon is that individuals can override the predominant response—reading the word—and produce the task-relevant response—naming the color.

Cognitive control is required whenever an individual performs novel activities, either because the task is novel or because the stimuli, responses, or task environment is unfamiliar. Aspects of cognitive control include: the deployment of visual attention, the selection of responses, the construction of arbitrary associations between stimuli and responses, the determination of which brain systems should process a stimulus, and the use of working memory to subserve ongoing processing. The functional organization of the brain, sometimes referred to as the *cognitive architecture*, is extremely flexible. The role of cognitive control is to reconfigure this general-purpose architecture to perform a specific task. But cognitive control involves a secondary, more subtle, ability—that of fine tuning the operation of the cognitive architecture to the environment. For example, consider searching for a key in a bowl of coins versus searching for a key on a black leather couch. In the former case, the environment dictates that the most relevant feature is the size or shape of the key, whereas in the latter case, the most relevant feature is the metallic luster of the key.

C.S. Calude et al. (Eds.): UC 2006, LNCS 4135, pp. 20–25, 2006.

2 Sequential Dependencies

The fine tuning of cognitive control to the structure of the environment is evidenced by *sequential dependencies* in human behavior. A sequential dependency is an influence of one incidental experience on subsequent experience. Sequential dependencies arise constantly in naturalistic settings. Individuals tend to perform the same activities repeatedly throughout their daily lives: driving to the office, preparing dinner, arguing with a spouse, searching for information on the web, etc. Sequential dependencies are also studied in a constrained environment via psychological experiments that require individuals to perform a task repeatedly or perform a series of tasks, and performing one task trial influences behavior on subsequent trials. Measures of behavior are diverse, including response latency, accuracy, type of errors produced, and interpretation of ambiguous stimuli.

What is the relationship between cognitive control and sequential dependencies? Sequential dependencies are a reflection of the effects of control processes. That is, cognitive control modulates information processing and internal representations, these modulations yield sequential dependencies. As a result, understanding sequential dependencies—fine tuning of behavior that follows each experience—should offer an insight into the operation of cognitive control—the tuning required to achieve flexible, adaptive behavior.

To illustrate a sequential dependency, consider the three columns of addition problems in Table 1. The first column is a list of easy problems; individuals are quick and accurate in naming the sum. The second column is a list of hard problems; individuals are slower and less accurate in responding. The third column contains a mixture of easy and hard problems. If sequential dependencies arise in repeatedly naming the sums, then the response time or accuracy to an easy problem will depend on the preceding context, i.e., whether it appears in an easy or mixed list; similarly, performance on a hard problem will depend on whether it appears in a hard or mixed list. Exactly this sort of dependency has been observed [4]: responses to a hard problem are faster but less accurate in a mixed list than in a pure list; similarly, responses to an easy problem are slower and more accurate in a mixed list than in a pure list of easy trials. Essentially, the presence of recent easy problems causes response-initiation processes to treat a hard problem as if it were easier, speeding up responses but causing them to be

Table 1. Three Series of Addition Problems

Easy List	Hard List	Mixed List
3 + 2	9 + 4	3 + 2
1 + 4	7 + 6	7 + 6
10 + 7	8 + 6	10 + 7
5 + 5	6 + 13	6 + 13

more error prone; the reverse effect occurs for easy problems in the presence of recent hard problems.

Sequential dependencies reflect cortical adaptation operating on the time scale of seconds, not—as one usually imagines when discussing learning—days or weeks. Sequential dependencies are robust and nearly ubiquitous across a wide range of experimental tasks, spanning all components of the cognitive architecture, including perception, attention, language, stimulus-response mapping, and response initiation [8]. Sequential dependencies arise in a variety of experimental paradigms. The aspect of the stimulus that produces the dependency ranges from the concrete, such as color or identity, to the abstract, such as cue validity, item difficulty, or syntax of language. Most sequential dependencies are fairly short lived, lasting roughly five intervening trials, but some varieties span hundreds of trials and weeks of passing time.

3 Rational Models of Cognition

We have proposed and evaluated a variety of cognitive models to explain sequential dependencies. A cognitive model captures essential aspects of cognitive function, matches human strengths and weaknesses, and can replicate patterns of data observed in human experimental studies. In contrast to Artificial Intelligence systems, cognitive models attempt to perform perceptual and cognitive tasks *in the same way that people do.*

We focus on a subclass of cognitive models that adopt a *rational* perspective [1], which views cognition as being optimized with respect to current goals and the statistical structure of the environment. Rational analysis has been used to understand many cognitive domains, including long-term memory [1], concept learning [9], language learning [2] [10], and low-level perceptual integration [11]. A rational account of some cognitive process does not imply the rationality of human reasoning and decision making, which is built upon many elemental cognitive processes, nor does a rational account imply that the cognitive process is ideal in an absolute sense. The notion of rationality is considered in light of limitations on processing hardware or knowledge state. In a successful rational account, a small set of assumptions concerning hardware and knowledge limitations, along with the assumption of rationality (i.e., that inference and performance is optimal subject to these limitations), leads to parsimonious, elegant accounts of data and strong predictions.

From a behavioral perspective, the natural goal of optimization is to make performance more fluid and efficient—concretely, to minimize reaction time or to maximize accuracy, or some trade-off between the two. A rational model must be sensitive to the statistical structure of the environment in which it is operating, because the structure of the environment can be exploited to make behavior more efficient. Optimality of behavior is possible only when the probabilities of various environmental states and outcomes can be estimated. As a result, rational models tend to adopt a probabilistic framework.

4 Domains Studied

We have constructed rational models to understand cognition and predict behavior across a range of tasks. We briefly summarize specific models that we have investigated, to perform the following tasks: *visual search* (locating a visual target in a cluttered display), *categorization* (labeling an item as belonging to one of a discrete set of classes), and *speeded discrimination* (making a rapid decision to a visual stimulus). In all three models, we account for behavior via the assumption that a predictive model of the environment is learned and updated over the course of experience, and control processes use this model to optimze future performance.

4.1 Sequential Effects Involving Visual Search

In a visual search task, individuals are asked to locate a target item in a cluttered display of distractor items, such as finding a red circle among green circles and red squares. Visual search shows strong sequential effects. The robust finding is that repetition of features of recent trials (e.g., target color) facilitates performance.

We view this facilitation as an adaptation to the statistical structure of the environment [6]. We suggest that control processes construct a probabilistic model of the environment that is updated after each trial to reflect the current trial. Attentional control then operates so as to optimize performance for the more likely states of the environment. For example, if a target appeared in the center of a display for several trials in a row, then an environmental models would predict that with high probability, the target will appear in the center again, and will tune visual search to be particularly efficient for a target in the center. We cast the environment model as a Bayes net, and make strong claims about how task instructions determine the structure (conditional dependencies) of the Bayes net. We obtain parsimonious explanations for data from four different experiments. Further, our model provides a rational explanation for why the influence of past experience decays so rapidly—in under a half dozen trials.

4.2 Sequential Effects Involving Categorization

Categorization is a central activity of human cognition. Individuals continually make decisions about characteristics of objects and other individuals: Is the fruit ripe? Does your friend seem unhappy? Is your car tire flat?

When an individual is asked to categorize a series of items, sequential effects arise: categorization of one item influences category decisions for subsequent items. Specifically, when experimental subjects are shown an exemplar of some target category, the category prototype appears to be pulled toward the exemplar, and the prototypes of all nontarget categories appear to be pushed away. These *push* and *pull* effects diminish with experience, and likely reflect long-term learning of category boundaries.

We propose a model to explain categorization phenomena that assumes the objective of category learning is to maximize the posterior probability of the

category given the exemplar [7]. Each category is encoded as a Gaussian density in feature space, and categorization involves computing category posteriors given an exemplar. Also essential to a complete account of the experimental data is an assumption of prior knowledge of category structure. Specifically, if the categories lie on a continuum (e.g., "small", "medium", and "large"), the structure built into the model includes ordinal information about the categories.

4.3 Sequential Effects Involving Speeded Discrimination

Consider a simple speeded discrimination task in which individuals are asked to classify a sequence of stimuli [3]. The stimuli are letters of the alphabet, A-Z, presented in rapid succession, and individuals are asked to press one response key if the letter is an X or another response key for any letter other than X (as a shorthand, we will refer to the alternative responses as X and Y). Even in a simple cognitive task like this, sequential effects arise from the relative frequency of X and Y. Response conflict arises when the two stimulus classes are unbalanced in frequency, resulting in more errors and slower reaction times. For example, when X's are frequent but Y is presented, individuals are predisposed toward producing X, and this predisposition must be overcome by the perceptual evidence from Y. Cognitive control is presumed to be required in situations involving response conflict.

How do control processes modulate behavior based on the relative class frequencies? We explain performance from a rational perspective that casts the goal of individuals as minimizing a cost that depends both on error rate and reaction time [5]. With two additional assumptions of rationality—that class prior probabilities are accurately estimated and that inference is optimal subject to limitations on rate of information transmission—we obtain a good fit to overall RT and error data, as well as trial-by-trial variations in performance.

5 Conclusions

Theories in cognitive science often hand the problem of cognitive control to an unspecified homunculus. Other theories consider cognitive control in terms of a central, unitary component of the cognitive architecture whose role is to direct processing in lower components of the architecture. In contrast, we view cognitive control as a collection of simple, specialized mechanisms, and the appearance of control emerges from this collection. We summarized three such mechanisms in this abstract: one that determines how to allocate attention and visual processing resources, one that determines where to draw boundaries in dividing our continuous world into discrete categories, and one that determines the predisposition to produce specific responses.

The central claim of all of our accounts is that a predictive model of the environment is constructed, and this model is used to optimize performance on subsequent trials. We view sequential dependencies as reflecting continual adaptation to the ongoing stream of experience, wherein each sensory and motor experience can affect subsequent behavior. Sequential dependencies suggest

that learning should be understood not only in terms of changes that occur on the time scale of hours or days, but also in terms of changes that occur from individual incidental experiences that occur on the scale of seconds.

References

1. Anderson, J. R. (1990). The Adaptive Character of Thought. Hillsdale, NJ: Erlbaum.
2. Brent, M. R., (1999). Speech segmentation and word discovery: A computational perspective. *Trends in Cognitive Science*, *3*, 294–301.
3. Jones, A. D., Cho, R. Y., Nystrom, L. E., Cohen, J. D., & Braver, T. S. (2002).A computational model of anterior cingulate function in speeded response tasks: Effects of frequency, sequence, and conflict. *Cognitive, Affective, & Behavioral Neuroscience*, *2*, 300–317.
4. Lupker, S. J., Kinoshita, S., Coltheart, M., & Taylor, T. (2003). Mixing costs and mixing benefits in naming words, pictures, and sums. *Journal of Memory and Language*, *49*, 556–575.
5. Mozer, M. C., Colagrosso, M. D., & Huber, D. H. (2002). A rational analysis of cognitive control in a speeded discrimination task. In T. Dietterich, S. Becker, & Ghahramani, Z. (Eds.) *Advances in Neural Information Processing Systems XIV* (pp. 51-57). Cambridge, MA: MIT Press.
6. Mozer, M. C., Shettel, M., & Vecera, S. P. (2006). Control of visual attention: A rational account. In Y. Weiss, B. Schoelkopf, & J. Platt (Eds.), *Neural Information Processing Systems 18* (pp. 923-930). Cambridge, MA: MIT Press.
7. Mozer, M. C., Jones, M., & Shettel, M. (2006). Context effects in categorization: An investigation of four probabilistic models. Submitted for publication.
8. Mozer, M. C., Kinoshita, S., & Shettel, M. (2006). Sequential dependencies offer insight into cognitive control. In W. Gray (Ed.), *Integrated Models of Cognitive Systems*. Oxford University Press.
9. Tenenbaum, J. (1999). Bayesian modeling of human concept learning. In M. S. Kearns, S. A. Solla, & D. A. Cohn (Eds.), *Advances in Neural Information Processing Systems 11*. Cambridge, MA: MIT Press.
10. Tenenbaum, J. B. & Xu, F. (2000). Word learning as Bayesian inference. In L. Gleitman and A. Joshi (eds.), *Proceedings of the 22nd Annual Conference of the Cognitive Science Society* (pp. 517-522). Hillsdale, NJ: Erlbaum.
11. Weiss, Y., & Adelson, E. H. (1998). *Slow and smooth: A Bayesian theory for the combination of local motion signals in human vision*. MIT AI Memo 1624 (CBCL Paper 158). Cambridge, MA: Department of Brain and Cognitive Sciences, Massachusetts Institute of Technology.

Fault-Tolerance in Biochemical Systems
(Abstract)

Erik Winfree

Computer Science and Computation & Neural Systems
California Institute of Technology
Pasadena, CA 91125, USA

Biochemistry is messy. It's a miracle any of it works. And yet it does. The wonderful diversity and amazing talents of living things derive from the biochemical processes that copy genetic information and use that information as a program to construct a sophisticated organization of matter and behaviour – reliably and robustly overcoming insult after insult from the environment. In this talk I will first discuss how known techniques for fault-tolerant computing, such as von Neumann's multiplexing technique for digital circuits, can be translated to the biochemical context. I will then discuss fault-tolerance in molecular self-assembly, which requires new techniques. Using a model of algorithmic self-assembly, a generalization of crystal growth processes, I will present techniques for controlling the nucleation of self-assembly, for reducing errors during growth, and for recovering after gross damage or fragmentation.

C.S. Calude et al. (Eds.): UC 2006, LNCS 4135, p. 26, 2006.

Optical Computing and Computational Complexity

Damien Woods

Boole Centre for Research in Informatics
School of Mathematics
University College Cork
Ireland
http://www.bcri.ucc.ie/~dw5
d.woods@bcri.ucc.ie

Abstract. This work concerns the computational complexity of a model of computation that is inspired by optical computers. The model is called the continuous space machine and operates in discrete timesteps over a number of two-dimensional images of fixed size and arbitrary spatial resolution. The (constant time) operations on images include Fourier transformation, multiplication, addition, thresholding, copying and scaling. We survey some of the work to date on the continuous space machine. This includes a characterisation of the power of an important discrete restriction of the model. Parallel time corresponds, within a polynomial, to sequential space on Turing machines, thus satisfying the parallel computation thesis. A characterisation of the complexity class NC in terms of the model is also given. Thus the model has computational power that is (polynomially) equivalent to that of many well-known parallel models. Such characterisations give a method to translate parallel algorithms to optical algorithms and facilitate the application of the complexity theory toolbox to optical computers. In the present work we improve on these results. Specifically we tighten a lower bound and present some new resource trade-offs.

1 Introduction

Over the years, optical computers were designed and built to emulate conventional microprocessors (digital optical computing), and for image processing over continuous wavefronts (analog optical computing). Here we are interested in the latter class: optical computers that store data as images. Numerous physical implementations exist and example applications include fast pattern recognition and matrix-vector algebra [9,24]. There have been much resources devoted to designs, implementations and algorithms for such optical information processing architectures (for example see [1,4,6,9,12,13,14,15,22,24,31] and their references). However the computational complexity theory of optical computers[1] has received

[1] That is, finding lower and upper bounds on computational power in terms of known complexity classes.

C.S. Calude et al. (Eds.): UC 2006, LNCS 4135, pp. 27–40, 2006.

relatively little attention when compared with other nature-insired computing paradigms. Some authors have even complained about the lack of suitable models [6,13].

We investigate the computational complexity of a model of computation that is inspired by such optical computers. The model is relatively new and is called the continuous space machine (CSM) [16,17,18,26,27,28,29,30]. The model was originally proposed by Naughton [16,17]. The CSM computes in discrete timesteps over a number of two-dimensional images of fixed size and arbitrary spatial resolution. The data and program are stored as images. The (constant time) operations on images include Fourier transformation, multiplication, addition, thresholding, copying and scaling. We analyse the model in terms of seven complexity measures inspired by real-world resources.

Subsequent to the original [17] CSM definition, Naughton [16] showed that the CSM (sequentially) simulates Turing machines, with a constant factor slowdown in time, thus giving a lower bound on its computational power. Later it was shown [18] that the model could simulate Type-2 Turing machines [25]. It was also shown that the CSM definition was perhaps too general as there is an ω-language that is Type-2 (and Turing machine) undecidable, but is CSM decidable [18], and furthermore any language is decided in finite time (and infinite space) [30]. In this paper we mostly focus on computational complexity results for a restricted CSM called the $\mathcal{C}_2$-CSM. Section 2 surveys some of the work to date on the model. This includes an analysis of complexity resources relevant to the CSM. Optical information processing is a highly parallel form of computing and we have made this intuition more concrete by relating the $\mathcal{C}_2$-CSM to parallel complexity theory. We discuss characterisations of $\mathcal{C}_2$-CSM computational power in terms of sequential space complexity classes and NC. Section 3 presents a new result that improves the lower bound for $\mathcal{C}_2$-CSM simulation of sequential space.

2 CSM and $\mathcal{C}_2$-CSM

We begin by describing the model in its most general sense, this brief overview is not intended to be complete and more details are to be found in [26].

2.1 CSM

A complex-valued image (or simply, image) is a function $f : [0,1) \times [0,1) \to \mathbb{C}$, where $[0,1)$ is the half-open real unit interval. We let $\mathcal{I}$ denote the set of complex-valued images. Let $\mathbb{N}^+ = \{1,2,3,\ldots\}$, $\mathbb{N} = \mathbb{N}^+ \cup \{0\}$, and for a given CSM M let $\mathcal{N}$ be a countable set of images that encode M's addresses. Additionally, for a given M there is an *address encoding function* $\mathfrak{E} : \mathbb{N} \to \mathcal{N}$ such that $\mathfrak{E}$ is Turing machine decidable, under some *reasonable* representation of images as words. An address is an element of $\mathbb{N} \times \mathbb{N}$.

Definition 1 (CSM). *A CSM is a quintuple* $M = (\mathfrak{E}, L, I, P, O)$*, where*

$\mathfrak{E} : \mathbb{N} \to \mathcal{N}$ *is the address encoding function,*
$L = ((s_\xi, s_\eta), (a_\xi, a_\eta), (b_\xi, b_\eta))$ *are the addresses: sta, a and b, where* $a \neq b$*,*
I *and* O *are finite sets of input and output addresses, respectively,*
$P = \{(\zeta_1, p_{1_\xi}, p_{1_\eta}), \ldots, (\zeta_r, p_{r_\xi}, p_{r_\eta})\}$ *are the* r *programming symbols* ζ_j *and their addresses where* $\zeta_j \in (\{h, v, *, \cdot, +, \rho, st, ld, br, hlt\} \cup \mathcal{N}) \subset \mathcal{I}$*.*

Each address is an element from $\{0, \ldots, \Xi - 1\} \times \{0, \ldots, \mathcal{Y} - 1\}$ *where* $\Xi, \mathcal{Y} \in \mathbb{N}^+$*.*

Addresses whose contents are not specified by P in a CSM definition are assumed to contain the constant image $f(x, y) = 0$. We interpret this definition to mean that M is (initially) defined on a grid of images bounded by the constants Ξ and $\mathcal{Y}$, in the horizontal and vertical directions respectively. The grid of images may grow in size as the computation progresses.

In our grid notation the first and second elements of an address tuple refer to the horizontal and vertical axes of the grid respectively, and image $(0, 0)$ is located at the lower left-hand corner of the grid. The images have the same orientation as the grid. For example the value $f(0, 0)$ is located at the lower left-hand corner of the image f.

In Definition 1 the tuple P specifies the CSM program using programming symbol images ζ_j that are from the (low-level) CSM programing language [26,30]. We refrain from giving a description of this programming language and instead describe a less cumbersome high-level language [26]. Figure 1 gives the basic instructions of this high-level language. The copy instruction is illustrated in Figure 3. There are also **if/else** and **while** control flow instructions with conditions of the form $(f_\psi == f_\phi)$ where f_ψ and f_ϕ are *binary symbol images* (see Figures 2(a) and 2(b)).

Address *sta* is the start location for the program so the programmer should write the first program instruction at *sta*. Addresses a and b define special images that are frequently used by some program instructions. The function $\mathfrak{E}$ is specified by the programmer and is used to map addresses to image pairs. This enables the programmer to choose her own address encoding scheme. We typically don't want $\mathfrak{E}$ to hide complicated behaviour thus the computational power of this function should be somewhat restricted. For example we put such a restriction on $\mathfrak{E}$ in Definition 7. Configurations are defined in a straightforward way as a tuple $\langle c, e \rangle$ where c is an address called the control and e represents the grid contents.

2.2 Complexity Measures

Next we define some CSM complexity measures. All resource bounding functions map from $\mathbb{N}$ into $\mathbb{N}$ and are assumed to have the usual properties [2]. Logarithms are to the base 2.

Definition 2. *The* TIME *complexity of a CSM* M *is the number of configurations in the computation sequence of* M*, beginning with the initial configuration and ending with the first final configuration.*

$\mathrm{h}(i_1;i_2)$: replace image at i_2 with horizontal 1D Fourier transform of i_1.
$\mathrm{v}(i_1;i_2)$: replace image at i_2 with vertical 1D Fourier transform of image at i_1.
$*(i_1;i_2)$: replace image at i_2 with the complex conjugate of image at i_1.
$\cdot\,(i_1,i_2;i_3)$: pointwise multiply the two images at i_1 and i_2. Store result at i_3.
$+(i_1,i_2;i_3)$: pointwise addition of the two images at i_1 and i_2. Store result at i_3.
$\rho(i_1,z_\mathrm{l},z_\mathrm{u};i_2)$: filter the image at i_1 by amplitude using z_l and z_u as lower and upper amplitude threshold images, respectively. Place result at i_2.
$[\xi_1',\xi_2',\eta_1',\eta_2'] \leftarrow [\xi_1,\xi_2,\eta_1,\eta_2]$: copy the rectangle of images whose bottom left-hand address is (ξ_1,η_1) and whose top right-hand address is (ξ_2,η_2) to the rectangle of images whose bottom left-hand address is (ξ_1',η_1') and whose top right-hand address is (ξ_2',η_2'). See illustration in Figure 3.

Fig. 1. CSM high-level programming language instructions. In these instructions $i, z_\mathrm{l}, z_\mathrm{u} \in \mathbb{N} \times \mathbb{N}$ are image addresses and $\xi, \eta \in \mathbb{N}$. The control flow instructions are described in the main text.

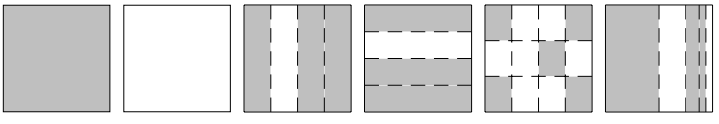


Fig. 2. Representing binary data. The shaded areas denote value 1 and the white areas denote value 0. (a) Binary symbol image representation of 1 and (b) of 0, (c) list (or row) image representation of the word 1011, (d) column image representation of 1011, (e) 3×4 matrix image, (e) binary stack image representation of 1101. Dashed lines are for illustration purposes only.

Definition 3. *The* GRID *complexity of a CSM M is the minimum number of images, arranged in a rectangular grid, for M to compute correctly on all inputs.*

Let $S : \mathcal{I} \times (\mathbb{N} \times \mathbb{N}) \to \mathcal{I}$, where $S(f(x,y),(\Phi,\Psi))$ is a raster image, with $\Phi\Psi$ constant-valued pixels arranged in Φ columns and Ψ rows, that approximates $f(x,y)$. If we choose a reasonable and realistic S then the details of S are not important.

Definition 4. *The* SPATIALRES *complexity of a CSM M is the minimum $\Phi\Psi$ such that if each image $f(x,y)$ in the computation of M is replaced with $S(f(x,y),(\Phi,\Psi))$ then M computes correctly on all inputs.*

Definition 5. *The* DYRANGE *complexity of a CSM M is the ceiling of the maximum of all the amplitude values stored in all of M's images during M's computation.*

We also use complexity measures called AMPLRES, PHASERES and FREQ [26,30]. Roughly speaking, the AMPLRES of a CSM M is the number of discrete, evenly spaced, amplitude values per unit amplitude of the complex numbers in the range of M's images. The PHASERES of M is the total number (per 2π) of discrete

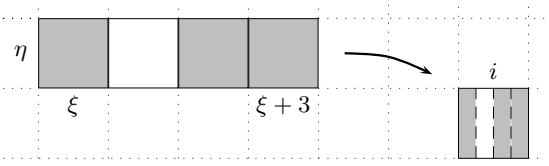


Fig. 3. Illustration of the instruction $i \leftarrow [\xi, \xi + 3, \eta, \eta]$ that copies four images to a single image that is denoted i

evenly spaced phase values in the range of M's images. FREQ is a measure of the optical frequency of M's images [30].

Often we wish to make analogies between space on some well-known model and CSM 'space-like' resources. Thus we define the following convenient term.

Definition 6. *The* SPACE *complexity of a CSM M is the product of all of M's complexity measures except* TIME.

2.3 Representing Data as Images

There are many ways to represent data as images. Here we mention some data representations that are commonly used and moreover are used in Section 3. Figures 2(a) and 2(b) are the binary symbol image representations of 1 and 0 respectively. These images have an everywhere constant value of 1 and 0 respectively, and both have SPATIALRES of 1. The row and column image representations of the word 1011 are respectively given in Figures 2(c) and 2(d). These row and column images both have SPATIALRES of 4. In the matrix image representation in Figure 2(e), the first matrix element is represented at the top left corner and elements are ordered in the usual matrix way. This 3×4 matrix image has SPATIALRES of 12. Finally the binary stack image representation, which has exponential SPATIALRES of 16, is given in Figure 2(f). Section 3.1 discusses the manipulation of stack images.

Figure 3 shows how we might form a list image by copying four images to one in a single timestep. All of the above mentioned images have DYRANGE, AMPLRES and PHASERES of 1.

2.4 Worst Case CSM Resource Usage

For the case of sequential computation it is usually obvious how the execution of a single operation will effect resource usage. In parallel models, execution of a single operation can lead to large growth in a single timestep. Characterising resource growth is useful for proving upper bounds on power and choosing reasonable model restrictions.

We investigated the growth of complexity resources over TIME, with respect to CSM operations [26,28]. As expected, under certain operations some measures do not grow at all. Others grow at rates comparable to massively parallel models.

Table 1. CSM resource usage after one timestep. For a given operation and complexity measure pair, the relevant table entry defines the worst case CSM resource usage at TIME $T+1$, in terms of the resources used at TIME T. At TIME T we have GRID $= G_T$, SPATIALRES $= R_{S,T}$, AMPLRES $= R_{A,T}$, DYRANGE $= R_{D,T}$, PHASERES $= R_{P,T}$ and FREQ $= \nu_T$.

	GRID	SPATIALRES	AMPLRES	DYRANGE	PHASERES	FREQ
h	G_T	∞	∞	∞	∞	∞
v	G_T	∞	∞	∞	∞	∞
$*$	G_T	$R_{S,T}$	$R_{A,T}$	$R_{D,T}$	$R_{P,T}$	ν_T
$\cdot$	G_T	$R_{S,T}$	$(R_{A,T})^2$	$(R_{D,T})^2$	$R_{P,T}$	ν_T
$+$	G_T	$R_{S,T}$	∞	$2R_{D,T}$	∞	ν_T
ρ	unbounded	$R_{S,T}$	$R_{A,T}$	$R_{D,T}$	$R_{P,T}$	ν_T
st	unbounded	$R_{S,T}$	$R_{A,T}$	$R_{D,T}$	$R_{P,T}$	ν_T
ld	unbounded	unbounded	$R_{A,T}$	$R_{D,T}$	$R_{P,T}$	unbounded
br	G_T	$R_{S,T}$	$R_{A,T}$	$R_{D,T}$	$R_{P,T}$	ν_T
hlt	G_T	$R_{S,T}$	$R_{A,T}$	$R_{D,T}$	$R_{P,T}$	ν_T

By allowing operations like the Fourier transform we are mixing the continuous and discrete worlds, hence some measures grow to infinity in one timestep. This gave strong motivation for CSM restrictions.

Table 1 summarises these results; the table defines the value of a complexity measure after execution of an operation (at TIME $T+1$). The complexity of a configuration at TIME $T+1$ is at least the value it was at TIME T, since complexity functions are nondecreasing. Our definition of TIME assigns unit time cost to each operation, hence we do not have a TIME column. Some entries are immediate from the complexity measure definitions, for others proofs are given in the references [26,28].

2.5 $\mathcal{C}_2$-CSM

Motivated by a desire to apply standard complexity theory tools to the model, we defined [26,28] the $\mathcal{C}_2$-CSM, a restricted and more realistic class of CSM.

Definition 7 ($\mathcal{C}_2$-CSM)**.** *A* $\mathcal{C}_2$-CSM *is a CSM whose computation* TIME *is defined for* $t \in \{1, 2, \ldots, T(n)\}$ *and has the following restrictions:*

- *For all* TIME t *both* AMPLRES *and* PHASERES *have constant value of 2.*
- *For all* TIME t *each of* GRID, SPATIALRES *and* DYRANGE *is* $O(2^t)$ *and* SPACE *is redefined to be the product of all complexity measures except* TIME *and* FREQ.
- *Operations h and v compute the discrete Fourier transform (DFT) in the horizontal and vertical directions respectively.*
- *Given some* reasonable *binary word representation of the set of addresses $\mathcal{N}$, the address encoding function $\mathfrak{E} : \mathbb{N} \to \mathcal{N}$ is decidable by a logspace Turing machine.*

Let us discuss these restrictions. The restrictions on amplRes and phaseRes imply that $\mathcal{C}_2$-CSM images are of the form $f : [0,1)\times[0,1) \rightarrow \{0, \pm\frac{1}{2}, \pm 1, \pm\frac{3}{2}, \ldots\}$. We have replaced the Fourier transform with the DFT [3], this essentially means that freq is now solely dependent on spatialRes; hence freq is not an interesting complexity measure for $\mathcal{C}_2$-CSMs and we do not analyse $\mathcal{C}_2$-CSMs in terms of freq complexity [26,28]. Restricting the growth of space is not unique to our model, such restrictions are to be found elsewhere [8,19,21].

In Section 2.1 we stated that the address encoding function $\mathfrak{E}$ should be Turing machine decidable, here we strengthen this condition. At first glance sequential logspace computability may perhaps seem like a strong restriction, but in fact it is quite weak. From an optical implementation point of view it should be the case that $\mathfrak{E}$ is not complicated, otherwise we cannot assume fast addressing. Other (sequential/parallel) models usually have a very restricted 'addressing function': in most cases it is simply the identity function on $\mathbb{N}$. Without an explicit or implicit restriction on the computational complexity of $\mathfrak{E}$, finding non-trivial upper bounds on the power of $\mathcal{C}_2$-CSMs is impossible as $\mathfrak{E}$ could encode an arbitrarily complex halting Turing machine. As a weaker restriction we could give a specific $\mathfrak{E}$. However, this restricts the generality of the model and prohibits the programmer from developing novel, reasonable, addressing schemes.

2.6 $\mathcal{C}_2$-CSM and Parallel Complexity Theory

We have given lower bounds on the computational power of the $\mathcal{C}_2$-CSM by showing that it is at least as powerful as models that verify the parallel computation thesis [26,29]. This thesis [5,7] states that parallel time corresponds, within a polynomial, to sequential space for reasonable parallel models. See, for example, [10,11,19,23] for details. Let $S(n)$ be a space bound that is $\Omega(\log n)$. The languages accepted by nondeterministic Turing machines in $S(n)$ space are accepted by $\mathcal{C}_2$-CSMs computing in time $O(S^4(n))$.

Theorem 1 ([26,29]). $\text{NSPACE}(S(n)) \subseteq \mathcal{C}_2\text{-CSM–TIME}(O(S^4(n)))$

For example polynomial time $\mathcal{C}_2$-CSMs accept the PSPACE languages. (Of course any polynomial time $\mathcal{C}_2$-CSM algorithm that we could presently write to solve PSPACE-complete or NP-complete problems would require exponential space.) Theorem 1 is established via $\mathcal{C}_2$-CSM simulation of vector machines [2,20,21]. In the simulation the space overhead is polynomial in vector machine space. Using this fact we find that $\mathcal{C}_2$-CSMs that simultaneously use polynomial space and polylogarithmic time accept the class NC.

Corollary 1 ([26,29]). $\text{NC} \subseteq \mathcal{C}_2\text{-CSM–SPACE, TIME}(n^{O(1)}, \log^{O(1)} n)$

We have also given the other of the two inclusions that are necessary in order to verify the parallel computation thesis; $\mathcal{C}_2$-CSMs computing in time $T(n)$ are no more powerful than $O(T^2(n))$ space bounded deterministic Turing machines.

Theorem 2 ([26,27]). $\mathcal{C}_2$-CSM-TIME$(T(n)) \subseteq$ DSPACE$(O(T^2(n)))$

Via the proof of Theorem 2 we get another (stronger) result. $\mathcal{C}_2$-CSMs that simultaneously use polynomial SPACE and polylogarithmic TIME accept at most NC.

Corollary 2 ([26,27]). $\mathcal{C}_2$-CSM-SPACE, TIME$(n^{O(1)}, \log^{O(1)} n) \subseteq$ NC

The latter two inclusions are established via $\mathcal{C}_2$-CSM simulation by logspace uniform circuits of size and depth polynomial in SPACE and TIME respectively. Thus $\mathcal{C}_2$-CSMs that simultaneously use both polynomial SPACE and polylogarithmic TIME characterise NC.

It turns out that the $\mathcal{C}_2$-CSM simulation of sequential space can be made more efficient. Theorem 3 in the next section improves the lower bound given by Theorem 1.

3 Improved $\mathcal{C}_2$-CSM Lower Bound

In this section we improve the lower bound given by Theorem 1 by proving the following result.

Theorem 3. NSPACE$(S(n)) \subseteq \mathcal{C}_2$-CSM–TIME$(O(S^2(n)))$

Moreover the GRID and DYRANGE complexities are both reduced from $O(2^{S(n)})$ to $O(1)$. However we see a trade-off here as the reduction in GRID and DYRANGE is swapped for an increase[2] in SPATIALRES from $O(2^{S(n)})$ to $O(2^{3S(n)}S^3)$. Thus the SPACE overhead in Theorem 3 has not decreased, nevertheless the trade-off is interesting. Also the simulation is achieved[3] with AMPLRES of 1 and PHASERES of 1. In summary, we have tightened the relationship between the $\mathcal{C}_2$-CSM and sequential space:

Corollary 3

$$\text{NSPACE}(S(n)) \subseteq \mathcal{C}_2\text{-CSM–TIME}(O(S(n))^2) \subseteq \text{DSPACE}(O(S(n))^4)$$

We prove Theorem 3 by giving a $\mathcal{C}_2$-CSM that efficiently generates (Lemma 1) and squares (Lemma 2) the transition matrix of a $S(n) = \Omega(\log n)$ space bounded Turing machine.

We assume that Turing machines have a single tape, use only binary tape symbols $\{0, 1\}$ and are nondeterministic. At each timesstep the tape head moves either left (denoted L) or right (denoted R). The proofs below are sketched in the sense that we refrain from giving explicit code.

[2] On a technical note we are abusing notation here. $\mathcal{C}_2$-CSMs are defined to use SPATIALRES $O(2^t)$ after t timesteps. To save the reader the burden of new notation we overload the notation "$\mathcal{C}_2$-CSM" by using it to also describe machines that are $\mathcal{C}_2$-CSMs except for the fact that they have a $O(2^{O(1)t})$ upperbound on SPATIALRES. Although we omit the details, we note that Theorem 2 and Corollary 2 still hold for such (more general) definitions of $\mathcal{C}_2$-CSM.

[3] This is in contrast to the proof of the previous lower bound proof [26,29] where AMPLRES and PHASERES were both 2. Subtraction (via addition of negative numbers) and devision by 2 (via multiplication by 1/2) are not needed in the present proof.

3.1 Iteration

In order to bound iterative loops we use a 'counter image'. In previous work [26,29] we used an image with value/range of k (and thus of DYRANGE k) as a counter for k iterations. At each iteration the counter image is decremented by 1 (by adding an image of value -1), and tested for equality with 0 (by addressing).

Here we are restricted to constant DYRANGE so a different approach is adopted. Our counter image for value k is a *unary stack image* that represents 1^k. A unary stack image is just like the binary stack illustrated in Figure 2(e) except that the represented word is a list of ones. To access the i^{th} bit in a stack image we 'pop' the stack i times. Popping involves spreading the stack over two horizontally adjacent images, the leftmost image now contains the topmost stack element, the rightmost image contains the remainder of the stack. Popping the stack in this way uses GRID $O(1)$ and TIME $O(k)$ to pop the entire stack. After each pop we test if the popped element is 0 by addressing, this happens only on pop $k+1$. The unary stack image representation requires SPATIALRES of $O(2^k)$, and AMPLRES, PHASERES and DYRANGE of 1.

In the sequel we simply write $S(n)$ as S. In the proof of Lemma 1 below all loops run for S or $\log S$ iterations. Thus their counter images have SPATIALRES of $O(2^S)$, which is no more than the SPATIALRES of other parts of the algorithm. A similar argument holds for Lemma 2.

3.2 Generating the Transition Matrix

The configuration graph of a space bounded Turing machine M is a graph with exactly one node for each configuration of M. There is a path from node i to node j iff configuration c_i leads to configuration c_j in exactly one step via some transition rule of M (formally we write $c_i \vdash_M c_j$). On input w, given the pair of nodes corresponding to the (unique) initial and accepting configurations, simulating the computation of $M(w)$ is the same as asking if there is a path from the initial node to the accepting node. We simulate M by computing the reflexive transitive closure of the transition graph. To do this we represent the graph by a binary matrix which we call the transition matrix of M. There is one row (respectively column) for each node. Entry (i, j) is 1 iff there is a path from node i to node j. The reflexive transitive closure is computed by squaring the matrix a number of times that is logarithmic in the number of nodes. Motivations and further details are to be found in van Emde Boas' survey [23].

We begin by constructing the binary transition matrix.

Lemma 1. *Let M be a Turing machine that accepts $L \in \{0,1\}^*$ in space $S = 2^i$ for some $i \in \mathbb{N}$. Then there is a $\mathcal{C}_2$-CSM that generates the transition matrix of M in* TIME $O(S)$, SPATIALRES $O(2^{2S}S^2)$, GRID $O(1)$, DYRANGE $O(1)$, AMPLRES 1 *and* PHASERES 1.

Proof (sketch). Let Q be the states of M and $t = (q_x, \sigma_1, \sigma_2, D, q_y)$ be an arbitrary transition rule of M, with initial state q_x, next state q_y, read symbol $\sigma_1 \in \{0,1\}$, write symbol $\sigma_2 \in \{0,1\}$, and tape head move direction $D \in \{L, R\}$.

Before generating the matrix we precompute some special images.

A Turing machine tape word is represented in a straightforward way as a binary list image. We quickly generate all 2^S possible words of length S in TIME $O(\log S)^2$ and SPATIALRES $O(2^S S)$. The output, denoted TapesVertical, is a list-matrix image with 2^S rows and S columns, where each row represents a unique tape word. To do this we use an algorithm that (recursively) generates the matrix image $\text{TapesVertical}_{s/2}$ of all words of length $S/2$. We let $f = \text{TapesVertical}_{s/2}$ then the following is repeated $\log S$ times: place one copy of f immediately above another, scale the two to one image, call the new image f. After this repeated scaling f contains S copies of $\text{TapesVertical}_{s/2}$. We place f immediately to the right of $\text{TapesVertical}_{s/2}$ and the two are scaled to a single image to give TapesVertical.

We generate the image TapesHorizontal that represents each possible tape word repeated S times. More precisely, TapesHorizontal is the list image representation of the binary word

$$(0^S)^S(0^{S-1}1)^S(0^{S-2}10)^S(0^{S-2}11)^S \dots (1^S)^S$$

TapesHorizontal is generated in TIME $O(S)$ and SPATIALRES $O(2^S S^2)$ from TapesVertical by copying and shifting subimages, the details are omitted.

A tape head position $k \in \{1, \dots, S\}$ is encoded as the list image representation of the word $0^{k-1}10^{S-k}$. There are S such words and we generate these in TIME $O(\log S)$ and SPATIALRES $O(S^2)$ by copying and scaling. The output P is a $S \times S$ matrix image with ones on the diagonal ($P_{i,i} = 1$) and zeros elswhere. Each row represents a unique tape head position. The image PositionsVertical consists of 2^S vertically juxtaposed copies of P and is easily generated in TIME $O(S)$.

We generate the image PositionsHorizontal that represents the list of all possible position words, repeated 2^S times. More precisely, PositionsHorizontal is the list image representation of the binary word

$$\left((10^{S-1})(010^{S-2})(0010^{S-3}) \dots (0^{S-1}1)\right)^{2^S}$$

PositionsHorizontal is generated in TIME $O(S)$ and SPATIALRES $O(2^S S^2)$ from PositionsVertical by copying and shifting subimages, the details are omitted.

Finally we precompute the image P_R which is identical to P except that the represented tapes have their head positions moved one cell to the right (if the head was on the rightmost tape cell then it is moved to the leftmost tape cell).

We are now ready to generate the transition matrix. The Turing machine has at most $8|Q|^2$ transition rules. For simplicity we assume that all $8|Q|^2$ possible transition rules are explicitly given. We begin by generating the transition matrix for one of these transition rules t that changes the machine from state q_ℓ to state q_m as follows: $t = (q_\ell, 1, 1, R, q_m)$. Thus we are generating a matrix image that represents a binary matrix with entry (i, j) equal to 1 iff $c_i \vdash c_j$ via t.

First we generate a column image, denoted $\bar{\sigma}_1$, with entry $i \in \{1, \dots, 2^S S\}$ equal to 1 iff the read symbol of c_i is $\sigma_1 = 1$. We use PositionsVertical as a mask to isolate the read symbols from TapesVertical; that is we pointwise multiply

PositionsVertical and TapesVertical in TIME $O(1)$. The resulting matrix is called MaskedReadSymbols. We vertically split MaskedReadSymbols into a left image and a right image, pointwise add the two, and repeat; after $\log S$ iterations the output is the column image $\bar{\sigma}_1$.

Secondly we generate a row image, denoted $\bar{\sigma}_2$, where entry $j \in \{1, \ldots, 2^S S\}$ is 1 iff the write symbol of c_j is $\sigma_2 = 1$. We use PositionsHorizontal as a mask to isolate the write symbols from TapesHorizontal; that is we pointwise multiply PositionsHorizontal and TapesHorizontal in TIME $O(1)$. The resulting matrix is called MaskedWriteSymbols. We 'shuffle' this row of 2^S lists to a column of 2^S lists, that is we repeat the following S times: vertically split into a left image and a right image, place the left image above the right and scale to one image. Then we vertically split the result (in half) into a left image and a right image, pointwise add the two, and repeat for a total of $\log S$ iterations. We then 'unshuffle' this column to a row in TIME $O(S)$ to get $\bar{\sigma}_2$.

Thirdly we generate a $2^S S \times 2^S S$ binary matrix image called $\overline{\text{positions}}$, where entry (i, j) is 1 iff the tape head position on configuration c_i, *after* a move to the right (recall $D = R$), is equal to the tape head position of configuration c_j. To do this we generate P'_R which is a $S \times S^2$ matrix image with S copies of P_R side by side. We then pointwise multiply P'_R by the row image that represents

$$(10^{S-1})(010^{S-2})(0010^{S-3})\ldots(0^{S-1}1)$$

The result of this multiplication is a $S \times S^2$ matrix image. Then (using the technique of shuffling and adding mentioned above) this $S \times S^2$ matrix image is 'shuffled' $\log S$ times, vertically split and added $\log S$ times, and 'unshuffled' $\log S$ times. The resulting $S \times S$ matrix image is replicated 2^{2S} times to create a 'square' $2^S S \times 2^S S$ matrix image denoted $\overline{\text{positions}}$.

We pointwise multiply $\bar{\sigma}_1$, $\bar{\sigma}_2$ and $\overline{\text{positions}}$ in TIME $O(1)$, and threshold between 0 and 1, to get a $2^S S \times 2^S S$ binary matrix image. Entry (i, j) of this matrix image is 1 iff c_i yields c_j in one step under the read symbol 1, write symbol 1 and tape head direction R.

This above procedure is repeated 8 times with different values for the triple (σ_1, σ_2, D) where $\sigma_1, \sigma_2 \in \{0, 1\}$ and $D \in \{L, R\}$. The resulting 8 matrix images are pointwise added in TIME $O(1)$ to give a matrix image denoted B. Entry (i, j) in B is 1 iff c_i yields c_j in one step under any (σ_1, σ_2, D). We then create a $|Q| \times |Q|$ matrix image where entry (i, j) is 1 iff state q_i yields q_j via some transition rule (this can be computed sequentially in a straightforward way in TIME $O(|Q|^2)$, or in parallel TIME $O(\log |Q|)$ using techniques similar to those above). We multiply this by a $2^S S|Q| \times 2^S S|Q|$ matrix image that consists of $|Q|^2$ copies of B. The result is the binary matrix image that represents the transition matrix of M. □

3.3 Squaring the Transition Matrix

Lemma 2. *Let n be a power of 2 and let A by a $n \times n$ binary matrix. The matrix A^2 is computed by a $\mathcal{C}_2$-*CSM*, using the matrix image representation, in* TIME $O(\log n)$, SPATIALRES $O(n^3)$, GRID $O(1)$, DYRANGE $O(1)$, AMPLRES 1 *and* PHASERES 1.

Proof (sketch). In this proof the matrix, and its matrix image representation are both denoted A. We being with some precomputation, then one parallel pointwise multiplication step followed by $\log n$ additions completes the algorithm.

We generate the matrix image A_1 that consists of n vertically juxtaposed copies of A. This is computed by placing one copy of A above the other, scaling to one image, and repeating to give a total of $\log n$ iterations. The image A_1 is constructed in TIME $O(\log n)$, GRID $O(1)$ and SPATIALRES $O(n^3)$.

Next we transpose A to the column image A_2. The first n elements of A_2 are row 1 of A, the second n elements of A_2 are row 2 of A, etc. This is computed in TIME $O(\log n)$, GRID $O(1)$ and SPATIALRES $O(n^2)$ as follows.

Let $A' = A$ and $i = 2n$. We horizontally split A' into a left image A'_L and a right image A'_R. Then A'_L is pointwise multiplied (or masked) by the column image that represents $(10)^i$, in TIME $O(1)$. Similarly A'_R is pointwise multiplied (or masked) by the column image that represents $(01)^i$. The masked images are added. The resulting image has half the number of columns as A' and double the number of rows, and for example: row 1 consists of the first half of the elements of row 1 of A' and row 2 consists of the latter half of the elements of row 1 of A'. We call the result A' and we double the value of i. We repeat the process to give a total of $\log n$ iterations. After these iterations the resulting column image is denoted A_2.

We pointwise multiply A_1 and A_2 to give A_3 in TIME $O(1)$, GRID $O(1)$ and SPATIALRES $O(n^3)$.

To facilitate a straightforward addition we first transpose A_3 in the following way: A_3 is vertically split into a bottom and a top image, the top image is placed to the left of the bottom and the two are scaled to a single image, this splitting and scaling is repeated to give a total of $\log n$ iterations and we call the result A_4. Then to perform the addition, we vertically split A_4 into a bottom and a top image. The top image is pointwise added to the bottom image and the result is thresholded between 0 and 1. This splitting, adding and thresholding is repeated a total of $\log n$ iterations to create A_5. We 'reverse' the transposition that created A_4: image A_5 is horizontally split into a left and a right image, the left image is placed above the right and the two are scaled to a single image, this splitting and scaling is repeated a total of $\log n$ iterations to give A^2. □

3.4 Proof of Main Result

At this point we have all the main ingredients for the proof of Theorem 3 which goes as follows. Using Lemma 1 we generate the $2^S S|Q| \times 2^S S|Q|$ binary transition matrix within the stated resource bounds. We put ones on the diagonal of this matrix by pointwise adding it to the $2^S S|Q| \times 2^S S|Q|$ identity matrix and thresholding the result between 0 and 1, all in constant TIME (however generating the identity matrix takes TIME $O(S)$). In TIME $O(S^2)$ we compute the reflexive and transitive closure of this matrix by squaring it $O(S)$ times via Lemma 2. In terms of M's input length n, the overall TIME is $O(S^2(n))$ and both SPATIALRES and SPACE are $O(2^{3S(n)}S^3(n))$.

Acknowledgements

Thanks to Cristian Calude, Grzegorz Rozenberg and the conference organisers for inviting me to UC'06. Also thanks to Tom Naughton and Paul Gibson who were collaborators on much of the previous work that is surveyed in Section 2. This work is funded by the Irish Research Council for Science, Engineering and Technology.

References

1. H. H. Arsenault and Y. Sheng. *An introduction to optics in computers*, volume TT 8 of *Tutorial texts in optical engineering*. SPIE, 1992.
2. J. L. Balcázar, J. Díaz, and J. Gabarró. *Structural complexity II*, volume 22 of *EATCS Monographs on Theoretical Computer Science*. Springer, Berlin, 1988.
3. R. N. Bracewell. *The Fourier transform and its applications*. Electrical and electronic engineering series. McGraw-Hill, second edition, 1978.
4. H. J. Caulfield. Space-time complexity in optical computing. In B. Javidi, editor, *Optical information-processing systems and architectures II*, volume 1347, pages 566–572. SPIE, July 1990.
5. A. K. Chandra and L. J. Stockmeyer. Alternation. In *17th annual symposium on Foundations of Computer Science*, pages 98–108, Houston, Texas, Oct. 1976. IEEE. Preliminary Version.
6. D. G. Feitelson. *Optical Computing: A survey for computer scientists*. MIT Press, 1988.
7. L. M. Goldschlager. *Synchronous parallel computation*. PhD thesis, University of Toronto, Computer Science Department, Dec. 1977.
8. L. M. Goldschlager. A universal interconnection pattern for parallel computers. *Journal of the ACM*, 29(4):1073–1086, Oct. 1982.
9. J. W. Goodman. *Introduction to Fourier optics*. McGraw-Hill, New York, second edition, 1996.
10. R. Greenlaw, H. J. Hoover, and W. L. Ruzzo. *Limits to parallel computation: P-completeness theory*. Oxford university Press, Oxford, 1995.
11. R. M. Karp and V. Ramachandran. *Parallel algorithms for shared memory machines*, volume A. Elsevier, Amsterdam, 1990.
12. J. N. Lee, editor. *Design issues in optical processing*. Cambridge studies in modern optics. Cambridge University Press, 1995.
13. A. Louri and A. Post. Complexity analysis of optical-computing paradigms. *Applied optics*, 31(26):5568–5583, Sept. 1992.
14. A. D. McAulay. *Optical computer architectures*. Wiley, 1991.
15. T. Naughton, Z. Javadpour, J. Keating, M. Klíma, and J. Rott. General-purpose acousto-optic connectionist processor. *Optical Engineering*, 38(7):1170–1177, July 1999.
16. T. J. Naughton. Continuous-space model of computation is Turing universal. In S. Bains and L. J. Irakliotis, editors, *Critical Technologies for the Future of Computing*, Proceedings of SPIE vol. 4109, pages 121–128, San Diego, California, Aug. 2000.
17. T. J. Naughton. A model of computation for Fourier optical processors. In R. A. Lessard and T. Galstian, editors, *Optics in Computing 2000*, Proc. SPIE vol. 4089, pages 24–34, Quebec, Canada, June 2000.

18. T. J. Naughton and D. Woods. On the computational power of a continuous-space optical model of computation. In M. Margenstern and Y. Rogozhin, editors, *Machines, Computations and Universality: Third International Conference (MCU'01)*, volume 2055 of *LNCS*, pages 288–299, Chişinău, Moldova, May 2001. Springer.
19. I. Parberry. *Parallel complexity theory*. Wiley, 1987.
20. V. R. Pratt, M. O. Rabin, and L. J. Stockmeyer. A characterisation of the power of vector machines. In *Proc. 6th annual ACM symposium on theory of computing*, pages 122–134. ACM press, 1974.
21. V. R. Pratt and L. J. Stockmeyer. A characterisation of the power of vector machines. *Journal of Computer and Systems Sciences*, 12:198–221, 1976.
22. J. H. Reif and A. Tyagi. Efficient parallel algorithms for optical computing with the discrete Fourier transform (DFT) primitive. *Applied optics*, 36(29):7327–7340, Oct. 1997.
23. P. van Emde Boas. Machine models and simulations. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, chapter 1. Elsevier, Amsterdam, 1990.
24. A. VanderLugt. *Optical Signal Processing*. Wiley Series in Pure and Applied Optics. Wiley, New York, 1992.
25. K. Weihrauch. *Computable Analysis: An Introduction*. Texts in Theoretical Computer Science. Springer, Berlin, 2000.
26. D. Woods. *Computational complexity of an optical model of computation*. PhD thesis, National University of Ireland, Maynooth, 2005.
27. D. Woods. Upper bounds on the computational power of an optical model of computation. In *16*$^{\text{th}}$ *International Symposium on Algorithms and Computation (ISAAC 2005)*, volume 3827 of *LNCS*, pages 777–788, Sanya, China, Dec. 2005. Springer.
28. D. Woods and J. P. Gibson. Complexity of continuous space machine operations. In S. B. Cooper, B. Löewe, and L. Torenvliet, editors, *New Computational Paradigms, First Conference on Computability in Europe (CiE 2005)*, volume 3526 of *LNCS*, pages 540–551, Amsterdam, June 2005. Springer.
29. D. Woods and J. P. Gibson. Lower bounds on the computational power of an optical model of computation. In C. S. Calude, M. J. Dinneen, G. Păun, M. J. Pérez-Jiménez, and G. Rozenberg, editors, *Fourth International Conference on Unconventional Computation (UC'05)*, volume 3699 of *LNCS*, pages 237–250, Sevilla, Oct. 2005. Springer.
30. D. Woods and T. J. Naughton. An optical model of computation. *Theoretical Computer Science*, 334(1–3):227–258, Apr. 2005.
31. F. T. S. Yu, S. Jutamulia, and S. Yin, editors. *Introduction to information optics*. Academic Press, 2001.

If a Tree Casts a Shadow Is It Telling the Time?

Russ Abbott

Department of Computer Science, California State University
Los Angeles, Ca, USA
Russ.Abbott@GMail.com

Abstract. Physical processes are computations only when we use them to externalize thought. Computation is the performance of one or more fixed processes within a contingent environment. We reformulate the Church-Turing thesis so that it applies to programs rather than to computability. When suitably formulated agent-based computing in an open, multi-scalar environment represents the current consensus view of how we interact with the world. But we don't know how to formulate multi-scalar environments.

1 Introduction

In the preface to the first edition of the *International Journal of Unconventional Computation*, the editorial board [1] welcomed papers in "information processing based on physics, chemistry and biology." But the Board left undefined what it means to say (a) that a physical, chemical, or biological system is doing "information processing" or (b) that information processing is "based on physics, chemistry, or biology." In this paper we explore these issues by focusing on these questions.

- *What is computation?*
- *How can computation be distinguished from other natural processes?*
- *What is the relationship between ideas and computations?*
- *What is the relationship between a computational process and the environment within which it occurs?*

Our conclusions will be that physical processes are considered computation when we treat them as externalized thought and that computation itself involves the playing out of fixed processes against a contingent environment. We re-interpret the Church-Turing Thesis: programs represent how we understand rigorous thought to be expressed. We then agree with Wegner [2] that the agent-based model of computation is the right way to think about interaction with an environment. But we claim that we are not yet in a position to specify environments that are multi-scalar.

1.1 Is Google Reading My Email?

That's the first question in the Google Gmail help center [3]. This question arises because Gmail places ads next to email messages, and the selection of ads

C.S. Calude et al. (Eds.): UC 2006, LNCS 4135, pp. 41–56, 2006.

is based on the contents of the messages. Google's answer to this question has varied over time. On March 13, 2006, the posted answer was as follows.

> Google computers scan the text of Gmail messages in order to filter spam and detect viruses, just as all major webmail services do. Google also uses this scanning technology to deliver targeted text ads and other related information. *The process is completely automated and involves no humans.* [Emphasis added.]

In other words, Google's computers are reading your email-but no human beings are. That most people find this reassuring illustrates the intuition that *it's what goes on in the mind of a human being that matters to us.*

One might object that if a computer is reading one's email (and storing its contents in a database), a person might read it later. That's quite true, and the fact that only Google computers (and not Google employees) are reading one's email when selecting ads does not guarantee one's privacy. But if no person *ever* reads one's email, then most people will not feel that their privacy has been violated.

After all, all email is read by a number of computers as it passes from sender to receiver. No one has ever worried about that. The moment of violation occurs when some living human being becomes consciously aware of one's personal information.

But, one might argue, the kind of reading that occurs when a computer transmits a message along a communication channel is qualitatively different from the kind of reading that occurs when a Google computer determines which ads to place next to a message. The former kind of reading treats messages as character strings. No meaning is extracted. The kind or reading that Google computers do extracts (or attempts to extract) meaning so that related ads can be displayed.

This raises the question of what we understand by the term *meaning*. That's clearly a larger topic than we can settle here, but our short answer is that our intuitive sense of meaning has something to do with an idea or thought forming in a mind.[1] At this stage in the development of technology, most people don't believe it makes sense to say that an idea has formed in the mind of a computer-or even to say that a computer has a mind at all. We may speak informally and say something like "the computer is doing this because it thinks that." But when we say these sorts of things, we are deliberately speaking metaphorically. Until we start to think of computers as having minds that have subjective experience, minds in which ideas can form-then most people will feel comfortable with Google's reply that its computers, but no human beings, are reading one's email.

1.2 To Come

Section 2 continues the discussion of thoughts and introduces the notion of thought tools, for which it provides a brief history. Section 3 considers how computation might be defined. Section 4 discusses the agent-based computing

[1] This clearly is different from the formal semantics sense in which meaning refers to a mapping from an expression to a model.

paradigm as more than just an approach to programming and modeling but as common to many of the ways we think about both thinking and our interaction with nature.

2 Thinking and Thought Tools

If a tree grows in a forest, but no one counts its rings is it counting years? Is it performing an unconventional computation? If a tree grows in a forest but no one knows it's there, is it instantiating the idea of a tree? These questions have the same sort of answers as does Bishop Berkeley's famous question: if a tree falls in a forest with no one around to hear it, does it make a sound?

Berkeley's question is not as difficult as it seems. Our answer, which is different from Berkeley's,[2] is that one must distinguish between physical events and subjective experience. If a tree falls in a forest, it generates (what we call) *sound waves* whether someone is there to hear them or not. But if no one is there to hear the sound, if no being has a subjective experience of the sound, then no sound will be heard.

The same holds for ideas. Like the subjective experience of a sound, the *idea* of a tree exists only as a subjective experience. If no one has that subjective experience, then a tree without anyone knowing about it will not be instantiating the idea of a tree.

Even if one grants that the idea of a tree is exactly the right way to describe that particular aspect of nature, that idea exists only as an idea, and it exists only in the mind of someone who is thinking it. Ideas exist only within the realm of mental events, i.e., as subjective experience. In saying this we are taking an explicitly anti-Platonist stance: there is no realm outside the mind in which ideas exist on their own.[3]

This is not intended as mystical or profound-just a statement of a brute fact: an idea is something that occurs only in someone's mind. The ideas in this paper exist only in the mind of the author and the minds of the readers as the author and readers are thinking them. These ideas don't exist on the paper or on the computer screens on which these words appear. They don't exist in the computer memory in which these words are stored. Just as the moment at which an invasion of privacy occurs is when some being-with-a-mind learns something personal about us, an idea exists only when someone is thinking it.[4] We go to such lengths to make this point because our position is that computations, like

[2] Berkeley's answer is that it makes a sound because God, who is always everywhere, hears it.

[3] We are not taking a stand on nominalism vs. realism. Although we believe that our (human) ideas about how nature should be described are not arbitrary and that entities other than the elementary particles exist (see Abbott [4]), that is not at issue here.

[4] This position requires some care in formulation. If an idea exists only when someone is thinking it, what does it mean to say that two people have or had "the same" idea? We believe that these issues can be worked out.

ideas, are also mental events, but mental events that we have externalized in a way that allow us to use physical processes to perform them.

When a tree grows rings, it just grows rings. But when we use that tree-ring growth as a way to count years, i.e., to help us work with ideas such as the idea of a year, then we can say that the tree has performed a computation-an unconventional one.

When a computer runs is it computing? Our answer is the same. A computer is computing only when it is understood to be performing some externalized mental activity. Otherwise, it's just an arena within which electrons are moving about.

2.1 A Brief History of the Internalization and Then the Externalization of Thought

One may trace one thread through the history of thought as the internalization and then the externalization of thought. Initially we looked outward for answers to questions about how to make sense of the world. Not knowing what else to do, we looked to sources of what we hoped were authority: priests, oracles, prophets, sacred writings, divinities, etc., to tell us what thoughts to install in our minds.[5]

We often fought with each other about whose sources of knowledge were right. In a recent op-ed piece [5] Lorenzo Albacete, a Roman Catholic priest, articulated the position of those who fear the use of religion as a source of knowledge.

> For [nonbelievers], what makes Christianity potentially dangerous [is not its other-worldliness but] its insistence that faith is the source of knowledge about this world.

As Albacete later notes, by the time of the Roman Empire, the use of religion as a source of ideas about how nature works had been discarded by enlightened thinkers. Greek and Roman philosophers believed that they themselves could be a source of knowledge about the world. The step from looking for external sources of knowledge to supposing that perhaps we can figure it out for ourselves is what we are referring to as the internalization of thought-attributing to oneself the power to produce thoughts of value and rejecting the notion that thoughts must originate externally to be valid.

2.2 Externalizing Thought and Tools to Work with It

The history of early computing may be traced along three paths. Each path traces devices that help us think about a particular (and fundamental) subject area: time, counting (arithmetic), and space (geometry).

[5] One wonders what priests, oracles, prophets, and other human authorities believed about how the ideas they transmitted arrived in their own minds. Perhaps they believed that the ideas had been implanted in their minds as a result of their special status or as a result of some special words or rituals that they performed. Perhaps they were just transmitting ideas that had been transmitted to them. Presumably they didn't believe that they themselves made up these ideas. Most likely they didn't ask themselves this question.

2.3 Time Computers

We used natural processes to help us express our ideas about time-the daily, monthly, and yearly cycles of the earth, moon, and sun. Not to beat this point into the ground, *day, month,* and *year* are ideas. As ideas, they exist only in the mind-no matter how accurate or true they are as descriptions of nature.

The first (analog[6]) time computers were the actual processes that corresponded to our thoughts. The rising and setting of the sun were the physical events that we used to keep track of the mental events: the *start* and *end* of a *day.* Similarly for the moon. Yearly events such as river floodings and the comings and goings of the seasons helped us keep track of the mental event: the yearly cycle.

It didn't take us long to invent more sophisticated analog computers. The sundial, for example, is an analog computing device. The position of the sun's shadow is an analog for the mental event *time-of-day* which corresponds to the physical relationships between the relative positions of the sun and the earth.

It is worth noting that with the sundial we started to arrange physical materials to help us track our thoughts. In building sundials we set up shadow casters, which in conjunction with the sun and markings on the surface on which the shadow is cast, helped us track (our ideas about) the passing of the day. Presumably this was not a very significant step from using existing shadow-casting objects, e.g., trees, for the same purpose. Hence our title: if a tree casts a shadow, is it telling the time?

2.4 Using Epiphenomenal Shadows to Tell the Time

The use of shadows as thought tools deserves special attention. In [4] we discuss naturally occurring entities that persist independently of human observation. These include atoms, molecules, animals, organizations, hurricanes, galaxies, and most of the things we intuitively think of as entities. Shadows are not in this category.

A shadow, after all, is that portion of a background that is not illuminated by a light source because an object is blocking the light. The shadow itself is not an entity. At best a shadow-and more importantly, the leading edge of a moving shadow-is an epiphenomena of the changing relationships among the light source, the background, and the object. Although the mechanisms are completely different, a (moving) shadow is very much like a (moving) pattern (such as a glider pattern) in the Game of Life. In both cases, the apparent object (the shadow or the Game of Life pattern) consists of illuminated/not illuminated or on/off elements on a surface. Over time, the on/off elements may appear to move across the surface. In fact, the on/off elements don't move; it is only the patterns of on/off elements that appear to move.

But patterns don't move either. With both Game of Life patterns and shadows, portions of the surface are *on* and portions of the surface are *off* at any

[6] An analog computer is so called because can be understood as analogous to something else.

given time. That the same or similar on/off configurations appear first at one location and then at another is a consequence of the mechanisms that generate those shadows and patterns. Patterns and shadows themselves are not capable of moving either under their own power or as a result of some force being applied to them. Neither shadows nor patterns can propel themselves. Nor can one push or pull them.

The mechanisms that produce both shadows and Game of Life patterns are fixed-and thoughtless. The Game of Life rules are simply rules for how and when cells turn on and off. The relative motion of the sun, a tree (or other shadow casting object), and a background surface is equally fixed and thoughtless. Yet in both cases, we can use the generated patterns to represent our thoughts.

Using Game of Life patterns we can generate very complex idea. Similarly, we can interpret sun/object/ground shadow patterns to help us think about ideas such as the time of day or day of the year. We had the idea of a day and a year, and we used shadows to help us think about them before we knew what produced them. In both cases, we used patterns generated by fixed rules to help us think.

2.5 Number Computers

Apparently we started to count quite early. Bones with notches carved into them appeared in western Europe 20,000 to 30,000 years ago. There is evidence of the use of a tally system-groups of five notches separated from each other. With tally systems not only did we mark physical materials to help us keep track of numbers (which are also mental events), we also invented ways to make counting easier by the way in which we arranged these markers, i.e., in groups. Soon we invented the abacus.

With these primitive computers we separated the computational process from its dependency on natural processes. Sundials and astronomical masonry depend on the sun and the stars. Counting depends on nothing other than human activity. Once we invented computational devices that were independent of non-human physical processes it was a short step to written notation. By approximately 3,000 BC cuneiform writing on clay tablets using positional notation was known in Babylonia.

2.6 Space Computers

Besides time and numbers, the Pythagoreans in Greece and Euclid in Egypt developed ways to think about space. We know that early geometers thought about construction issues. The straight edge and compass were their (human-powered) thought tools. They used them to externalize, to create representations of, and to manipulate the ideas of straight lines and circles.

2.7 Is It Reasonable to Call Abaci and Geometers' Tools Computers?

Even though abaci and geometers' tools are completely independent of non-human physical processes, i.e., they are entirely dependent on human activity

to make them "run," we feel justified in calling them computers because they are used according to mechanical rules. Even though the source of energy for an abacus is the user, the abacus user follows strict rules-rules which could be automated.

2.8 Thought Tools for Symbol Manipulation

Beyond time, numbers, and space, we have also built thought tools to represent symbolic thoughts and relationships. Sowa [6] describes the Tree of Porphyry.

> The oldest known semantic network was drawn in the 3rd century AD by the Greek philosopher Porphyry in his commentary on Aristotle's categories. Porphyry used it to illustrate Aristotle's method of defining categories by specifying the *genus* or general type and the *differentiae* that distinguish different subtypes of the same supertype.

Another attempt to externalize symbolic thought has been credited to Ramon Lull in the late 13th century. Smart [7] describes it as follows.

> Ramon Lull's logic machine consisted of a stack of concentric disks mounted on an axis where they could rotate independently. The disks, made of card stock, wood, or metal, were progressively larger from top to bottom. As many as 16 words or symbols were visible on each disk. By rotating the disks, random statements were generated from the alignment of words. Lull's most ambitious device held 14 disks.
>
> The idea for the machine came to Lull in a mystical vision that appeared to him after a period of fasting and contemplation. It was not unusual in that day... scientific advances to be attributed to divine inspiration. He thought of his wheels as divine, and his goal was to use them to prove the truth of the Bible...
>
> In "Gulliver's Travels," Swift satirizes the machine without naming Lull. In the story, a professor shows Gulliver a huge contraption that generates random sequences of words. Whenever any three or four adjacent words made sense to-gether, they were written down. The professor told Gulliver the machine would let the most ignorant person effortlessly write books in philosophy, poetry, law, mathematics, and theology.

This may be the first use of non-determinism in computing.

Soon thereafter William of Ockham discovered the foundations of what were to become De Morgan's laws of logic. More specifically, from Sowa [8]:

> (Ockham, 1323) showed how to determine the truth value of compound propositions in terms of the truth or falsity of their components and to determine the validity of rules of inference... in terms of the truth of their antecedents and consequents.

2.9 Thought Tools and the Scientific Process

Clocks, abaci, straight-edges, hierarchies, non-determinism, laws of logic, and other thought tools differ in kind from microscopes, telescopes, and other scientific instruments of observation. The former are intended to allow us to externalize and manipulate our thoughts. The latter allow us to investigate nature-to see what's out there and perhaps to see things that will require new ideas to understand them. Thought tools are constructive; instruments of scientific observation are reductive.

Thus after having convinced ourselves that we are capable of generating our own ideas, an important next step was to realize the necessity of testing our ideas against nature. Simply coming up with an idea is not enough. It's important both to externalize it as a way to work with it and to test it by looking at nature though it. Thus science consists fundamentally of three kinds of activity.

1. Uncovering new facts (observations) about nature.
2. Reverse engineering nature to figure out how nature may have harnessed understood principles to produce the observed facts.[7] Although reverse engineering sounds unglamorous, it is a fundamental activity. Determining that our genome is encoded as a double helix was reverse engineering.
3. Establishing new fundamental principles and then using them as the basis of the reverse engineering process. This occurs only in fundamental physics.

Scientific instruments help us with (1). Thought tools help with (2) and (3).

2.10 The State of the Art of Thought Externalization

Every computer application is a thought tool. The thoughts that are being manipulated are the thoughts that are represented by the conceptual model implemented by the application. More importantly every programming language is a thought tool. Programming languages allows us to externalize in the form of computer programs our thoughts about symbolic behaviors. Since one writes computer applications in programming languages, a programming language is a thought tool for building thought tools, i.e., a thought tool for externalizing thought.

It is important to realize that a programming language is itself a computer application. As a computer application, it implements a conceptual model; it allows its users to express their thoughts in certain limited ways, namely in terms of the constructs defined by the programming language. But all modern programming languages are also conceptually extensible. Using a programming language one can define a collection of concepts and then use those concepts to build other concepts.

We are still learning to use the power of computers to externalize thought. In one way or another, much of software-related research is about developing more

[7] Reductionism has recently received a lot of bad press. As explicated here, the reductionist impulse often leads to the development of important new ideas.

powerful, more specialized, faster, easier to use, or more abstract thought tools. We also develop increasingly powerful languages in which to externalize and work with our thoughts. The more we learn about externalizing our thoughts the higher we ascend the mountain of abstraction and the broader the vistas we see.

Work in externalizing thought includes declarative programming (e.g., logic programming, functional programming, constraint-based programming, rules-based systems such as expert systems, etc.), meta and markup languages such as XML and its extensions and derivatives, the Unified (and Systems) Modeling Language (UML and SysML), and the Semantic Web and the OWL Web Ontology Language for externalizing how we look at the world. With OWL we are working in a tradition that dates back to Porphyry-and before. Domain-specific applications also represent externalization of how we think about those domains. Thought tools for the manipulation of images, sounds, videos, etc. have externalized ways of thinking about those domains.

3 Defining Computation

In this section we turn to the question of how to define *computation*. It is surprisingly difficult to find a well considered definition. The one offered by Eliasmith [9] appears to be the most carefully thought out. Here is his definition and his commentary.

> *Computation.* A series of rule governed state transitions whose rules can be altered. There are numerous competing definitions of *computation*. Along with the initial definition provided here, the following three definitions are often encountered:
>
> 1. Rule governed state transitions.
> 2. Discrete rule governed state transitions.
> 3. Rule governed state transitions between interpretable states.
>
> The difficulties with these definitions can be summarized as follows:
>
> a) The first admits all physical systems into the class of computational systems, making the definition somewhat vacuous.
> b) The second excludes all forms of analog computation, perhaps including the sorts of processing taking place in the brain.
> c) The third necessitates accepting all computational systems as representational systems. In other words, there is no computation without representation on this definition.

Contrary to Eliasmith we suggest the following.

a) The notion of alterable rules is not well defined, and hence all physical systems *are* potentially computational systems.
b) But, it is exactly the fact of interpretability that makes a physical process into a computation. (Eliasmith doesn't explain why he rejects the notion that computation requires interpretation.)

Eliasmith requires that the rules governing some identified state transitions must be alterable in order to distinguish a computation from a naturally occurring process-which presumably follows rules that can't be altered. But all computing that takes place in the physical world is based on physical processes. If we set aside the probabilistic nature of quantum physics, and if we suppose that physical processes operate according to unalterable rules, it's not clear what it means to say that it must be possible to alter a set of rules.

This is not just being difficult. Certainly we all know what it means to say that one program is different from another-that "the rules" which govern a computation, may be altered. But the question we wish to raise is how can one distinguish the altering of a program from the altering of any other contingent element in an environment?[8]

It is the particular program that is loaded into a computer's memory that distinguishes the situation in which one program is being executed from that in which some other program is executing. But a computer's memory is the environment within which the computer's cpu (or some virtual machine) finds itself, and a loaded program defines the state of that environment. The cpu (or the virtual machine) is (let's presume) fixed in the same way that the laws of nature are fixed. But depending on the environment within which it finds itself– i.e., the program it finds in its environment–the cpu operates differently, i.e., it performs a different computation.

This same sort of analysis may be applied to virtually any natural process. When we put objects on a balance scale, the scale's behavior will depend on the objects loaded, i.e., on the environmental contingencies.[9] In both the case of programs loaded into a computer and objects put in the pans of a balance scale, we (the user) determine the environment within which some fixed process (i.e., the rules) proceeds.

This brings us back to our original perspective. A process in nature may be considered a computation only when we use it as a way to work with externalized thought. A physical or otherwise established process-be it the operation of a balance scale, a cpu, the Game of Life, or the sun in motion with respect to trees and the ground-is just what it is, a fixed process.[10] But for almost all processes,[11] whether we create them or they arise naturally, how the process

[8] We don't address the issue of "hard-wired" computations. How fixed must state transitions be before one is no longer willing to say they aren't alterable-and hence not a computation?

[9] When a balance scale compares two objects and returns an "output" (selected from *left-is-heavier, equal-weights,* and *right-is-heavier*), is it performing a computation? It is if we are using it for this purpose. It isn't if we are using it as a designer setting for flower pots.

[10] Of course many processes-such as the operation of a cpu and the operation of a balance scale-are what they are because we built them to be that way-because we anticipated using contingencies that we could control in their environment to help us think.

[11] Some quantum processes may occur on their own without regard to their environment-although even they are environmentally constrained by the Pauli exclusion principle.

proceeds depends on environmental contingencies. When we control (or interpret) the contingencies so that we can use the resulting process to work with our own thoughts, then the process may be considered a computation. This is the case whether we control the contingencies by loading a program into a computer, by placing objects on a balance scale, by establishing initial conditions for the Game of Life, or by giving meaning to shadows cast by trees.

Consequently we agree with Eliasmith that it must be possible to alter a process for it to be considered a computation, but we would express that condition in other words. For a process to be considered a computation there must be something contingent about the environment within which it operates which both determines how it proceeds and determines how we interpret the result.

In other words, we can always separate a computational process into its fixed part and its contingent or alterable part. The fixed part may be some concrete instances of the playing out of the laws of nature – in which case the contingent environment is the context within which that playing out occurs. Or it may be the operation of a cpu-in which case the contingent environment is the memory which contains the program that is being executed. Or it may be the operation of a program that a cpu is executing – in which case the contingent environment is the input to that program. A computation occurs when we alter the contingencies in the environment of an fixed process as a way to work with our thoughts.

This perspective contrasts traditional (theoretical) computation with real-world computation. Normally, one thinks of a (theoretical) computation as a contingent process-one which is defined in a programming language. Like a Turing Machine it runs for free. We contrast this with real-world computations, which result from non-contingent processes which have built-in energy sources and that operate in contingent environments.

3.1 Non-algorithmic Computing

A corollary of the preceding is that all computation performed by real-world processes are environmentally driven. Computing involves configuring environmental contingencies, i.e., setting up an environment within which a process (or multiple processes) will play themselves out. We refer to this as *non-algorithmic computing* because one's focus is on how an environment will shape a process rather than on a specific sequence of steps that the shaped process will take. No explicit algorithm is involved. Most of what we call unconventional computation is non-algorithmic.

It may seem ironic that what we think of as conventional computation is a constrained form of unconventional computation. We are attracted to it because its single threaded linearity makes it easy to manage. But nature is not linear. Any computer engineer will confirm how much work it takes to shape what really goes on in nature into a von Neumann computer. Even more ironically, we then turn around and use conventional single-threaded computers to simulate nonlinear unconventional computation. One might say that a goal of this conference is to eliminate the von Neumann middle man – to find ways to compute, i.e., to externalize our thoughts, by mapping them more directly onto the forces of

nature operating in constrained environments. The operations performed by the forces are nature are real–world individual Turing Machines. A general purpose computer is a real-world Universal Turing Machine.

3.2 Turing *Machines* vs. Turing *Computability*

Why can't we look to Turing Machines (and their equivalents) for a definition of computation which is defined independently of thought? Turing Machines, recursive functions, and formally equivalent models rely on the notions of symbols and symbol manipulation, which are fundamentally mental constructs. Eliasmith's definition doesn't – although his definition does depend on the notion of rule-governed state transitions, which appears difficult to define non-symbolically. The saving grace of states and state transitions is that they are intentional; they are our way of thinking about what happens in nature. Symbol manipulation is a purely mental activity.

But Turing Machines – and their Church-Turing Thesis equivalents – offer an important insight. They identify symbol manipulation to be what we intuitively think of as computational activity. The Turing Machine model is our way of externalizing an entire class of mental activities, the class that we intuitively identify as computational.

In saying this we are separating (a) the sorts of *computational activities* characterized by Turing Machines, i.e., the Turing Machines themselves, from (b) the *class of functions* that these models compute, i.e., Turing computability. The various models of computational activities are all defined constructively, i.e., in terms of the operations one may perform *when constructing a computational procedure*. Furthermore, the equivalence proofs among the standard models are also constructive. We can constructively transform any Turing Machine into a recursive function and vice versa. Turing Machines, recursive functions, etc. are equivalent *as programming languages*.

Computability theory then takes the generic class of software defined in this way and applies it to the task of computing functions. But this second step isn't necessary. What's important about the Church-Turing Thesis is not the class of functions that can be computed but the possible programs one may write, i.e., that Turing Machines, recursive functions, etc. are our way of externalizing a fundamental mode of thought. Our revised version of the Church-Turing Thesis is that to be considered rigorous a thought process must, at least in principle, be expressible as a software.

4 Agent-Based Computing

The Turing Machine model is single threaded-as are the single processor von Neumann computers that we built based on it. But many of our computer science (and other) thought models are either parallel, asynchronous, or non-deterministic. Not all rigorously defined models are linear and single threaded. Yet we have been unable to build thought tools to help us externalize these kinds of non-deterministic

computational ideas. Attempts to perform non-deterministic computations on a single-threaded computer result in unrealizable demands for resources.[12]

Four decades ago agent-based computing, an intermediate form of computational framework, began to emerge Dahl [10]. Agent-based computing is an attractive form of asynchronicity because it relies on manageable parallelism-asynchronous computing threads that don't result in an unrealizable demand for computing re-sources. Its price is *chaotic asynchronicity*: minimally different event orderings may yield different results.

4.1 Open and Far-from-Equilibrium Computing

Goldin and Wegner [11] have defined what they called *persistent Turing Machines* (and elsewhere *interaction machines*). These are Turing Machines that perform their computations over an indefinite period-continually accepting input and producing output without ever completing what might be understood as a traditional computation. Results of computations performed after accepting one input may be retained (on the machine's "working tape") and are available when processing future inputs. Although Wegner's focus is not on agent-based computing, his model is essentially that: agents which interact with their environments and maintain information between interactions. From here on we use *agent* to refer to an object that embodies a program.

Goldin and Wegner claim that their "interactive finite computing agents are more expressive than Turing machines." There has been much debate about this claim. We believe that to ask about the level of computability of agents is to ask the wrong question. We believe that what Wegner and Goldin have done is to have taken implicitly the same stance that we took explicitly above, i.e., to distinguish between the programs one can write and the functions those programs can compute. In making this implicit distinction Wegner and Goldin point out that one need not think of the program that a Turing Machine embodies in functional terms, i.e., as closed with respect to information flow. One can also think of a Turing Machine as open with respect to information flow. This parallels the distinction in physics between systems that are closed and open with respect to energy flows. Wegner has outlined this position most recently in [12]. Complex systems are famously far from equilibrium with respect to environmental energy flows. Wegner and Goldin's interaction machines (and agents in general) are similarly far from equilibrium with respect to information flows.

What might one gain from being open to information flows? An illustrative example is Prisoner's Dilemma (PD). If one were to develop an optimized PD player for a one-shot PD exchange-since it's one shot, the system is closed-it will Defect. Playing against itself, it will gain 1 point on each side-using the usual scoring rules. If one were to develop an optimized PD player to engage in an iterative PD sequence-the system is open-it will Cooperate indefinitely (presumably by playing a variant of Tit-for-Tat), gaining 3 points on each side at each time.

[12] If we get it to work on a useful scale quantum computing may be the first such thought tool.

Thus the same problem (PD) yields a different solution depending on whether one's system is presumed to be open or closed with respect to information flows.

4.2 Agents and Their Environments

Computation involves the interaction of a process with its environment. In all cases with which we are familiar, the environment is modeled as a simply structured collection of symbols, e.g., a tape, a grid, etc. None of these models are adequate when compared to the real-world environment within which we actually find ourselves. We do not know how to model the multi-scalar face that nature presents to us – but almost certainly it won't be as a tape or a grid.

- In our actual environment new entities and new kinds of entities may come into existence. We are able to perceive and interact with them. We are aware of no formal environmental framework capable of representing such phenomena.
- We do not understand the ultimate set of primitives – if indeed there are any – upon which everything is built.

We have referred [4] to these problems as the difficulty of looking upwards and the difficulty of looking downwards respectively.

We are just beginning [4] to understand the nature of entities and of the multi-scalar environment within which they exist. That environment involves entities on multiple levels, but it also involves forces at only the most primitive level. All other interactions are epiphenomenal. This is not simply a layered hierarchy, although it has some layered hierarchy properties.

Given our lack of understanding about these issues it is not surprising that we have not been able to develop a formal model of such an environment. Thus a fundamental open problem in computing is to develop a formal model of an environment that has the same sorts of multi-scalar properties as our real-life environment.

Our revised version of the Church-Turing Thesis gives us confidence that our current understanding of agents as entities that embody programs is reasonably close to how we think about thinking. We are still quite far from the goal of formalizing appropriate environments within which such agents should be situated.

4.3 The Inevitable Evolution and Acceleration of Intelligence

As we saw in the PD example, thinking in terms of open computation model leads to different results from thinking in terms of closed models. Yet both use the same class of possible programs-whatever is programmable in a general purpose programming language. Since open computation models include the class of Oracle machines, computability doesn't seem like the appropriate perspective when analyzing these systems. Is there another approach? We suggest that the notion of results achieved is more relevant. In the PD case, the result achieved is the number of points scored.

Under what circumstances would it make sense to think of an agent in terms of results achieved? In [4] we discuss the nature of emergent entities. Static entities persist at an energy equilibrium in energy wells; but the more interesting dynamic entities persist only so long as they can extract energy from their environment.

Unfortunately most agent-based computer models either ignore the issue of energy or treat it very superficially. We believe that an integrated theory of energy and information would clarify how information flows enable evolution. A real-world agent would be a dynamic entity that embodied some software. If, through a random mutation, such an entity developed an enhanced ability to extract information from its environment then it will be more likely to survive and reproduce. What evolves in this model is an enhanced ability to extract information from the environment. The need of dynamic entities for energy drives evolution toward increasingly more powerful informational processing capabilities.[13]

In this picture, information is being extracted from the environment at two levels. Each individual extracts information from the environment, which it processes as a way to help it find energy. Very simple real-life examples are plant tropisms and bacterial tendencies to follow nutrient gradients. More interestingly, the evolutionary process itself extracts information from the environment, which it then encodes (in DNA) as the "program" which individual agents use to process information from their environment. Thus the real intelligence is in the program, and the real information extracting activity is the evolutionary process that constructs the program.[14]

Can evolution itself evolve? Is there something that will enable an entity to extract information from the environment more effectively? Modern society stores information about how to process information from the environment as science. Can we go beyond science? Can the scientific process itself evolve? Science is the process of constructing mechanisms to extract and process information from the environment. Since science is a thought process, tools that enable us to externalize and improve our scientific thought processes will enhance our ability to do science.

5 Conclusion

An environmentally sophisticated agent-based paradigm involves agents, each of which has the computing capability of a Turing machine, situated in an environment that reveals itself reluctantly. Such an agent in a real-world environment is like an Oracle machine, with nature as the oracle. Combining agents with dynamic entities yields real-world agents, which (a) must extract energy from their environment to persist and (b) embody software capable of processing information

[13] This seems to answer the question of whether evolution will always produce intelligence. It will whenever increased intelligence yields enhanced access to energy.

[14] Systems that have attempted to model this process have failed because their environments are too poor.

flows from the environment. The agent-based thesis is that this paradigm represents how, at the start of the 21st century, we think about our place with the world.

Acknowledgment. Many of the ideas in this paper were elaborated in discussions with Debora Shuger.

References

1. IJUC Editorial Board, Preface [to the first edition] from the Editorial Board, *International Journal of Unconventional Computing*, 1, 1 (2004), 1–2.
2. P. Wegner, D. Goldin, Interaction, Computability, and Church's Thesis, *British Computing Journal*, 1999.
3. Google, Help Center, undated. Accessed March 12, 2006: `http://mail.google.com/support/bin/answer.py?answer= 29433&query=faq&topic=0&type=f`.
4. R. Abbott, Emergence Explained, to appear in *Complexity.*
5. L. Albacete, For the Love of God, *New York Times*, Feb 3, 2006.
6. J. Sowa, Semantic Networks, `http://www.jfsowa.com/pubs/semnet.htm`, June 2, 2006, revised from S.C. Shapiro (ed)., *Artificial Intelligence-Encyclopedias*, John Willy & Sons, Inc, 1992, 1493–1511.
7. Smart Computing, undated. Accessed February 20, 2006: `http://www.smartcomputing.com/editorial/dictionary/detail.asp?guid=&searchtype=1&DicID=18707&RefType=Encyclopedia`.
8. J. Sowa, Existential Graphs, 2002. Accessed February 20, 2006: `http://www.jfsowa.com/peirce/ms514.htm`.
9. C. Eliasmith, Computation, entry in *Dictionary of the Philosophy of Mind*, C. Eliasmith (ed.), May 11, 2004 `http://artsci.wustl.edu/∼philos/MindDict/entry.html`.
10. O.-J. Dahl, K. Nygaard, Simula: an ALGOL-based simulation language, *Communications of the ACM*, 9, 9 (1966), 671–678.
11. D. Goldin, P. Wegner, Behavior and Expressiveness of Persistent Turing Machines, *Computer Science Technical Report*, Brown Univ., 1999, `http://www.cs.montana.edu/∼elser/turing papers/Behavior and Expressiveness of PTM.pdf`.
12. P. Wegner et al., The Role of Agent Interaction in Models of Computing, *Electronic Notes in Theoretical Computer Science*, 141 (2005).

Peptide Computing – Universality and Theoretical Model

M. Sakthi Balan[1] and Helmut Jürgensen[1,2]

[1] Department of Computer Science
The University of Western Ontario
London, Ontario, Canada, N6A 5B7
sakthi@csd.uwo.ca
[2] Institut für Informatik, Universität Potsdam,
August-Bebel-Str. 89, 14482 Potsdam, Germany

Abstract. We present a new simulation of Turing machines by peptide-antibody interactions. In contrast to a simulation presented previously, this new technique simulates the computation steps automatically and does not rely on a "look-and-do" approach, in which the Turing machine program would be interpreted by an extraneous computing agent. We determine the resource requirements of the simulation. Towards a precise definition for peptide computing we construct a new theoretical model. We examine how the simulations presented in this paper fit this model. We prove that a peptide computing model can be simulated by a Turing machine under certain conditions.

1 Introduction

To use peptide-antibody interactions as a model of computation was proposed by H. Hug et al. [5]. In [1] it was shown that the model is universal; the proof of this result uses a simulation of the computation of a Turing machine by peptide-antibody interactions. While that simulation is clearly correct, some of its properties are not convincing intuitively. We address some of these issues in this paper.

A peptide is a sequence of amino acids attached by covalent bonds called peptide bonds. A peptide consists of recognition sites, called *epitopes,* for the antibodies. A peptide can contain more than one epitope for the same or different antibodies. With each antibody, which attaches to a specific epitope, a binding power is associated, called its *affinity.* When antibodies compete for recognition sites – which may overlap in the given peptide – then the antibodies with greater affinity have a higher priority.

Replacing an antibody by one with a greater affinity can be considered a computational step in a re-writing process: the antibody with smaller affinity is removed – *affinity-based removal* – and an antibody with greater affinity attaches to the epitope which became free. Antibodies can also be removed by adding excess epitopes – *epitope-based removal.* In the sequel, the term *peptide computing* refers to computational processes based on these as elementary operations.

C.S. Calude et al. (Eds.): UC 2006, LNCS 4135, pp. 57–71, 2006.

In [5], it was shown how to solve the satisfiability problem using peptide computing. To show that peptide computing is universal, a simulation of Turing machines by peptide computing was presented in [1]. The simulation uses epitope-based removal. Thus peptide computing is at least as powerful as Turing computing. Whether the converse holds true depends on the precise limitations imposed on the model for peptide computing and, in particular, on the way how affinity is modelled.

The simulation of a Turing machine $\mathcal{M}$ by peptide computing as presented in [1] is not totally convincing in the following sense:

First, the simulation relies on an extraneous computing agent to interpret the computation of the Turing machine step-by-step and to simulate the observed behaviour by appropriate peptide-antibody interactions. This agent could, for instance, be a human taking notes of the steps of the Turing machine and selecting and mixing the required molecules. Thus, the simulation is not automatic; it is a "look-and-do" method. "Extraneous computing agents" are part of any formal model of computation, even the Turing model, – usually hidden in the definition of computational steps. The important issue is to limit the "power" of this agent.

Second, the simulation requires an unbounded number of epitopes and antibodies. The length of the peptide sequence in terms of epitopes and the number of antibodies needed are both approximately proportional to the amount $s_{\mathcal{M}}$ of space used for the Turing computation.

As mentioned above, the size of the alphabets used in the simulation is not bounded, but depends on the size of the input and the specific computation. To encode the antibodies and epitopes over a finite alphabet would increase the resource and time requirements. It seems that a comma-free encoding, even a solid code, would be needed to identify the locations of the epitopes in the peptide uniquely (see [7] for a survey of the relevant properties of codes). In this case, for s epitopes or antibodies to be encoded over an alphabet with r symbols, $r > 1$, about $c_1 \log_r s$ symbols are needed per epitope or antibody for some $c_1 > 1$. Thus, the size of the peptide is $\Theta(s_{\mathcal{M}}(w) \cdot \log s_{\mathcal{M}}(w))$. Moreover, also the time bound increases by a factor of $\Theta(\log s_{\mathcal{M}}(w))$ to account for the search for and the handling of, the appropriate epitopes and antibodies. While such an encoding is mathematically feasible, it is not clear at this point, whether this idea would work in the bio-chemical setting.

In the present paper we address the first of the short-comings of the previous simulation. We propose a method for encoding the definitions of the transition in the interactions between peptides and antibodies. We prepare the peptide sequences and antibodies in such a way that they select and execute the transitions automatically. The new model relies primarily on affinity-based removal, with epitope-based removal restricted to intermediate operations in the simulation. Moreover, we present a formal model of peptide computing which enables us to express the converse simulation, that of a peptide system on a Turing machine. Within this model one can formulate the precise conditions for this simulation to be possible.

Whether our formal model is adequate for the bio-chemical realities will not be discussed in this paper. In many ways, the abstraction is an extreme simplification. Hence one may be justified to conclude for our work that peptide computing is at least as powerful as Turing computing, if not more powerful. It is not clear whether the constraints making the models equivalent can be made to hold true bio-chemically. Moreover, it is certainly also important to investigate the actual usability of a peptide computing system.

Our paper is structured as follows. In Section 2 we review some notational conventions. The new proposal is presented in some detail in Section 3. In Section 4 we construct a theoretical model for peptide computing. We also examine how the proposed new simulation fits with our theoretical model in the Section 5. In Section 6 we present the simulation of a peptide computing model by a Turing machine under certain conditions. We summarize and discuss the results in Section 7. The reader should consult [1] to compare the simulation methods and to determine how the former simulation fits in the framework of the new model.

2 Notation, Basic Notions

For a set S, $|S|$ denotes the cardinality of S. When S is a singleton set, $S = \{x\}$ say, we write x instead of $\{x\}$. For sets S and T, consider a relation $\varrho \subseteq S \times T$. Then ϱ^{-1} is the relation $\varrho^{-1} = \{(t,s) \mid (s,t) \in \varrho\}$ and, for $s \in S$, $\varrho(s) = \{t \mid (s,t) \in \varrho\}$. We use the notation $\varrho : S \xrightarrow{\circ} T$ to denote a partial mapping of S into T. In that case *dom* ϱ is the subset of S on which ϱ is defined. The notation $\varrho : S \to T$ means that S is a total mapping of S into T, hence *dom* $\varrho = S$ in this case.

Let S be a non-empty set. A *multiset* on S is a pair $M = (I, \iota)$ where I is a set, the index set, and ι is a mapping of I into S, the *indexing.* A multiset M is non-empty, if I is non-empty; it is finite if I is finite. For $s \in S$, the number $|\{i \mid i \in I, \iota(i) = s\}|$ is the *multiplicity* of s. When I is countable, we write $M = \{m_i \mid i \in I\}$ where $m_i = \iota(i)$ is implied. With this notation, it is possible that $m_i = m_j$ while $i \neq j$ for $i, j \in I$. We use the standard symbols for set theoretic operations also for multisets. However, on multisets, union is disjoint union and both intersection and set difference take multiplicities into account. Formally this can be handled by appropriate operations on the index sets.

By $\mathbb{N}$ and $\mathbb{N}_0$ we denote the sets of positive integers and of non-negative integers, respectively. The set $\mathbb{B} = \{0, 1\}$ represents the set of Boolean values. For $n \in \mathbb{N}_0$, $\mathfrak{n} = \{i \mid i \in \mathbb{N}_0, i < n\}$. Thus, for example, $\mathfrak{o} = \emptyset$, $\mathfrak{1} = \{0\}$ and, in general, $\mathfrak{n} = \{0, 1, \ldots, n-1\}$. By $\mathbb{R}$ we denote the set of real numbers, and $\mathbb{R}_+ = \{r \mid r \in \mathbb{R}, r \geq 0\}$.

An alphabet is a non-empty set. Let X be an alphabet. Then X^* is the set of all words over X including the empty word λ, and $X^+ = X^* \backslash \{\lambda\}$. For a word $w \in X^*$, $|w|$ is its length. Any word $u \in X^*$ with $w \in uX^*$ is a prefix of w; let $\mathrm{Pref}(w)$ be the set of prefixes of w; the words in $\mathrm{Pref}_+(w) = \{u \mid u \in X^+, w \in uX^+\}$ are the proper prefixes of w. Similarly, a word $u \in X^*$ with $w \in X^*uX^*$ is an infix of

w, $\mathrm{Inf}(w)$ is the set of infixes of w and $\mathrm{Inf}_+(w) = \{u \mid u \in X^+, u \in \mathrm{Inf}(w), u \neq w\}$ is the set of proper infixes of w. A language over X is a subset of X^*. For a language L over X and $Y \in \{\mathrm{Pref}, \mathrm{Pref}_+, \mathrm{Inf}, \mathrm{Inf}_+\}$, $Y(L) = \bigcup_{w \in L} Y(w)$.

Let L be a language over X and $w \in X^*$. An *L-decomposition* of w is a pair of sequences $(u_0, u_1, \ldots, u_k)$, $(v_0, v_1, \ldots, v_{k-1})$ of words in X^* such that $u_0 v_0 u_1 v_1 \cdots v_{k-1} u_k = w$, $v_0, v_1, \ldots, v_{k-1} \in L$ and $u_0, u_1, \ldots, u_k \notin X^* L X^*$. A language in X^+ such that every word has a unique L-decomposition is called a *solid code* [7]. Consider $w \in X^+$ of length n, say $w = x_0 x_1 \cdots x_{n-1}$ with $x_i \in X$ for $i = 0, 1, \ldots, n-1$. An L-decomposition of w as above can be specified by a set of pairs $\{(i_l, j_l) \mid l = 0, 1, \ldots, k-1\}$ such that, for $l = 0, 1, \ldots, k-1$, $v_l = x_{i_l} x_{i_l+1} \cdots x_{j_l}$. Let $\partial_L(w)$ be the set of L-decompositions when represented in this way. Let $\mathcal{D}(L) = \{(w, d) \mid w \in X^*, d \in \partial_L(w)\}$ be the set of words together with all their L-decompositions.

A (deterministic) Turing machine is a construct $\mathcal{M} = (Q, \Sigma, \delta, q_0, F, \flat)$ such that Q is a finite non-empty set of states, Σ is a finite non-empty alphabet with $Q \cap \Sigma = \emptyset$, $q_0 \in Q$ is the start state, $F \subseteq Q$ is the set of final states, $\flat$ is the blank symbol, $\flat \notin \Sigma \cup Q$, and $\delta : Q \times (\Sigma \cup \flat) \to Q \times (\Sigma \cup \flat) \times \{\mathtt{L}, \mathtt{R}\}$ is the (partial) transition function. Here `L` and `R`, for 'left' and 'right', denote the directions of the movement of the read-write head on the Turing tape. We assume that $\delta(q, a)$ is undefined for all $q \in F$. We generally denote the movement of the head by $\mathtt{d} \in \{\mathtt{L}, \mathtt{R}\}$. For more details on Turing machines see [4]. If $\mathcal{M}$ is Turing machine we represent the language accepted by $\mathcal{M}$ as $L(\mathcal{M})$. We denote the space and time complexity functions of $\mathcal{M}$ as $s_{\mathcal{M}}$ and $t_{\mathcal{M}}$.

In the sequel, it is sometimes convenient to have special symbols for states and inputs. In this case, let $Q = \{q_0, q_1, \ldots, q_{m-1}\}$ and $\Sigma = \{a_0, a_1, \ldots, a_{l-1}\}$.

3 Automatic Simulation of a Turing Machine by Peptides

In the simulation presented in [1], the transition function of the Turing machine is not encoded in the peptide system. To remedy this we not only need such an encoding but also a method for looking up instructions and for their execution.

Theorem 1. *Let $\mathcal{M} = (Q, \Sigma, \delta, q_0, F, \flat)$ be a Turing machine. There is a simulation of $\mathcal{M}$ by peptide computing with the following properties:*

(1) There is a constant $c > 0$, independent of $\mathcal{M}$, such that the number of peptide-antibody interactions needed for the simulation of a computation step of $\mathcal{M}$ is no greater than c. As a consequence, the number of peptide antibody interactions needed for the simulation of a computation of $\mathcal{M}$ on input $w \in \Sigma^*$ is no greater than $c \cdot t_{\mathcal{M}}(w)$.

(2) The number of peptide sequences needed for the simulation of a computation of $\mathcal{M}$ on input $w \in \Sigma^*$ is in $\Theta(s_{\mathcal{M}}(w))$; moreover the number of antibodies needed is in $\Theta((|Q| + |\Sigma|) \cdot s_{\mathcal{M}}(w))$.

Proof Idea. We assume that $\mathcal{M}$ has only a single final state. This restriction is easily lifted at the cost of a more complicated argument.

We use five sets of peptide sequences: a set T to simulate the cells of the tape of $\mathcal{M}$; P to hold the program of $\mathcal{M}$; S to synchronize the operation; and two sets I_1 and I_2 for carrying out intermediate steps.

Each sequence in T consists of four epitopes and uniquely denotes a cell on the tape of $\mathcal{M}$. For cell i, the epitopes are $e^{(T)}_{i,1}, \ldots, e^{(T)}_{i,4}$ such that the peptide is $p^{(T)}_i = e^{(T)}_{i,1} x_i e^{(T)}_{i,2} y_i e^{(T)}_{i,3}$ with $e^{(T)}_{i,4} = x_i e^{(T)}_{i,2} y_i$ for some words x_i and y_i. The set T, with antibodies attached to the epitopes represents the respective configuration of the Turing machine.

The set P contains a peptide sequence for each pair $(q, a) \in Q \times \Sigma$ for which $\delta(q, a)$ is defined. It will capture the transition applied when $\mathcal{M}$ is in state q and reading the symbol a. A typical peptide sequence in P has three epitopes $e^{(P)}_{(q,a),1}$, $e^{(P)}_{(q,a),2}$ and $e^{(P)}_{(q,a),3}$, and has the form $p^{(P)}_{(q,a)} = e^{(P)}_{(q,a),1} e^{(P)}_{(q,a),2}$ with $e^{(P)}_{(q,a),3} \in \mathrm{Inf}_+(p^{(P)}_{(q,a)})$ and which intersects both $e^{(P)}_{(q,a),1}$ and $e^{(P)}_{(q,a),2}$.

The set S contains a peptide sequence for each pair $(q, a) \in Q \times \Sigma$ for which $\delta(q, a)$ is defined. It will control the execution of a transition step. A peptide sequence in S has the form $p^{(S)}_{(q,a)} = z_{(q,a)} e^{(S)}_{(q,a),1} e^{(S)}_{(q,a),2}$. It has the three epitopes $e^{(S)}_{(q,a),1}$, $e^{(S)}_{(q,a),2}$ and the whole sequence itself is an epitope.

Finally the sets I_1 and I_2 contain peptide sequences as follows. Each sequence in I_1 contains epitopes $e^{(I_1)}_{(q,a),1}$ and $e^{(I_1)}_{(q,a),2}$ and is represented by $p^{(I_1)}_{(q,a)} = e^{(I_1)}_{(q,a),1} e^{(I_1)}_{(q,a),2}$. All the peptide sequences in I_1 are initialized with antibodies $A_{q,a}$ which bind to the epitope $e^{(I_1)}_{(q,a),1}$. Each sequence in the set I_2 contains only one epitope, namely $e^{(I_2)}_{(q,a)}$, and is represented by $p^{(I_2)}_{(q,a)} = e^{(I_2)}_{(q,a)}$.

If, as in [1], one starts the simulation with a space bound s initially, one has: $|T| = s$ using $4s$ epitopes; $|P| = |Q| \cdot |\Sigma|$ using $3|P|$ epitopes; $|S| = |Q| \cdot |\Sigma|$ using $3|S|$ epitopes; $|I_1| = |Q| \cdot |\Sigma|$ with $2|I_1|$ epitopes; and $|I_2| = |Q| \cdot |\Sigma|$ with $|I_2|$ epitopes. Of course, each of the sets would contain multiple copies of each peptide sequence.

We now describe the encoding of the transition function δ of $\mathcal{M}$. Suppose $\delta(q, a) = (q', a', \mathtt{D})$ for $\mathtt{D} \in \{\mathtt{L}, \mathtt{R}\}$. Then, we have a peptide sequence $p^{(P)}_{(q,a)}$ in P with antibodies $A_{q'}$ and $A_{a',\mathtt{D}}$ attached to it at epitopes $e^{(P)}_{(q,a),1}$ and $e^{(P)}_{(q,a),2}$, respectively. Thus each sequence in P encodes the transition for state q and symbol a; The antibodies $A_{q'}$ and $A_{a',\mathtt{D}}$ need to be 'read,' that is, removed, to execute the transition. To achieve this we need to use the sets I_1, I_2, and S as explained further below. If $q' \in F$ then the antibody $A_{q'}$ will be a labelled one; this helps to know that the simulation has halted.

Now we describe the encoding of a configuration of $\mathcal{M}$. For each cell i we record its contents, what its neighbours are and, possibly the state of $\mathcal{M}$ if the cell is currently the one being scanned. For each cell i we need antibodies A_i which attach to the epitopes $e^{(T)}_{i+1,1}$ and $e^{(T)}_{i-1,3}$; moreover, for each $a \in \Sigma$ we need an antibody A_a which can attach to $x_i e^{(T)}_{i,2} y_i$. Thus, if $a \in \Sigma \cup \flat$ is the current contents of cell i, then $p^{(T)}_i$ has A_{i-1}, A_a and A_{i+1} attached to its epitopes $e^{(T)}_{i,1}$,

$e^{(T)}_{i,4}$ and $e^{(T)}_{i,2}$, respectively. We assume, as before, that peptide sequences for enough cells are available to conduct the computation. Those not occupied by input symbols are initialized to $\flat$.

For $q \in Q$ and $a \in \Sigma \cup \flat$, the antibodies A_q and (A_a or $A_{a,\mathtt{D}}$) can attach to the epitopes $e^{(S)}_{(q,a),1}$ and $e^{(S)}_{(q,a),2}$ in S, respectively.

1. For sequences in T: We need antibodies $A_{a,\mathtt{D}}$ and A_j. The epitope of $A_{a,\mathtt{D}}$ is $x_i e^{(T)}_{i,2} y_i$. The epitopes of A_j are $e^{(T)}_{j,2}$, $e^{(T)}_{k,3}$ and $e^{(T)}_{l,1}$ where $k = j-1$ and $l = j+1$ with more affinity for A_j to the epitope $e^{(T)}_{j,2}$. The affinity of A_j is greater than that of $A_{a,\mathtt{D}}$.
2. For sequences in S: We need antibodies $B_{q,a}$. The epitope of $B_{q,a}$ is $p^{(S)}_{(q,a)}$. The antibodies A_q and $A_{a,\mathtt{D}}$ from T also attach to this sequence. The epitopes for A_q and $A_{a,\mathtt{D}}$ are $e^{(S)}_{(q,a),1}$ and $e^{(S)}_{(q,a),2}$ respectively. The affinity of $B_{q,a}$ is greater than that of antibodies A_q and $A_{a,\mathtt{D}}$.
3. For sequences in I_1: We need antibodies $A_{q,a}$ and $B_{a,\mathtt{D}}$. The epitope of $A_{q,a}$ is $e^{(I_1)}_{(q,a),1}$. The epitope of $B_{a,\mathtt{D}}$ is $e^{(I_1)}_{(q,a),2}$. The antibodies A_q and $A_{a,\mathtt{D}}$ from T also attach to this sequence. The epitope of A_q is $e^{(I_1)}_{(q,a),1}$ and the epitope for $A_{a,\mathtt{D}}$ is $e^{(I_1)}_{(q,a),2}$. The affinity of A_q is greater than that of $A_{q,b}$ for all $b \in \Sigma$. The affinity of $B_{a,\mathtt{D}}$ is greater than that of $A_{a,\mathtt{D}}$.
4. For sequences in I_2: The antibodies $A_{q,a}$ and $A_{a,\mathtt{D}}$ attach to the sequences in I_2. The epitope for both of them is $e^{(I_2)}_{(q,a)}$. The affinity of $A_{a,\mathtt{D}}$ is greater than that of $A_{q,a}$
5. For sequences in P: We need antibodies A_q and $A_{a,\mathtt{D}}$ which are initialized to these. The antibodies $A_{q,a}$ also attach to these. The epitopes of A_q, $A_{a,\mathtt{D}}$ and $A_{q,a}$ are $e^{(P)}_{(q,a),1}$, $e^{(P)}_{(q,a),2}$ and $e^{(P)}_{(q,a),3}$. The affinity of $A_{q,a}$ is greater than that of both the antibodies $A_{\bar{q}}$ and $A_{\bar{a},\bar{\mathtt{D}}}$ provided there is a transition $\delta(q,a) = \{(\bar{q},\bar{a},\bar{\mathtt{D}})\}$.

We can now define the simulation of a step of $\mathcal{M}$ by peptide computing. Each such step consists of a cycle of reactions which is initiated by having antibodies A_q for the current state and antibodies A_i for the current cell floating; moreover, we assume that no antibodies are attached to the peptide sequences in S and I_2. Thus, to start the computation, we add antibodies A_{q_0} and A_1 corresponding to the configuration in which $\mathcal{M}$ is in the initial state q_0 and is reading the first cell.

Suppose now that the floating antibodies are A_q and A_i and that the antibody A_a is attached to $p^{(T)}_i$ where $a \in \Sigma \cup \flat$. Then A_i attaches to the epitope $e^{(T)}_{i,2}$ by greater affinity and removes A_a. Hence the two antibodies A_q and A_a attach to their respective epitopes $e^{(S)}_{(q,a),1}$ and $e^{(S)}_{(q,a),2}$ in S. The presence of two antibodies binding to a peptide sequence S denotes the fact the machine is about to select the transition from the sequences in P_1 and P_2.

Now we flush out all the unnecessary epitopes and antibodies still floating in the liquid.

Next, we add antibodies $B_{p,b}$ for all $p \in Q$ and $b \in \Sigma$ which have a greater affinity than the corresponding A_p and A_b. They attach to the epitopes $z_{(q,a),1}e^{(S)}_{(q,a),1}e^{(S)}_{(q,a),2}$ for $a \in \Sigma \cup \flat$. This will remove the antibodies A_q and A_a from the sequence in S.

The antibody A_q which is removed from the sequence in S attaches to a sequence in I_1 to the epitope $e^{(I_1)}_{(q,a),1}$ with greater affinity and removes the initialized antibodies $A_{q,b}$ for all $b \in \Sigma$. The antibody A_a attaches to the epitope $e^{(I_1)}_{(q,a),2}$. The antibodies $A_{q,b}$ attach to the sequences in I_2 with the epitope $e^{(I_2)}_{(q,a)}$. Now we add antibodies B_b for all $b \in \Sigma$ and these attach to the epitopes $e^{(I_1)}_{(q,a),2}$ with greater affinity and remove the antibody A_a. The antibody A_a then attach to sequences in I_2 with higher affinity to the epitopes $e^{(I_2)}_{(q,a)}$ and removes the antibody $A_{q,a}$. Hence with these sequence operations two distinct antibodies A_q and A_a denoting the state of the system and the symbol to be read has given rise to a single antibody $A_{q,a}$ which denotes both the state and the symbol to be read. This takes care of circular arguments arising from cycles in $\mathcal{M}$.

Now $A_{q,a}$ attaches to the epitope $e^{(P)}_{(q,a),3}$ with greater affinity and removes the antibodies $A_{\bar{q}}$ and $A_{\bar{a},\bar{\mathtt{D}}}$ where $\bar{q}$ and $\bar{a}$ are such that $\delta(q,b) = (\bar{q},\bar{b},\bar{\mathtt{D}})$. Hence the antibodies $A_{\bar{q}}$ and $A_{\bar{b},\bar{\mathtt{D}}}$ which were previously initialized with this sequence are now set free. For this to work, $A_{q,a}$ is assumed to have a greater affinity than both $A_{\bar{q}}$ and $A_{\bar{a},\bar{\mathtt{D}}}$ for all $\bar{q} \in Q$, $\bar{B} \in \Sigma \cup \flat$ and $\bar{\mathtt{D}} \in \{\mathtt{L},\mathtt{R}\}$.

Hence the system has selected the correct antibodies corresponding to the next state as $A_{\bar{q}}$ and the symbol to be rewritten as $A_{\bar{a},\bar{\mathtt{D}}}$. The antibody $A_{\bar{q}}$ attaches to the sequence in S and waits for the antibody denoting the next symbol. Here we add excess epitopes $e^{(T)}_{i,2}$ which will eventually remove antibody A_j (supposing the j^{th} cell has been read by $\mathcal{M}$) from the peptide sequence $p^{(T)}_j$. Now the antibody $A_{\bar{a},\bar{\mathtt{D}}}$ attaches to the sequence in T to the epitope $x_i e^{(T)}_{i,2} y_i e^{(T)}_{i,3}$ if $\bar{\mathtt{D}} = \mathtt{R}$ or to the epitope $e^{(T)}_{i,1} x_i e^{(T)}_{i,2} y_i$ if $\bar{\mathtt{D}} = \mathtt{L}$. This in turn removes the antibody A_{j+1} (if it is a right move) or A_{j-1} (if it is a left move) which will bind to the next peptide sequence in the epitope $e^{(T)}_{i,2}$ and remove the antibody denoting the next symbol to be read from the sequence, say A_b. The antibody A_b attaches to the sequence in S. Thus the system is ready for the next transition. This process continues until a labelled antibody attaches to a sequence in S. After this step there will be various epitopes and antibodies unnecessarily floating in the liquid; hence we have to flush out all the floating molecules.

This peptide system accepts a string if and only if it is accepted by the Turing machine. In the procedure above, for the peptide sequences in T we need $3s_{\mathcal{M}} + 2 \cdot |\Sigma|$ antibodies; for the sequences in S, $|Q| \cdot |\Sigma|$; for I_1, $|\Sigma| + |Q| \cdot |\Sigma|$; for I_2 there is no need for new antibodies; and for the sequences in P, the exact number depends on the transition table of $\mathcal{M}$.

The simulation presented above requires an infinite number of antibodies which, however is recursively enumerable. We can consider antibodies as being encoded over a finite alphabet (at least in our formal model). For instance

the set A could be an infinite solid code over a finite alphabet Y with $|Y| \geq 2$; such codes exist as shown in [6].

To encode n symbols by a solid code the maximal code word length is in $\Theta(\log n)$ [8]. Thus we obtain the following corollary of Theorem 1.

Corollary 1. *Let $\mathcal{M} = (Q, \Sigma, \delta, q_0, F, \flat)$ be a Turing machine. There is a simulation of $\mathcal{M}$ by peptide computing with the following properties:*

1. *Only a finite alphabet is required,*
2. *A step is simulated in $\Theta(\log s_{\mathcal{M}})$ steps.*

4 Modelling Peptide Computations

In this section we give a rigorous definition of *peptide computing*. This will allow us to determine, precisely, the capabilities and limitations of this computing paradigm.

It is usually easy to invent a new model of computation. Moreover, showing its universality only requires the simulation of Turing machines. To show that the proposed model is no more powerful than Turing machines is often quite a bit harder. For some pitfalls see the insightful discussion of computing models in [3]. The explication of the notion of *computability* as provided, for instance, by Church's (or Turing's) Thesis, regardless of its precise wording, is, essentially, recursive as it explains intuitive computability by steps which are, themselves, assumed to be intuitively computable and a program, the execution of which, is intuitively computable. On the one hand, the random access machine, in which an unbounded number of memory cells holding integers of unbounded size is used, can be shown to be polynomially equivalent to a Turing machine – although it is not obvious at all that moving around unbounded information is intuitively computable. On the other hand, a very simple look-alike of a Turing machine decides problems which are undecidable when using a Turing machine. Indeed, it is a simple consequence of a result of [3] that any degree of non-computability can be achieved just by the topology of the memory structure, that is, the way by which addresses are calculated.

A model M of computation defines *M-computability*. By *computability* without reference to a specific model we mean *intuitive computability*. Thus *Church's Thesis* (or *Turing's Thesis*), in its simplest form, states that Turing-computability and computability are equivalent notions. In [2] some of the subtleties of Church's Thesis are pointed out, which, for our purposes, are not relevant, however. Moreover, there are various stronger versions of the thesis, which we do not need here either, but which are useful as axioms in other contexts. By [1], Turing computability implies peptide computability; hence, using Church's Thesis, computability implies peptide computability.

To study the converse implication a significantly more formal definition of peptide computing is needed. In the sequel we present a formal model. Whether this model is adequate for the bio-chemical realities, is a matter of further research.

Definition 1. *A* peptide computer *is a quintuple* $\mathcal{P} = (X, E, A, \alpha, \beta)$ *where* X *is a finite alphabet (to represent basic building units like molecules),* $E \subseteq X^+$ *is a language (to represent epitopes),* A *is a countable alphabet with* $A \cap X^* = \emptyset$ *(to represent antibodies),* $\alpha \subseteq E \times A$ *is a relation (such that* $a \in \alpha(e)$ *means that antibody* a *can be attached to epitope* e*),* $\beta : E \times A \to \mathbb{R}_+$ *is a mapping such that* $\beta(e, a) > 0$ *if and only if* $(e, a) \in \alpha$ *(denoting the affinity between epitope* e *and antibody* a*).*

Consider a word $w \in X^+$ and $d \in \partial_E(w)$. An *A-attachment* is a partial mapping $\tau : d \stackrel{\circ}{\to} A$. Suppose $w = x_0x_1 \cdots x_n$ and $d = \{(i_l, j_l) \mid l = 0, 1, \ldots, k-1\}$. Then τ defines a word $w_\tau \in \big(X \cup (E \times A)\big)^*$ as follows: For all $l = 0, 1, \ldots, k-1$, if $(i_l, j_l) \in dom\,\tau$ replace $e = x_{i_l}x_{i_l+1} \cdots x_{j_l}$ by $(e, \tau(i_l, j_l))$ in w. Such a mapping τ is *legal* if $(e, \tau(i_l, j_l)) \in \alpha$ for all l. When τ is legal then $w_\tau \in (X \cup \alpha)^*$ and τ is called an *A-attachment to* w. For a language $L \subseteq X^+$, let $\mathcal{T}(L)$ be the set of A-attachments to words in L. Conversely, a word $z \in (X \cup \alpha)^*$ defines a word $w \in X^*$ and a set of A-attachments τ, such that $w_\tau = z$. Note that w is uniquely defined, but that τ may apply to several $d \in \partial_E w$.

Consider a word $z \in (X \cup \alpha)^+$ and a symbol $a \in A$. Let w and τ be such that $w_\tau = z$. Moreover, let $w = x_0x_1 \cdots x_n$ with $x_0, x_1, \ldots, x_n \in X$. Consider any $d \in \partial_E w$ with $dom\,\tau \subseteq d$ and any $d' \in \partial_E w$. For $(i, j) \in d'$ let $e_{i,j} = x_ix_{i+1} \cdots x_j$. We say that a *dominates* (i, j) in z when the following condition is satisfied: For all $(i', j') \in d$ such that $\{i', i'+1, \ldots, j'\} \cap \{i, i+1, \ldots, j\} \neq \emptyset$ and $(i', j') \in dom\,\tau$,

$$\beta(e_{i,j}, a) > \beta(x_{i'}x_{i'+1} \cdots x_{j'}, \tau(i', j')).$$

In such a case, all pairs $(i', j') \in d$ with $\{i', i'+1, \ldots, j'\} \cap \{i, i+1, \ldots, j\} \neq \emptyset$ are said to be *affected.* If a dominates (i, j) in z, the following *basic reaction will happen* forming a multiset $R(z, a)$: For each affected pair (i', j'), a copy of $\tau(i'j')$ is put into $R(z, a)$; let $Y \subseteq dom\,\tau$ be the set of pairs which are not affected and let $d'' \in \partial_E w$ be such that $Y \cup (i, j) \subseteq d''$. Define the A-attachment $\bar{\tau} : d'' \stackrel{\circ}{\to} A$ by $\bar{\tau}(p) = \tau(p)$ for $p \in Y$ and $\bar{\tau}(i, j) = a$. Put a copy of $w_{\bar{\tau}}$ into $R(z, a)$. The multiset $R(z, a)$ is the *result of a basic reaction* between z and a. If a is binding with z and some symbols are released from z when $R(z, a)$ is formed then we denote the set of released symbols by $Out(z, a)$. If nothing is released when a binds then $Out(z, a)$ will be $\{\lambda\}$.

We also need to consider *basic reactions* between words $z, z' \in (x \cup \alpha)^+$, where z and z' need not be different. Again we want to define the resulting multiset $R(z, z')$. We use w, d and τ as above. Now $z' = w'_{\tau'}$ where $\tau' : d' \stackrel{\circ}{\to} A$ for some $d' \in \partial_E w'$. Consider $(i', j') \in dom\,\tau'$ and let $a = \tau'(i'j')$. Moreover, let $e'_{i',j'}$ be the infix of w' which starts at i' and ends at j'. Suppose a dominates (i, j) in z for some $(i, j) \in \bar{d} \in \partial_E w$ and $\beta(e_{i,j}, a) > \beta(e'_{i',j'}, a)$, then the reaction is as follows.

Since the basic reaction between two words z and z' are with respect to a, we represent the result of this reaction by $R_a(z, z')$. This reaction takes place in two steps: first the reaction $SepR_a(z, z')$ takes place. This reaction produces the multiset containing z, z'' and a, where z'' is defined as follows: let τ'' be the

restriction of τ' to $dom\,\tau' \setminus (i'j')$. and z'' is defined as $z'' = w'_{\tau''}$. Then the next step is the reaction resulting in $R(z,a)$. As in the previous reaction $Out(z_1,z_2)$ denotes the set of symbols released from z_1 when a binds with z_1. Note that when z and z' are the same occurrence of a word then $SepR_a(z,z')$ consists only of z'' and a.

The basic reactions resulting in $R(z,a)$ and $R_a(z,z')$ take place only when there is instability. Instability between z and a occurs when a dominates $(i,j) \in \partial_E w$ where $z = w_\tau$. Likewise instability between two words z and z' occurs when there is a symbol $a = \tau'(i',j')$ where $(i',j') \in dom\ \tau'$ and $\tau' : d' \overset{\circ}{\to} A$ for some $d' \in \partial_E(w')$.

We also note that one basic reaction can trigger a sequence of reactions; this might even lead to a cycle which in turn will not lead to any stable configuration.

In the sequel we refer to $R(z,a)$ (or $R_a(z_1,z_2)$) as the result of a basic reaction or as a multiset, whichever is appropriate to the context.

Definition 2. *Let $\mathcal{P}$ be a peptide computer. A* peptide configuration *is a finite multiset of words in $(X \cup \alpha)^+ \cup A$.*

We denote a peptide configuration as P. To a peptide configuration P, a basic reaction may apply when instability exists in the configuration, that is, there may be $z, z' \in (X \cup \alpha)^+$ or $a \in A$ which occur in P such that $R(z,a)$ differs from the multiset consisting of z and a or $R(z,z')$ differs from the multiset consisting of z and z'. In either case a basic reaction non-deterministically removes (z,a) or (z,z') from P and adds $R(z,a)$ or $R(z,z')$, respectively. Let $R(P)$ be the set of peptide configurations which result from P through one basic reaction. For $n \in \mathbb{N}_0$, let R^n be the n-fold iteration of R.

Definition 3. *A peptide configuration P is said to be* stable *if $R(P) = \{P\}$.*

If $R^n(P)$ consists of stable configurations only, for some n, define $R^*(P) = R^n(P)$ for this n. Otherwise, $R^*(P) = \emptyset$. Let Γ be the class of stable peptide configurations.

To define peptide computations, we also need the following objects:

Definition 4. *A* peptide instruction *has the form $+P$ or $-P$ where P is a peptide configuration.*

When P' is a peptide configuration and I is a peptide instruction then

$$I(P') = \begin{cases} P' \cup P, & \text{if } I = +P, \\ P' \setminus P, & \text{if } I = -P, \end{cases}$$

with union and difference taken as multiset operations.

Definition 5. *A* peptide program *is a pair $(\mathfrak{P}, \chi)$ where $\mathfrak{P}$ is a mapping from Γ^* into the set of peptide instructions and χ is a (halting) function $\chi : \Gamma \to \mathbb{B}$.*

Definition 6. *Let $\mathcal{P}$ be a peptide computing model and let $(\mathfrak{P}, \chi)$ be a peptide program for $\mathcal{P}$. A* peptide computation *is a word $c = c_0c_1\cdots c_t \in \Gamma^*$ with $c_0, c_1, \ldots, c_t \in \Gamma$ such that*

$$c_i \in R^*\big(\mathfrak{P}(c_0c_1\cdots c_{i-1})(c_{i-1})\big)$$

for $i = 0, 1, \ldots, c_t$.

A computation as above starts with $c_0 \in R^*(\mathfrak{P}(\lambda))$ and ends when $\chi(c_i) = 1$ for the first time.

To encode inputs we need a mapping γ from inputs to Γ, an input encoding; we also need an output decoding, that is, a mapping δ from Γ to outputs.

Definition 7. *A function f from inputs to outputs is* peptide computable *if there is a peptide program $\mathfrak{P}$, a computable input encoding γ of inputs into $\mathfrak{P}(\lambda)$ and a computable decoding of Γ into outputs such that, for every $x \in \operatorname{dom} f$, there is a peptide computation $c_0c_1\cdots c_t$ with $c_0, c_1, \ldots, c_t \in \Gamma$ and $\gamma(x) = c_0$ satisfying $\chi(c_t) = 1$ and $\delta(c_t) = f(x)$.*

5 How the New Simulation Fits the Peptide Model

We describe how our proposed simulation of Section 3 can be carried out by the peptide model presented in Section 4.

We define the sets X, E, A and describe the relations α and β as below:

1. $X = \{X_1, X_2, \cdots, X_{20}\}$;
2. $E = \{e^{(T)}_{i,1}, e^{(T)}_{i,2}, e^{(T)}_{i,3}, x_i e^{(T)}_{i,2} y_i, e^{(T)}_{i,2} y_i e^{(T)}_{i,3}, e^{(T)}_{i,1} x_i e^{(T)}_{i,2} y_i\} \cup \{e^{(S)}_{(q,a),1}, e^{(S)}_{(q,a),2}\} \cup \{e^{(I_1)}_{(q,a),1}, e^{(I_1)}_{(q,a),2}\} \cup \{e^{(I_2)}_{(q,a)}\} \cup \{e^{(P)}_{(q,a),1}, e^{(S)}_{(q,a),2}, e^{(P)}_{(q,a),3}\}$; and
3. $A = \{A_{a,\mathtt{D}}, A_i, B_{q,a}, A_{q,a}, B_{a,\mathtt{D}}, A_q\}$.
4. $\alpha = \left\{ \begin{array}{l} (A_a, e^{(T)}_{i,4}), (A_{a,\mathtt{L}}, e^{(T)}_{i,1} e^{(T)}_{i,4}), (A_{a,\mathtt{R}}, e^{(T)}_{i,4} e^{(T)}_{i,3}), (A_i, e^{(T)}_{i,2}), \\ (A_{i+1}, e^{(T)}_{i,3}), (A_{i-1}, e^{(T)}_{i,1}), (A_q, e^{(S)}_{(q,a),1}), (A_a, e^{(S)}_{(q,a),2}), \\ (A_{a,\mathtt{D}}, e^{(S)}_{(q,a),2}), (B_{q,a}, p^{(S)}_{(q,a)}), (A_q, e^{(I_1)}_{(q,a),1}), (A_{q,a}, e^{(I_1)}_{(q,a),1}), \\ (A_{a,\mathtt{D}}, e^{(I_1)}_{(q,a),2}), (B_{a,\mathtt{D}}, e^{(I_1)}_{(q,a),2}), (A_{q,a}, e^{(I_2)}_{(q,a)}), (A_{a,\mathtt{D}}, e^{(I_2)}_{(q,a)}), \\ (A_q, e^{(P)}_{(q,a),1}), (A_{a,\mathtt{D}}, e^{(P)}_{(q,a),2}), (A_{q,a}, e^{(P)}_{(q,a),3}), (A_i, \bar{e}^{(T)}_{i,2}) \end{array} \right\}$
5. To define β, we can assign a positive real number (non-zero) from $\mathbb{R}$ to each pair in α such that the affinity relation described in Section 3 is valid. Together with this we define β such that A_i dominates $\bar{e}^{(T)}_{i,2}$.
6. If c_k is some configuration then the halting function $\chi(c_k)$ is defined as 1 if a labelled symbol A_q is present in the configuration c_k with the sequence in S.

The sequences used for this simulation are the same ones as in Section 3. The only new sequences are $\bar{e}^{(T)}_{i,2}$. Hence we have five sets of sequences over X denoted by T, P, S, I_1 and I_2.

The number of symbols in A is infinite. The peptide sequences S and I_2 are used as they are. The other sequences are extended to sequences over $(X \cup \alpha)^*$.

First we take the sequences from P: these sequences are extended to sequences over $(X \cup \alpha)^*$ as follows:

$$p^{(P)}_{(q,a)} \;=\; (e^{(P)}_{(q,a),1}, A_{q'})(e^{(P)}_{(q,a),2}, A_{a',\mathtt{D}}).$$

These sequences are formed for every transition $\delta(q,a) = (q', a', \mathtt{D})$. Every sequence in T representing a cell of the Turing machine is formed as follows: for $p^{(T)}_i \in T$,

$$p^{(T)}_i \;= (e^{(T)}_{i,1}, A_{i-1})(x_i e^{(T)}_{i,2} y_i, A_a)(e^{(T)}_{i,3}, A_{i+1}).$$

The sequence $p^{(I_1)}_{(q,a)} \in I_1$ is extended as follows:

$$p^{(I_1)}_{(q,a)} \;=\; (e^{(I_1)}_{(q,a),1}, A_{q,b}) e^{(I_1)}_{(q,a),2}.$$

Hence we have sets of sequences T, P and I_1 over $(X \cup \alpha)^*$ and sets S and I_2 over X: these sequences are taken as the initial configuration. Processing starts by adding two symbols A_{q_0} and A_1 to the initial configuration. In general let A_q and A_i be the symbols added to the configuration; assume that A_a is present with the sequence $p^{(T)}_i$. Now two reactions take place resulting in $R(A_q, p^{(S)}_{(q,a)})$ and $R(A_i, p^{(T)}_i)$. The first reaction changes $p^{(S)}_{(q,a)}$ into,

$$p^{(S)}_{(q,a)} \;=\; (e^{(S)}_{(q,a),1}, A_q) e^{(S)}_{(q,a),2}.$$

The second reaction results in $Out(A_i, p^{(T)}_i) \;=\; \{A_a\}$. It changes $p^{(T)}_i$ into,

$$p^{(T)}_i \;=\; e^{(T)}_{i,1} x_i (e^{(T)}_{i,2}, A_i) y_i e^{(T)}_{i,3}.$$

After these reactions the free symbol A_a triggers a new reaction resulting in $R(A_a, p^{(S)}_{(q,a)})$, that is, the sequence

$$p^{(S)}_{(q,a)} \;=\; (e^{(S)}_{(q,a),1}, A_q)(e^{(S)}_{(q,a),2}, A_a).$$

Then symbols $B_{p,b}$ are added for all $p \in Q$ and $b \in \Sigma$; this stimulates reactions resulting in $R(p^{(S)}_{(p,b)}, B_{p,b})$. Hence the sequences $p^{(S)}_{(p,b)}$ become,

$$p^{(S)}_{(p,b)} \;=\; (z_{p,b} e^{(S)}_{(q,a),1} e^{(S)}_{(q,a),2}, B_{p,b})$$

with $Out(p^{(S)}_{(q,a)}, B_{q,a}) = \{A_q, A_a\}$; when $b \neq a$ or $p \neq q$, $Out(p^{(S)}_{(p,b)}, B_{p,b}) = \{\lambda\}$. The presence of symbols A_q and A_a paves the way for two more reactions leading to $R(p^{(I_1)}_{(q,a)}, A_q)$ and $R(p^{(I_1)}_{(q,a)}, A_a)$. Hence the sequence $p^{(I_1)}_{(q,a)}$ changes to,

$$p^{(I_1)}_{(q,a)} \;=\; (e^{(I_1)}_{(q,a),1}, A_q)(e^{(I_1)}_{(q,a),2}, A_a)$$

with $Out(p^{(I_1)}_{(q,a)}, A_q) = \{A_{q,b}\}$ for all $b \in \Sigma$. The symbols $A_{q,b}$ which are released give rise to reaction resulting in $R(p^{(I_2)}_{(q,a)}, A_{q,b})$ and the sequence $p^{(I_2)}_{(q,a)}$ becomes,

$$p^{(I_2)}_{(q,a)} \;=\; (e^{(I_2)}_{(q,a)}, A_{q,b}).$$

Then we add symbols B_b for all $b \in \Sigma$ which results in $R(p^{(I_1)}_{(q,a)} B_b)$, and

$$p^{(I_1)}_{(q,a)} \;=\; (e^{(I_1)}_{(q,a),1}, A_q)(e^{(I_1)}_{(q,a),2}, B_b)$$

with $Out(p^{(I_1)}_{(q,a)}, B_b) = \{A_a\}$. The presence of the symbol A_a leads to $R(p^{(I_2)}_{(q,a)}, A_a)$ and $p^{(I_2)}_{(q,a)}$ becomes

$$p^{(I_2)}_{(q,a)} \;=\; (e^{(I_2)}_{(q,a)}, A_a)$$

with $Out(p^{(I_2)}_{(q,a)}, A_a) \;=\; \{A_{q,a}\}$.

Hence two symbols denoting the state q and the input symbol a, through a sequence of reactions have been transformed into a single symbol $A_{q,a}$ which denotes both the state and the input symbol of the machine.

The symbol $A_{q,a}$ stimulates the reaction resulting in $R(p^{(P)}_{(q,a)}, A_{q,a})$ and this gives rise to the sequence

$$p^{(P)}_{(q,a)} \;=\; x(e^{(P)}_{(q,a),3}, A_{q,a})y$$

with $Out(p^{(P)}_{(q,a)}, A_{q,a}) = \{A_{\bar{q}}, A_{\bar{a},\bar{\mathtt{D}}}\}$ where $\bar{q}$ and $\bar{a}$ are such that $\delta(q,a) = (\bar{q}, \bar{a}, \bar{\mathtt{D}})$. Hence the peptide computer has reached the next state $\bar{q}$ from q and selected the symbol to be rewritten as $\bar{a}$. Now we add sequences $\bar{e}^{(T)}_{i,2}$ which leads to $R(p^{(T)}_i, \bar{e}^{(T)}_{i,2})$ (where A_i dominates $\bar{e}^{(T)}_{i,2}$) and results in the separation of A_i from $p^{(T)}_i$ and the sequence $\bar{e}^{(T)}_{i,2}$ becomes $(\bar{e}^{(T)}_{i,2}, A_i)$. This creates space for $A_{\bar{a},\bar{\mathtt{D}}}$ to bind to it by the reaction resulting in $R(p^{(T)}_i, A_{\bar{a},\bar{\mathtt{D}}})$. Hence the sequence $p^{(T)}_i$ becomes,

$$p^{(T)}_i \;=\; (e^{(T)}_{i,1}, A_{i-1})(x_i e^{(T)}_{i,2} y_i e^{(T)}_{i,3}, A_{\bar{a},\bar{\mathtt{D}}})$$

with $Out(p^{(T)}_i, A_{\bar{a},\bar{\mathtt{D}}}) \;=\; \{A_{i+1}\}$ when $\bar{\mathtt{D}} = \mathtt{R}$; when $\bar{\mathtt{D}} = \mathtt{L}$,

$$p^{(T)}_i \;=\; (e^{(T)}_{i,1} x_i e^{(T)}_{i,2} y_i, A_{\bar{a},\bar{\mathtt{D}}})(e^{(T)}_{i,3}, A_{i+1})$$

with $Out(p^{(T)}_i, A_{\bar{a},\bar{\mathtt{D}}}) \;=\; \{A_{i-1}\}$

Now the system is ready to carry out the next transition of the machine. The system continues as above until it reaches at a labelled symbol A_q in a sequence in S.

6 Simulation of Peptide System by Turing Machine

We present an informal construction of a Turing machine which simulates a peptide computer under certain conditions.

Theorem 2. *For every peptide computer $\mathcal{P} = (X, E, A, \alpha, \beta)$ with the following conditions:*

1. *E and A are (at least) computably enumerable;*
2. *α is decidable;*
3. *β and χ are computable;*

and for every computably enumerable peptide program $\mathfrak{P}$ for $\mathcal{P}$, there is a Turing machine simulating the peptide computations of $\mathcal{P}$ according to $\mathfrak{P}$.

Proof Idea. Just invoking Church's Thesis is insufficient for a proof, tempting as this may be. We need to prove that peptide computing under the restrictions as stated is not more powerful than intuitive computing. Let $\mathcal{P} = (X, E, A, \alpha, \beta)$ be the peptide system. We simulate $\mathcal{P}$ by a multi-tape Turing machine $\mathcal{M}$.

The instruction set is encoded on the first tape. The encoding distinguishes between a symbol $a \in A$ and a sequence $z \in (X \cup \alpha)^*$. In the sequence representation we have delimiters for the epitopes. Thus the machine can recognize the beginnings and ends of epitopes. Let $z = z_1(e, a)z_2$ be a sequence, where $z_1, z_2 \in (X \cup \alpha)^*$, $(e, a) \in \alpha$ and $\beta(e, a) = n$, encoded on the tape. Assume that $e = x_1 x_2 \cdots x_k$, then the encoding of e would be $(x_1, a, n)(x_2, a, n) \cdots (x_k, a, n)$.

On the second tape we encode, for each of $a \in A$, the set of all epitopes to which it can attach. This set of epitopes is ordered in decreasing order of the relation $\geq$. Since α is a poset relation it consists of finite number of chains (at any point of time there will be only a finite number of epitopes to which a symbol can attach). Each chain is arranged in decreasing order and the chains can be ordered arbitrarily. When encoding this on the tape each chain has markers for the start and the end.

The third tape stores the configuration of the system and the output of each reaction which might trigger more reactions. The fourth tape encodes the function χ.

Let C be the configuration stored on the third tape. Let $a \in A$ be the symbol which is now added to the configuration C. This amounts to a reaction resulting in $R(z, a)$ in $\mathcal{P}$. We explain the simulation of this reaction by $\mathcal{M}$.

First a is copied onto the third tape; then we look, on the second tape, for all epitopes to which a can attach; we non-deterministically select a chain and, for each epitope in that chain, we check the configuration on the third tape for an occurrence; if there is such an occurrence we check if there are symbols bound to that site already; the affinity is compared; if it is greater then the symbol presently attached is erased and a is written onto the tape together with its affinity as a triplet using the same representation as on the first tape. Then the system $\mathcal{P}$ can check if there is any possibility of further reactions as described further below. If there is no epitope found in the configuration or if there is no possibility of further reactions then the next instruction is chosen according to the program $\mathfrak{P}$.

Suppose we add z_2 to the configuration C (let z_1 be a sequence in C and $a \in A$ is attached to an epitope in z_1); hence in one of the possible cases $R_a(z_1, z_2)$ is obtained if a dominates any epitope in z_2. In the sequel we describe how this

reaction is simulated by $\mathcal{M}$. First the sequence is copied to the third tape. Then the symbol a is erased from the sequence z_1 and it is written in z_2 corresponding to its epitope. In general, if this reaction has to happen for each sequence z_2 put into C and for each symbol attaching with it we need to check with all the other sequences z_1 in C if it dominates any epitope. Likewise for each z_1 in C and for each symbol a attached to it we need to check if a dominates any epitope in z_2. If either of this is true then the corresponding reactions resulting in $R_a(z_1, z_2)$ or $R_a(z_1, z_2)$ take place. After these checks the system has to examine if there are any further reactions (1) by checking if there are any outputs from the reactions; and (2) by checking for other reactions.

After finishing each instruction we can check in the fourth tape if $\chi(C) = 1$. If so then the machine halts.

7 Conclusion

We presented a new model for simulating a Turing machine using peptide-antibody interactions. In this model we encoded the transition functions of Turing machine in the sequences themselves so that the simulation automatically selects the next transition. Moreover, we established conditions on a peptide computer to be simulated by a Turing machine.

References

1. M. S. Balan, K. Krithivasan, Y. Sivasubramanyam: Peptide computing: Universality and computing. In N. Jonoska, N. Seeman (editors): *Proceedings of Seventh International Conference on DNA based Computers, LNCS 2340.* 290–299, 2002.
2. A. M. Ben-Amram: The Church-Turing thesis and its look-alikes. *Sigact News* **36**(3) (2005), 113–114.
3. S. A. Cook, S. O. Aanderaa: On the minimum computation time of functions. *Trans. Amer. Math. Soc.* **142** (1969), 291–314.
4. J. E. Hopcroft, J. D. Ullman: *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.
5. H. Hug, R. Schuler: Strategies for the developement of a peptide computer. *Bioinformatics* **17** (2001), 364–368.
6. H. Jürgensen, M. Katsura, S. Konstantinidis: Maximal solid codes. *Journal of Automata, Languages and Combinatorics* **6** (2001), 25–50.
7. H. Jürgensen, S. Konstantinidis: Codes. In G. Rozenberg, A. Salomaa (editors): *Handbook of Formal Languages*, 1. 511–607. Springer-Verlag, Berlin, 1997.
8. H. Jürgensen, S. Konstantinidis, N. H. Lâm: Asymptotically optimal low-cost solid codes. *J. of Automata, Languages and Combinatorics* **9** (2004), 81–102.

Handling Markov Chains with Membrane Computing

Mónica Cardona[1], M. Angels Colomer[1],
Mario J. Pérez-Jiménez[2], and Alba Zaragoza[1]

[1] Department of Mathematics, University of Lleida
Av. Alcalde Rovira Roure, 191. 25198 LLeida, Spain
{colomer, alba, mcardona}@matematica.udl.es
[2] Research Group on Natural Computing
Department of Computer Science and Artificial Intelligence
University of Sevilla
Avda. Reina Mercedes s/n, 41012 Sevilla, Spain
marper@us.es

Abstract. In this paper we approach the problem of computing the n–th power of the transition matrix of an arbitrary Markov chain through membrane computing. The proposed solution is described in a semi–uniform way in the framework of P systems with external output. The amount of resources required in the construction is polynomial in the number of states of the Markov chain and in the power. The time of execution is linear in the power and is independent of the number of states involved in the Markov chain.

1 Introduction

In the field of the Natural Computing, two areas that have attracted a great interest are the *molecular computing based on DNA* and, more recently, the *cellular computing with membranes*. One of the advantages of these models with respect to the classic ones is the massive parallelism that in these models is implemented in a natural way and allowing the simultaneous execution of many operations in an unit of the time.

The molecular computing provides a model of computation oriented to program and so, the computing devices proposed follow a structure similar to the classic algorithms, in which the operations realized in each step depend on the result obtained in the previous step. However, the cellular computing with membranes provides a model of computation oriented to machines. In this model, the computing devices, similar to a Turing machine, start from an initial configuration (a structure of membranes with certain chemical compounds in its compartments) which evolves by means of rules of the system (abstraction of the different chemical reactions which are allowed in membranes). The rules are applied according to a specific semantic, that is to say, the execution of such devices modifies the content of their components until arriving in a halting state in which the machine does not evolve any longer.

C.S. Calude et al. (Eds.): UC 2006, LNCS 4135, pp. 72–85, 2006.

The calculation of the natural powers of the transition matrix of a finite and homogenous Markov chain is important, because it allows us to estimate its limit in the case that it is convergent and so, we can know the stationary distribution of the process. This subject have been treated in [1], where two algorithms based on DNA are described that only allow us to obtain an *estimation* of the powers. These algorithms run in polynomial time and require a polynomial amount of resources.

In this work this problem is approached within the framework of cellular computing with membranes, and an exact solution is provided in time which is linear in the order of the power and is independent of the number of states of the Markov chain. The amount of used resources is polynomial in the power and the number of states.

The paper is structured as follows. In the next section, basic concepts concerning Markov chains and P systems that are necessary for the development of the work are introduced. In Section 3, a P system with external output solving (in a semi–uniform way) the problem to find the n–th power of the transition matrix, and a formal verification of the system is presented; the run time and the resources required in the description of the system are analyzed.

2 Preliminaries

2.1 Markov Chains

Roughly speaking, a (discrete–time) Markov chain is a discrete–time stochastic process such that the past is irrelevant for predicting the future given knowledge of the present, i.e., the conditional distribution of what happens in the future given everything up to now depends only on the present state, and not otherwise on the past.

More formally, a *finite Markov chain* is a sequence $\{X_t : t \in \mathbf{N}\}$ of random variables verifying the following (Markov) property:

$$P(X_{t+1} = j/X_0 = i_0, X_1 = i_1, \ldots, X_t = i_t) = P(X_{t+1} = j/X_t = i_t).$$

That is, given the present, the future does not depend of the past: the result of each event only depends on the result of the previous event.

The range of the random variables is called the *state space* of the Markov chain, and the value of X_t is interpreted as the state of the process at time t. We suppose that the state space is finite, that is, the random variables only take the discrete values $e_1, \ldots, e_k$, called *states* or *results*.

Hence, a Markov chain $\{X_t : t \in \mathbf{N}\}$ provides a random process by a change of states or results $e_1, \ldots, e_k$ in certain instants of discrete times $t \in \mathbf{N}$, and where the result of each event only depends on the result of the previous event. So, such a Markov chain is characterized by the conditional distribution

$$p_{ij}(t) = P(X_t = e_j/X_{t-1} = e_i), \text{ for all } t \geq 1,$$

which is called the *transition probability* of the process, providing one–step transition probability.

The matrix $P(t) = (p_{ij}(t))_{1\leq i,j\leq k}$ is called the *transition matrix* associated with the Markov chain $\{X_t : t \in \mathbf{N}\}$. The term (i, j) of the transition matrix is the probability of a transition from the state e_i to the state e_j. For that, every element of the transition probability matrix is positive, and the sum of each row is 1 because for all i $(1 \leq i \leq k)$ we have

$$\sum_{j=1}^{k} p_{ij}(t) = \sum_{j=1}^{k} P(X_t = e_j / X_{t-1} = e_i) = 1.$$

Hence, the matrix of transition probabilities associated with a Markov chain is stochastic. Moreover, every stochastic matrix can be viewed as the matrix of transition probabilities of some Markov chain.

We say that a finite Markov chain $\{X_t : t \in \mathbf{N}\}$ with k states is *stationary* or *homogeneous* if the transition probabilities do not depend on time, that is, $\forall t\ \forall i, j\ (1 \leq i, j \leq k \rightarrow p_{ij}(t) = p_{ij}(t + 1))$. In this case, we denote $p_{ij}(t) = p_{ij}$, for all $t \in \mathbf{N}$, and $P = (p_{ij})_{1\leq i,j\leq k} = P(t) = (p_{ij}(t))_{1\leq i,j\leq k}$.

The probability of a transition in two, three or more steps is derived in a natural way from the one–step transition probability and the Markov property. From the law of total probability, for all $t \geq 2$ we have:

$$p_{ij}^{(2)}(t) = \sum_{r=1}^{k} P(X_t = e_j / X_{t-1} = e_r) \cdot P(X_{t-1} = e_r / X_{t-2} = e_i).$$

That is, $p_{ij}^{(2)}(t) = \sum_{r=1}^{k} p_{rj}^{(1)}(t) \cdot p_{ir}^{(1)}(t-1)$, where $p_{ij}^{(1)}(t) = p_{ij}(t)$, for all $t \geq 1$.

Then, if the Markov chain is homogeneous, the transition matrix for the two–steps transition is:

$$(p_{ij}^{(2)})_{1\leq i,j\leq k} = (\sum_{r=1}^{k} p_{rj}^{(1)} \cdot p_{ir}^{(1)})_{1\leq i,j\leq k} = P \cdot P = P^2.$$

In general, for each $n \geq 2$ we have

$$p_{ij}^{(n)}(t) = \sum_{r=1}^{k} p_{rj}^{(1)}(t) \cdot p_{ir}^{(n-1)}(t-1)$$

If the Markov chain is homogeneous, then the transition matrix for the n–steps transition is:

$$(p_{ij}^{(n)})_{1\leq i,j\leq k} = (\sum_{r=1}^{k} p_{rj}^{(1)} \cdot p_{ir}^{(n-1)})_{1\leq i,j\leq k} = P^{n-1} \cdot P = P^n.$$

The conditions

$$\begin{cases} p_{ij}^{(1)} = p_{ij} \\ p_{ij}^{(n)} = \sum_{r=1}^{k} p_{rj}^{(1)} \cdot p_{ir}^{(n-1)}, \text{ for all } n \geq 2 \end{cases}$$

are called the *Kolmogorov–Chapmann equations* associated with the homogeneous Markov chain, whose transition matrix is $(p_{ij})_{1 \leq i,j \leq k}$.

A finite and homogeneous Markov chain $\{X_t : t \in \mathbf{N}\}$ where the set of states is $\{e_1, \ldots, e_k\}$, is characterized by the initial probabilities $q_0^j = P(X_0 = e_j)$ $(1 \leq j \leq k)$ to get the state e_j in the first event, and the transition probability matrix $P = (p_{ij})_{1 \leq i,j \leq k}$.

We denote the initial probabilities by means of the vector $q_0 = (q_0^1, \ldots, q_0^k)$, and for each $n \geq 1$, we consider the vector $q_n = (q_n^1, \ldots, q_n^k)$, where q_n^j $(1 \leq j \leq k)$ the probability to reach the state e_j after n–steps of the random process.

Notice that we have $q_n = q_0 \cdot P^n$, for each $n \geq 1$. So, in order to determine the distribution q_n it is enough to study the matrix P^n. Moreover, the limit of the sequence $\{P^n : n \in \mathbf{N}\}$ of these matrices allows us to obtain the distribution limit in the case it exits. For more details see [2] and [3].

Markov chains have many applications. For example, they are used in chemical engineering (modelling the probabilities in chemical reactions and in flow systems), in biology (to model processes that are analogous to biological populations), in bioinformatics (for coding region/gene prediction), in physics (for simulation of particle systems and spatial statistics), in telecommunications (using Markov models for queues), and in geostatistics (in two or three dimensional stochastic simulations of discrete variables conditional on observed data).

2.2 Membrane Systems

Membrane computing is an emergent branch of Natural Computing initiated in the fall of 1998 by Gh. Paun by a paper circulated at that time on web and published in 2000 [4]. Since then, the area has simply flourished and it has received important attention from the scientific community. In February 2003, the Institute for Scientific Information, ISI, has mentioned the foundational paper [4] as a *fast breaking paper* in Computer Science, and in October 2003, the domain itself was qualified by ISI as *fast emerging research front in computer science*.

This new model of computation has been introduced with the aim of defining a computing device, called P system, which abstracts from the structure and functioning of living cells, as well as from the organization of cell in tissues, organs, and other higher order structures.

The main *syntactic* ingredients of a P system are the following:

– A cell–like *membrane structure* consisting of several membranes arranged hierarchically inside a main membrane, the *skin*, and delimiting compartments or *regions*.

- *Multisets* of symbol–objects corresponding to chemical substances present in the compartments of a cell.
- *Evolution rules* corresponding to chemical reactions that can take place inside the cell, and that permit evolve the objects in a synchronous maximally parallel manner.

The *semantics* of P systems is defined through a non–deterministic and synchronous model (a global clock is assumed). A *configuration* of a P system consists of a membrane structure and a family of multisets of objects associated with each region of the structure. At the beginning, there is a configuration called the *initial configuration* of the system. We get *transitions* from one configuration of the system to the next one by applying the evolution rules to the objects placed inside the regions, in a non–deterministic, maximally parallel manner (in each region all objects that can evolve must do it). A *computation* of the system is a (finite or infinite) sequence of configurations such that each configuration –except the initial one– is obtained from the previous one by a transition. A computation which reaches a configuration where no more rules can be applied to the existing objects and membranes, is called a *halting computation*. The result of a halting computation is usually defined through the multiset associated with a specific output membrane (or the environment, as it is the case in this paper) in the final configuration.

More formally, a P *system with external output* of degree m is a tuple $\Pi = (\Gamma, \mu, \mathcal{M}_1, \cdots \mathcal{M}_m, (R_1, \rho_1), \ldots, (R_m, \rho_m))$ where:

- Π is an alphabet. Its elements are called objects.
- μ is a membrane structure consisting of m membranes, with the membranes injectively labeled with $1, 2, \ldots, m$.
- $\mathcal{M}_i, 1 \leq i \leq m$, are strings which represent multisets over Γ associated with the regions $1, \ldots, m$ of μ .
- $R_i, 1 \leq i \leq m$, are finite sets of evolution rules over Γ and ρ_i, $1 \leq i \leq m$, are partial orders over R. The sets R_i and ρ_i are associated with the region i of μ. An evolution rule is a pair (u, v), which usually write in the form $u \rightarrow v$, where u is a string over Γ and v is a string over $\Gamma \times (\{here, out\} \cup \{in_j : 1 \leq j \leq m\})$.

In this way, a comprehensive and systematic interdisciplinary research area was developed, a high generality and versatility, where models can be devised for a large range of processes where compartmentalization and multiset processing are natural ingredients. Thus, although the initial goal of membrane computing was only to learn new ideas, tools, and techniques from cell biology to the help of standard computers, much in the same way as, e.g., evolutionary computing suggests algorithms to be implemented on electronic computer, membrane computing became a new framework for building models for a large variety of processes, especially from biology (cell biology, tissues, populations of bacteria, controlling networks of complex phenomena, tumor growth, etc.), but also from linguistics, management, with several applications to computer science (computer graphics,

approximative solutions to computationally hard problems, modelling parallel architectures, cryptography).

Most of these models were proven to be computationally universal, able to compute whatever a Turing machine can compute. In the case when an enhanced parallelism is available, by means of membrane division, string-object replication, or membrane creation, polynomial (often linear) time solutions to **NP**-complete problems were found.

In many variants, P systems are seen as devices of a *generative* nature, that is, from a given initial configuration several distinct computations may be developed, in a non–deterministic manner, producing different outputs.

In this paper we work with P systems with external output and that perform *computing* tasks. For example, if a certain natural number, n, is encoded by the multiplicity of a special object in the initial configuration and we consider the cardinality of the multiset contained in the environment of a halting configuration as the result of a successful computation, then we can say that the system *computes* a partial function from natural numbers onto the set of natural numbers.

In the following, we assume that the reader is familiar with the basic notions of P systems, and we refer to [5] for details.

3 Computing the Natural Power of a Markov Chain

The calculation of the natural powers of the transition matrix of a finite and homogenous Markov chain allows us to estimate its limit in the case that it is convergent and so, we can know the stationary distribution of the process.

In this section, the natural powers of such Markov chains within the framework of the cellular computing with membranes are computed. The solution provided is linear in the order of the power and is independent of the number of states of the Markov chain. The amount of used resources is polynomial in the power and the number of states. Also, a formal verification of the solution is presented.

3.1 Designing a P System

For each Markov chain and each natural number, $n \geq 2$, we construct a P system with external output computing the n–th power of the matrix P_k associated with the Markov chain. Therefore, we provide a *semi–uniform* solution to this problem, that is, we give a family $\mathbf{\Pi} = \{\Pi(P_k, n) : \ n \in \mathbf{N}\}$ such that:

- There exists a deterministic Turing machine working in polynomial time which constructs the system $\Pi(P_k, n)$ from $n \in \mathbf{N}$.
- The output of the P system $\Pi(P_k, n)$ encodes the n–th power of the matrix P_k.

Let $P_k = (p_{ij})_{1 \leq i,j \leq k}$ be the matrix of the transition probabilities associated with a finite and homogeneous Markov chain of order k. Having in mind that

p_{ij} are real numbers in $[0, 1]$ and P systems only work with natural numbers, we have to prefix the approximation to be used in order to represent those numbers in our system. In this paper, as an example, we will work with an approximation to one decimal digit, reason why several objects appear with a factor of 10 in their multiplicities in the description of our system (similarly, if we want to work with m decimal digits, then we must use a 10^m factor).

Let $n \geq 2$ be a natural number. We define a P system of degree 3 with external output,

$$\Pi(P_k, n) = (\Gamma(P_k, n), \mu(P_k, n), \mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3, R)$$

associated with the matrix P_k and the natural number n, computing the n–th power of P_k, as follows:

– Working alphabet:

$$\begin{aligned}\Gamma(P_k, n) = &\{s_{ij}^{(r)} : \ 1 \leq i, j \leq k, \ 0 \leq r \leq n\} \ \cup \{s_{ij} : \ 1 \leq i, j \leq k\} \cup \\ &\{t_{ij} : \ 1 \leq i, j \leq k\} \ \cup \ \{t_{iju}^{(r)} : \ 1 \leq i, j, u \leq k, \ 0 \leq r \leq n-1\} \cup \\ &\{p_{ij} : \ 1 \leq i, j \leq k\} \ \cup \ \{c_i : \ 1 \leq i \leq n-1\}.\end{aligned}$$

– Membrane structure: $\mu(P_k, n) = [_1 \ [_2 \ [_3 \ ]_3 \ ]_2 \ ]_1$.

– Initial multisets:

$$\mathcal{M}_1 = \mathcal{M}_2 = \emptyset;$$

$$\mathcal{M}_3 = \{t_{ij}^{k \cdot 10 \cdot p_{ij}} : \ 1 \leq i, j \leq k\} \ \cup \ \{s_{ii}^{(0)10} : \ 1 \leq i \leq k\} \ \cup \ \{c_1\}.$$

– The set R of evolution rules consists of the following rules:

- Rules in the skin membrane labeled by 1:

$$\{t_{iju}^{(r)10p_{ij}} \longrightarrow (t_{ij}^{10p_{ij}} s_{uj}^{(r+1)10p_{ij}}, in_2) : \ 1 \leq i, j, u \leq k, \ 0 \leq r \leq n-1\} \ \cup$$
$$\{s_{ij} \longrightarrow (s_{ij}, out) : \ 1 \leq i, j \leq k\}.$$

- Rules in the membrane labeled by 2:

$$\{s_{ij}^{(r)10} t_{j1}^{10p_{j1}} \ldots t_{jk}^{10p_{jk}} \longrightarrow (t_{j1i}^{(r)10p_{j1}} \ldots t_{jki}^{(r)10p_{jk}}, out) : \ 1 \leq i, j \leq k, \ 0 \leq r \leq n-1\} \ \cup \ \{s_{ij}^{(n)} \longrightarrow (s_{ij}, out) : \ 1 \leq i, j \leq k\}.$$

- Rules in the membrane labeled by 3:

$$\{s_{ii}^{(0)} \longrightarrow s_{ii}^{(0)10} : \ 1 \leq i \leq k\} \ \cup \ \{t_{ij} \longrightarrow t_{ij}^{10} : \ 1 \leq i, j \leq k\} \ \cup \ \{c_i \longrightarrow c_{i+1} : \ 1 \leq i \leq n-2\} \ \cup \ \{c_{n-1} \longrightarrow \delta\}.$$

3.2 An Overview of Computations

The P system $\Pi(P_k, n)$ works in the following way. At the beginning, the skin membrane and the membrane labeled by 2 are empty and the membrane labeled by 3 has: (a) objects t_{ij} $(1 \leq i, j \leq k)$ encoding the elements p_{ij} of the trasition matrix of the Markov chain; (b) objects s_{ii} $(1 \leq i \leq k)$ encoding the states e_i of the chain; and (c) objects c_i $(1 \leq i \leq n-1)$ interpreted as counters used to know when a suitable number of objects t_{ij} and s_{ii} have been produced.

In the $n-2$ first steps only rules in the internal membrane labeled by 3 are applied. During this (so called) first stage, each object $s_{ii}^{(0)}$ and each t_{ij} is replicated 10 times in each transition step. Furthermore, in membrane 3 a counter c_i appears, initialized to c_1, whose subindex increase by one unit during each step. For that reason, after those $n-2$ steps, membrane 3 contains the multiset of objects $s_{ii}^{(0)\cdot 10^{n-1}}\ t_{ij}^{k\cdot 10^{n-1}\cdot p_{ij}}$, and the counter c_{n-1}. In the $(n-1)$–th step, each object $s_{ii}^{(0)}$ and each object t_{ij} are replicated 10 times and, moreover, membrane 3 is dissolved, and its content pass to the internal membrane labeled by 2.

Therefore, when the system is going to execute the n–th step, the skin membrane continues being empty and the content of the internal membrane labeled by 2 is $s_{ii}^{(0)\cdot 10^{n}}\ t_{ij}^{k\cdot 10^{n}\cdot p_{ij}}$.

In a second stage, in each $(n-1+2m+1)$–th step, with $m \in \mathbf{N}$, only rules of membrane 2 will be applied; they will consume all the objects $s_{ij}^{(m)}$ and some objects t_{ij}, sending to the skin certain objects t_{jui}. In each $(n-1+2m)$–th step, with $m \in \mathbf{N}-\{0\}$, only rules in the skin are applied (because there do not exist objects s_{ij} in membrane 2), sending new objects t_{ij} and objects $s_{ij}^{(m)}$ to membrane 2. This second phase finalizes when $m = n-1$.

A third stage begins with the execution of the $(3n-1)$–th step, after which the skin membrane will be empty and in membrane 2 objects $s_{ij}^{(n)}$ appear. Then, the execution of the rules $s_{ij}^{(n)} \longrightarrow (s_{ij}, out)$ sends these objects to the skin membrane, and in the following step, they sent to the environment by means of the rules skin $s_{ij} \longrightarrow (s_{ij}, out)$. In this moment, no rule of the system will be applicable and so, the configuration obtained after the $(3n+1)$–th step will be a halting one. Moreover, in the last step, the content of the environment will be $s_{ij}^{w_{ij}^{(n)}}$.

Finally, it will remain to show that the multiplicity $w_{ij}^{(n)}$ of the object s_{ij} is equal to the (i,j)–term of the matrix $10^n \cdot P_k^n$.

3.3 Formal Verification

Throughout this section we are going to justify that the system $\Pi(P_k, n)$ is deterministic and that the computation of the system codifies in the environment of the halting configuration the n–th power of the transition probability matrix, P_k, associated with a finite and homogeneous Markov chain.

First of all, let us list the necessary resources to build the system $\Pi(P_k, n)$ from the matrix P_k and the natural number $n \geq 2$:

- Size of the alphabet: $(n+1)k^2 + k^2 + k^2 + nk^3 + k^2 + (n-1) \in \Theta(nk^3)$.
- Sum of the sizes of initial multisets: $\leq 10k^3 + 10k + 1 \in \Theta(k^3)$.
- Maximum of rules' lengths: $20k + 10 \in \Theta(k)$.
- Number of rules: $nk^3 + k^2 + nk^2 + k^2 + k + k^2 + (n-1) \in \Theta(nk^3)$.

Bearing in mind the recursive description of the rules and that the amount of resources is polynomial in $n \cdot k$, it is possible to construct the system $\Pi(P_k, n)$ from the matrix P_k and the natural number $n \geq 2$, by means of a deterministic Turing machine working in polynomial time.

Next, we are going to define in a recursive manner in r the expression $w_{ij}^{(r)}$, for each i, j such that $1 \leq i, j \leq k$, that it is necessary in the formal verification of the system that will follow.

Definition 1. *Let $w_0 = 10^n$. For each i, j such that $1 \leq i, j \leq k$, we define:*

$$w_{ij}^{(0)} = \begin{cases} w_0 & \text{if } i = j, \\ 0 & \text{if } i \neq j. \end{cases}$$

$$w_{ij}^{(r+1)} = \sum_{u=1}^{k} w_{iu}^{(r)} p_{uj}, \text{ for } r < n.$$

Remark: In the case of a finite and homogeneous Markov chain, with states $e_1, \ldots, e_k$ and transition matrix P_k, the definition of the values w_{ij}^{r+1} can be interpreted as an *abstraction* of the equation of Kolmogorov–Chapmann: the transition of the state e_i to the state e_j in $(r+1)$ steps (that is, the value $w_{ij}^{(r+1)}$) is obtained from all the transitions in r steps of the state e_i to any state e_u (that is, the value $w_{iu}^{(r)}$), with $1 \leq u \leq k$, multiplied by the transitions of e_u to e_j in only one step (that is, the value p_{uj}).

Next, we establish the relation that exists between the elements $w_{ij}^{(r)}$ and the term (i, j) of the matrix $10^n \cdot P_k^r$, for each $1 \leq r \leq n$.

Proposition 1. *Let $n \geq 2$. Let us denote $B(r, n) = 10^n \cdot P_k^r$, for each r such that $r \geq 1, r \leq n$. If $B(r, n) = (b_{ij}^{(r,n)})_{1 \leq i,j \leq k}$, then*

$\forall r \geq 1 \ (r \leq n \longrightarrow \forall i, j \ (1 \leq i, j \leq k \longrightarrow b_{ij}^{(r,n)} = w_{ij}^{(r)}))$.

Proof. By induction on r.
The base case, $r = 1$, follows from the following remark:

$$w_{ij}^{(1)} = \sum_{u=1}^{k} w_{iu}^{(0)} p_{sj} = w_0 p_{ij} = 10^n p_{ij} = b_{ij}^{(1,n)}.$$

Let $r \geq 1$ be such that $r < n$ and suppose that $b_{ij}^{(r,n)} = w_{ij}^{(r)}$ is verified. Bearing in mind that $10^n \cdot P_k^{r+1} = 10^n P_k^r \cdot P_k = B(r, n) \cdot P_k$, we deduce that for each i, j such that $1 \leq i, j \leq k$:

$$b_{ij}^{(r+1,n)} = \sum_{u=1}^{k} b_{iu}^{(r,n)} p_{uj} \overset{h.i.}{=} \sum_{u=1}^{k} w_{iu}^{(r)} p_{uj} = w_{ij}^{(r+1)}. \qquad \square$$

For each $m \in \mathbf{N}$, we denote by $\mathcal{C}_m$ the configuration of the system obtained after the execution of m steps. For each label $l \in \{1, 2, 3\}$, we denote by $C_m(l)$ the multiset of objects contained in the membrane labeled by l in the configuration

$\mathcal{C}_m$. Also, we denote by $C_m(env)$ the content of the environment of the system in the configuration $\mathcal{C}_m$.

From the definition of the system $\Pi(P_k, n)$, the following holds: $C_0(1) = C_0(2) = \emptyset$, and $C_0(3) = \{t_{ij}^{k\cdot 10\cdot p_{ij}} : 1 \leq i, j \leq k\} \cup \{s_{ii}^{(0)10} : 1 \leq i \leq k\} \cup \{c_1\}$.

Next, we are going to determine the content of the different membranes of the system along the execution, and will show that after $3n+1$ steps the system reaches a halting configuration, in which the content of the environment is the multiset of objects $\{s_{ij}^{w_{ij}^{(n)}} : 1 \leq i, j \leq k\}$.

In the proof of the following result, it can be checked that there exists only one multiset of rules applicable to a non halting configuration in each transition step, and, consequently, the membrane system $\Pi(P_k, n)$ is deterministic.

Theorem 1. *Let $n \geq 2$.*

(a) For each $r \in \mathbf{N}$ such that $1 \leq r \leq n-2$ we have:

$$\begin{cases} C_r(1) = \emptyset \\ C_r(2) = \emptyset \\ C_r(3) = \{t_{ij}^{k\cdot 10^{r+1}\cdot p_{ij}} : 1 \leq i, j \leq k\} \cup \{s_{ii}^{(0)10^{r+1}} : 1 \leq i \leq k\} \cup \{c_{r+1}\} \end{cases}$$

(b) Moreover, we have:

$$\begin{cases} C_{n-1}(1) = \emptyset, \\ C_{n-1}(2) = \{t_{ij}^{k\cdot 10^{n}\cdot p_{ij}} : 1 \leq i, j \leq k\} \cup \{s_{ii}^{(0)10^{n}} : 1 \leq i \leq k\} \end{cases}$$

(c) For each $m \in \mathbf{N}$ such that $m < n$ we have:

$$\begin{cases} C_{(n-1)+2m}(1) & = \emptyset, \\ C_{(n-1)+2m}(2) & = \{t_{ij}^{kw_0p_{ij}} : 1 \leq i, j \leq k\} \cup \{s_{ij}^{(m)w_{ij}^{(m)}} : 1 \leq i, j \leq k\}, \\ C_{(n-1)+2m+1}(1) & = \{t_{jui}^{(m)w_{ij}^{(m)}p_{ju}} : 1 \leq i, j, u \leq k\}, \\ C_{(n-1)+2m+1}(2) & = \{t_{ij}^{kw_0p_{ij}-\sum_{u=1}^{k} w_{ui}^{(m)}p_{ij}} : 1 \leq i, j \leq k\}. \end{cases}$$

(d) The configuration $\mathcal{C}_{3n+1}$ is a halting one, and

$$C_{3n+1}(env) = \{s_{ij}^{w_{ij}^{(n)}} : 1 \leq i, j \leq k\}.$$

Proof.

(a) If $n = 2$, then the result is obvious. Let us assume now that $n > 2$. We will prove the result by induction on r.

In order to prove the base case $r = 1$, let us observe that from the initial configuration of the system we have: $C_0(1) = C_0(2) = \emptyset$, and $C_0(3) = \{t_{ij}^{k\cdot 10\cdot p_{ij}} : 1 \leq i, j \leq k\} \cup \{s_{ii}^{(0)10} : 1 \leq i \leq k\} \cup \{c_1\}$.

Then, only in the membrane 3 of configuration $\mathcal{C}_0$ there are applicable rules and, concretely, the rules: $s_{ii}^{(0)} \longrightarrow s_{ii}^{(0)10}$, for $1 \leq i \leq k$; $t_{ij} \longrightarrow t_{ij}^{10}$, for $1 \leq i, j \leq k$, and $c_1 \longrightarrow c_2$.

Therefore, $C_1(1) = C_1(2) = \emptyset$, and $C_1(3) = \{t_{ij}^{k\cdot 10^2\cdot p_{ij}} : 1 \leq i, j \leq k\} \cup \{s_{ii}^{(0)10^2} : 1 \leq i \leq k\} \cup \{c_2\}$.

Let r be a natural number such that $1 \leq r < n-2$. Let us suppose that $C_r(1) = C_r(2) = \emptyset$, and $C_r(3) = \{t_{ij}^{k \cdot 10^{r+1} \cdot p_{ij}} : 1 \leq i,j \leq k\} \cup \{s_{ii}^{(0)10^{r+1}} : 1 \leq i \leq k\} \cup \{c_{r+1}\}$.

Let us note that only in membrane 3 there are applicable rules to configuration $\mathcal{C}_r$; specifically, this is the case for the rules: $s_{ii}^{(0)} \longrightarrow s_{ii}^{(0)10}$, for $1 \leq i \leq k$; $t_{ij} \longrightarrow t_{ij}^{10}$, for $1 \leq i,j \leq k$; and $c_{r+1} \longrightarrow c_{r+2}$ (recall that $r < n-2$ and then $r+1 < n-1$).

Therefore, $C_{r+1}(1) = C_{r+1}(2) = \emptyset$, and $C_{r+1}(3) = \{t_{ij}^{k \cdot 10^{r+2} \cdot p_{ij}} : 1 \leq i,j \leq k\} \cup \{s_{ii}^{(0)10^{r+2}} : 1 \leq i \leq k\} \cup \{c_{r+2}\}$.

(b) Directly from (a) it follows that: $C_{n-2}(1) = C_{n-2}(2) = \emptyset$, and $C_{n-2}(3) = \{t_{ij}^{k \cdot 10^{n-1} \cdot p_{ij}} : 1 \leq i,j \leq k\} \cup \{s_{ii}^{(0)10^{n-1}} : 1 \leq i \leq k\} \cup \{c_{n-1}\}$.

In order to obtain the configuration $\mathcal{C}_{n-1}$, let us note that only in membrane 3 there are applicable rules, namely the rules: $s_{ii}^{(0)} \longrightarrow s_{ii}^{(0)10}$, for $1 \leq i \leq k$; $t_{ij} \longrightarrow t_{ij}^{10}$, for $1 \leq i,j \leq k$; and $c_{n-1} \longrightarrow \delta$.

Then, membrane 3 will be dissolved, its content goes to membrane 2, and the counters c_i disappear. Thus, $C_{n-1}(1) = \emptyset$, and $C_{n-1}(2) = \{t_{ij}^{k \cdot 10^n \cdot p_{ij}} : 1 \leq i,j \leq k\} \cup \{s_{ii}^{(0)10^n} : 1 \leq i \leq k\}$.

(c) We prove the result by induction on m. From (b) we deduce that $C_{n-1}(1) = \emptyset$, and $C_{n-1}(2) = \{t_{ij}^{kw_0p_{ij}} : 1 \leq i,j \leq k\} \cup \{s_{ij}^{(0)w_{ij}^{(0)}} : 1 \leq i,j \leq k\}$.

From Definition 1 we have

$$\{s_{ii}^{(0)10^n} : 1 \leq i \leq k\} = \{s_{ii}^{(0)w_0} : 1 \leq i \leq k\} = \{s_{ij}^{(0)w_{ij}^{(0)}} : 1 \leq i,j \leq k\}.$$

Let us note that only in the internal membrane (labeled by 2) there are applicable rules to configuration $\mathcal{C}_{n-1}$, namely the rules:

$$s_{ij}^{(0)10} t_{j1}^{10p_{j1}} \ldots t_{jk}^{10p_{jk}} \longrightarrow (t_{j1i}^{(0)10p_{j1}} \ldots t_{jki}^{(0)10p_{jk}}, out), \text{ for } 1 \leq i,j \leq k.$$

By the condition of maximal parallelism, each one of these rules will be applied $\frac{w_{ij}^{(0)}}{10}$ times. Consequently, we have:

$$\begin{cases} C_n(1) = \{t_{jui}^{(0)w_{ij}^{(0)}p_{ju}} : 1 \leq i,j,u \leq k\}, \\ C_n(2) = \{t_{ij}^{kw_0p_{ij} - w_0p_{ij}} : 1 \leq i,j \leq k\} = \{t_{ij}^{kw_0p_{ij} - \sum_{u=1}^{k} w_{ui}^{(0)}p_{ij}} : 1 \leq i,j \leq k\}. \end{cases}$$

Hence, the result is true for $m = 0$. Let $m < n-1$. Let us suppose that the result holds for m, that is,

$$\begin{cases} C_{(n-1)+2m}(1) = \emptyset, \\ C_{(n-1)+2m}(2) = \{t_{ij}^{kw_0p_{ij}} : 1 \leq i,j \leq k\} \cup \{s_{ij}^{(m)w_{ij}^{(m)}} : 1 \leq i,j \leq k\}, \\ C_{(n-1)+2m+1}(1) = \{t_{jui}^{(m)w_{ij}^{(m)}p_{ju}} : 1 \leq i,j,u \leq k\}, \\ C_{(n-1)+2m+1}(2) = \{t_{ij}^{kw_0p_{ij} - \sum_{u=1}^{k} w_{ui}^{(m)}p_{ij}} : 1 \leq i,j \leq k\}. \end{cases}$$

In order to obtain the configuration $\mathcal{C}_{(n-1)+2m+2}$, let us note that there are applicable rules only in the skin membrane of the configuration $\mathcal{C}_{(n-1)+2m+1}$. Specifically, this is the case with the rules:

$$t_{jui}^{(m)10p_{ju}} \longrightarrow (t_{ju}^{10p_{ju}} s_{iu}^{(m+1)10p_{ju}}, in_2), \text{ for } 1 \leq i, j, u \leq k.$$

By the condition of maximal parallelism, each one of these rules will be applied $\frac{w_{ij}^{(m)} p_{js}}{10p_{js}} = \frac{w_{ij}^{(m)}}{10}$ times. Hence,

$$\begin{cases} C_{(n-1)+2m+2}(1) = \emptyset, \\ C_{(n-1)+2m+2}(2) = \{t_{ij}^{kw_0 p_{ij} - \sum_{u=1}^{k} w_{ui}^{(m)} p_{ij}} : 1 \leq i, j \leq k\} \cup \\ \qquad \{t_{ij}^{w_{1i}^{(m)} p_{ij} + \cdots + w_{ki}^{(m)} p_{ij}} : 1 \leq i, j \leq k\} \cup \\ \qquad \{s_{ij}^{(m+1) w_{i1}^{(m)} p_{1j} + \cdots + w_{ik}^{(m)} p_{kj}} : 1 \leq i, j \leq k\} \\ \qquad = \{t_{ij}^{kw_0 p_{ij}} : 1 \leq i, j \leq k\} \cup \{s_{ij}^{(m+1) w_{ij}^{(m+1)}} : 1 \leq i, j \leq k\}. \end{cases}$$

In order to obtain the configuration $\mathcal{C}_{(n-1)+2m+3}$, let us note that there are applicable rules only in the internal membrane of the configuration $\mathcal{C}_{(n-1)+2m+2}$ (let us recall that $m < n-1$, and then $m+1 < n$). Specifically the following rules can be applied:

$$s_{ij}^{(m+1)10} t_{j1}^{10p_{j1}} \ldots t_{jk}^{10p_{jk}} \longrightarrow (t_{j1i}^{(m+1)10p_{j1}} \ldots p_{jki}^{(m+1)10p_{jk}}, out), \ 1 \leq i, j \leq k.$$

By the condition of maximal parallelism, each one of these rules will be applied $\frac{w_{ij}^{(m+1)}}{10}$ times. Thus,

$$\begin{cases} C_{(n-1)+2m+3}(1) = \{t_{jui}^{(m+1) w_{ij}^{(m+1)} p_{ju}} : 1 \leq i, j, u \leq k\}, \\ C_{(n-1)+2m+3}(2) = \{t_{ij}^{kw_0 p_{ij} - \sum_{u=1}^{k} w_{ui}^{(m+1)} p_{ij}} : 1 \leq i, j \leq k\}. \end{cases}$$

Hence, the result is true for $m+1$, concluding the proof of (c).

(d) From (c) we deduce that

$$\begin{cases} C_{3n-2}(1) = \{t_{jui}^{(n-1) w_{ij}^{(n-1)} p_{ju}} : 1 \leq i, j, u \leq k\}, \\ C_{3n-2}(2) = \{t_{ij}^{kw_0 p_{ij} - \sum_{u=1}^{k} w_{ui}^{(n-1)} p_{ij}} : 1 \leq i, j \leq k\}. \end{cases}$$

Let us observe that there are applicable rules only in the skin membrane of the configuration $\mathcal{C}_{3n-2}$. Specifically, the following rules can be used:

$$t_{jui}^{(n-1)10p_{ju}} \longrightarrow (t_{ju}^{10p_{ju}} s_{iu}^{(n)10p_{ju}}, in_2), \text{ for } 1 \leq i, j, u \leq k.$$

By the condition of maximal parallelism, each one of these rules will be applied $\frac{w_{ij}^{(n-1)} p_{js}}{10p_{js}} = \frac{w_{ij}^{(n-1)}}{10}$ times. Consequently, we have:

$$\begin{cases} C_{3n-1}(1) = \emptyset, \\ C_{3n-1}(2) = \{t_{ij}^{kw_0p_{ij} - \sum_{u=1}^{k} w_{ui}^{(n-1)} p_{ij}} : 1 \leq i,j \leq k\} \cup \\ \qquad \{t_{ij}^{w_{1i}^{(n-1)} p_{ij} + \cdots + w_{ki}^{(n-1)} p_{ij}} : 1 \leq i,j \leq k\} \cup \\ \qquad \{s_{ij}^{(n) w_{i1}^{(n-1)} p_{1j} + \cdots + w_{ik}^{(n-1)} p_{kj}} : 1 \leq i,j \leq k\} \\ \qquad = \{t_{ij}^{kw_0p_{ij}} : 1 \leq i,j \leq k\} \cup \{s_{ij}^{(n) w_{ij}^{(n)}} : 1 \leq i,j \leq k\}. \end{cases}$$

Then, to obtain the configuration $\mathcal{C}_{3n}$, it is possible to apply rules only in the internal membrane of the configuration $\mathcal{C}_{3n-1}$, namely, the following rules:

$$s_{ij}^{(n)} \longrightarrow (s_{ij}, out), \text{ for } 1 \leq i,j \leq k.$$

Thus, $C_{3n}(1) = \{s_{ij}^{w_{ij}^{(n)}} : 1 \leq i,j \leq k\}$, and $C_{3n}(2) = \{t_{ij}^{kw_0p_{ij}} : 1 \leq i,j \leq k\}$.

Then, there are applicable rules only in the skin membrane of the configuration $\mathcal{C}_{3n}$. Specifically, the following rules can be applied:

$$s_{ij} \longrightarrow (s_{ij}, out), \text{ for } 1 \leq i,j \leq k.$$

Hence, we have: $C_{3n+1}(1) = \emptyset; C_{3n+1}(2) = \{t_{ij}^{kw_0p_{ij}} : 1 \leq i,j \leq k\}$, and $C_{3n+1}(env) = \{s_{ij}^{w_{ij}^{(n)}} : 1 \leq i,j \leq k\}$.

Then, there is no other applicable rule to configuration $\mathcal{C}_{3n+1}$. Consequently, this configuration is a halting one. □

Theorem 2. *Let $k \geq 1, n \geq 2$. Let $P_k = (p_{ij})_{1 \leq i,j \leq k}$ be the transition matrix associated with a finite and homogeneous Markov chain. Let $\Pi(P_k, n)$ be the P system defined in Section 3.1. The output of the only computation of this system (that is, the content of the environment in the halting configuration) codifies the matrix $B(n,n) = 10^n \cdot P_k^n$.*

Proof. From (d) in Theorem 1 we deduce that the configuration $\mathcal{C}_{3n+1}$ of the system $\Pi(P_k, n)$ is a halting one and, moreover, the multiset associated with the environment is $C_{3n+1}(env) = \{s_{ij}^{w_{ij}^{(n)}} : 1 \leq i,j \leq k\}$.

Directly from Proposition 1, with $r = n$, it follows that $\forall i \; \forall j \; (1 \leq i,j \leq k \rightarrow w_{ij}^{(n)} = b_{ij}^{(n,n)})$. That is, the multiplicity $w_{ij}^{(n)}$ of the object s_{ij} in the environment of the halting configuration $\mathcal{C}_{3n+1}$ coincides with $b_{ij}^{(n,n)}$, the (i,j)–term of the matrix $B(n,n) = 10^n \cdot P_k^n$. □

4 Conclusions

In this paper, a deterministic P system with external output associated with a natural number, $n \geq 2$, and to a finite and homogeneous Markov chain, is described. This P system provides the n–th power of the transition matrix associated with the Markov chain, encoding the power in the environment of a halting configuration of the system.

In [1] this problem has been addressed by means of a molecular DNA based algorithms, giving an *estimation* of this power in polynomial time, and providing a new approach to the problem of computing the limit of a Markov chain.

The solution presented in this work is placed in the scope of the cellular computing with membranes. It is a *semi–uniform* solution, because for each Markov chain and each power, a specific P system is designed. The solution is efficient, because it is linear in the power and independent of the number of states of the Markov chain. Furthermore, the amount of resources initially required to construct the system is polynomial in the power and in the order of the Markov chain.

The paper also provides a new example of formal verification of P systems designed to solve a problem, following a specific methodology valid in some cases like those considered in the paper. These examples are always interesting, for instance, in order to find systematic processes of formal verification in a model of computation oriented to machines, like the cellular model, in where it is well known that the mechanisms of verification are often a very hard task.

Acknowledgement

The third author wishes to acknowledge the support of the Project TIN2005-09345-C04-01 of the Ministerio de Educación y Ciencia of Spain, cofinanced by FEDER funds, and the support of the Project of Excelence TIC-581 of the Junta de Andalucía.

References

1. M. Cardona, M.A. Colomer, J. Conde, J.M. Miret, J. Miró, A. Zaragoza, Markov chains: computing limit existence and approximations with DNA, *Biosystems*, 81, 3 (2005) 261–266.
2. O. Häggström, *Finite Markov chains and algorithmic applications*, London Mathematical Society, Cambridge University Press, 2002.
3. R. Nelson, *Probability, stochastic processes, and queueing theory:the mathematics of computer performance modeling*, Springer-Verlag, New York, 1995.
4. Gh. Păun, Computing with membranes, *Journal of Computer and System Sciences*, 61, 1 (2000), 108–143, and *Turku Center for Computer Science-TUCS Report* Nr. 208, 1998.
5. Gh. Păun, *Membrane Computing. An Introduction*, Springer-Verlag, Berlin, 2002.

Approximation Classes for Real Number Optimization Problems

Uffe Flarup and Klaus Meer*

Department of Mathematics and Computer Science
Syddansk Universitet, Campusvej 55, 5230 Odense M, Denmark
{flarup, meer}@imada.sdu.dk

Abstract. A fundamental research area in relation with analyzing the complexity of optimization problems are approximation algorithms. For combinatorial optimization a vast theory of approximation algorithms has been developed, see [1]. Many natural optimization problems involve real numbers and thus an uncountable search space of feasible solutions. A uniform complexity theory for real number decision problems was introduced by Blum, Shub, and Smale [4]. However, approximation algorithms were not yet formally studied in their model.

In this paper we develop a structural theory of optimization problems and approximation algorithms for the BSS model similar to the above mentioned one for combinatorial optimization. We introduce a class $NPO_{\mathbb{R}}$ of real optimization problems closely related to $NP_{\mathbb{R}}$. The class $NPO_{\mathbb{R}}$ has four natural subclasses. For each of those we introduce and study real approximation classes $APX_{\mathbb{R}}$ and $PTAS_{\mathbb{R}}$ together with reducibility and completeness notions. As main results we establish the existence of natural complete problems for all these classes.

1 Introduction

Many important problems in mathematics and computer science are optimization problems. In the framework of complexity theory a lot of such problems are obtained as optimization versions of NP-complete decision problems. Typical examples are the TSP problem, the MAX-SAT problem or the Knapsack problem in its optimization form. Such problems constitute the class NPO of combinatorial optimization problems having an exponential size search space. An important field of complexity theory is dealing with approximation properties for such NP-hard optimization problems [1].

Another tradition of approximation algorithms comes from continuous mathematics. A typical example is Newton's method for approximating a zero of a function. Another one is the approximation of the number of solutions of a polynomial system, being extremely important for numerical solution algorithms, see [8] as an example for such investigations.

* Partially supported by the IST Programme of the European Community, under the PASCAL Network of Excellence, IST-2002-506778 and by the Danish Natural Science Research Council SNF. This publication only reflects the authors' views.

C.S. Calude et al. (Eds.): UC 2006, LNCS 4135, pp. 86–100, 2006.

A structural complexity theory for real number problems along the lines of the classical study of decision problems was developed in [4], where the BSS model was introduced. It is a hypercomputer in that the possibility to use real number machine constants implies the computability of any function $f : \mathbb{N} \to \mathbb{N}$ in the model. In [4] the authors study a real version $NP_{\mathbb{R}}$ of NP together with an analogue $P_{\mathbb{R}} = NP_{\mathbb{R}}?$ of the famous P versus NP question. A typical $NP_{\mathbb{R}}$-complete problem is QPS which asks for deciding whether a system of quadratic real polynomials has a common real zero. Real number decision problems have been extensively studied in recent years, see [11] and [7] for getting a good impression. Given the immense importance of the theory of approximation algorithms within the Turing model it is tempting to develop a similar framework for real number optimization problems as well. A typical example of such a problem is to approximate the maximal number of commonly solvable real polynomials in a system. This problem has strong relations to a potential real version of the PCP theorem as shown in [9]. Another such problem is to minimize the norm of a root of a given polynomial; good bounds on this minimum (i.e. good approximations) are of huge importance for algorithms dealing with problems in semi-algebraic geometry [2,10].

In the present paper we start the structural investigation of approximation algorithms for certain classes of real number optimization problems. The idea is to take the classical notions of approximation algorithms, polynomial approximation schemes and AP-reducibility as starting point and transfer them in a suitable way to the real number model. However, in doing so one has to take care of some intrinsic differences. One of the most fundamental such difference is that our optimization problems constituting the real class $NPO_{\mathbb{R}}$ allow for uncountable sets of feasible solutions. This is a quite natural aspect of real number problems, yet it leads to some subtle differences when defining the main concepts below. As example, approximation algorithms will not have to compute a feasible solution with an approximation ratio but only guarantee its existence. This requirement is intrinsically related to the impossibility of computing in general polynomial roots exactly by a BSS algorithm. Another difference will be that $NPO_{\mathbb{R}}$ naturally gives rise to study four subclasses, each representing a different kind of optimization problems. We introduce as well real versions $APX_{\mathbb{R}}, PTAS_{\mathbb{R}}$ of the corresponding discrete classes; they similarly have four natural subclasses.

Our main results are as follows: For each of the four classes that define $NPO_{\mathbb{R}}, APX_{\mathbb{R}}$, and $PTAS_{\mathbb{R}}$ we show the existence of complete problems under a real version of AP-reducibility (to be defined). This holds both for maximization and minimization problems. In addition, some further completeness results for natural optimization problems will be given.

Our hope is that the present approach will serve as starting point for an analysis of approximation algorithms for real number optimization problems as fruitful as the corresponding approach for combinatorial optimization problems.

2 Definitions

In the BSS model over $\mathbb{R}$ real numbers are considered as entities. The basic arithmetic operations $+, -, *, /$ can be performed at unit cost and there is a test operation "is $x_i \geq 0$?" reflecting the underlying ordering of the reals. Decision problems are subsets $L \subseteq \mathbb{R}^\infty := \bigcup_{n=1}^{\infty} \mathbb{R}^n$. The (algebraic) size of a point $x \in \mathbb{R}^n$, denoted $|x|$, is n. Having fixed these notions it is easy to define real analogues $P_\mathbb{R}$ and $NP_\mathbb{R}$ of the classes P and NP as well as the notion of $NP_\mathbb{R}$-completeness. $DNP_\mathbb{R}$ (Digital $NP_\mathbb{R}$) is the subset of $NP_\mathbb{R}$ where we are restricted to guesses encoded over $\mathbb{Z}_2$. For more details on the BSS model we refer to [3].

2.1 The Class $NPO_\mathbb{R}$ and Its Subclasses

In classical complexity the most relevant class of optimization problems is NPO which consist of combinatorial optimization problems with exponential size search space. We shall define a real number counterpart $NPO_\mathbb{R}$. There is more than just one immediate way of doing so. Whereas the problem instances for such a class are encoded over $\mathbb{R}$, problems naturally differ as to where feasible solutions are located and what quantity is measured by the objective function. Feasible solutions could be encoded over both $\mathbb{R}$ or $\mathbb{Z}_2$ depending on the search space of the problem. Similarly, the measure of a feasible solution could either be a non-negative real number or a non-negative integer depending on the problem. It turns out that all four resulting combinations are meaningful.

In the definition below we use R_s to denote the ring over which feasible solutions are encoded. We consider $R_s \in \{\mathbb{R}, \mathbb{Z}_2\}$. By R_m we denote the image set of a measure function; here $R_m \in \{\mathbb{R}^{\geq 0}, \mathbb{Z}_2^\infty\}$ are suitable choices.

We are now ready to define real analogues of NPO:

Definition 1. *a) A triple $\mathcal{P} := (\mathcal{I}, \{Sol(x)\}_{x\in\mathcal{I}}, m)$ belongs to the class $NPO^{R_s,R_m}_{\mathbb{R},max}$ if it is a maximization problem over $\mathbb{R}$ that consists of the following:*

i) A set $\mathcal{I} \subseteq \mathbb{R}^\infty$ as instances of the problem;
ii) for every $x \in \mathcal{I}$ a set $Sol(x) \subseteq R_s^\infty$ of feasible solutions. For every $x \in \mathcal{I}$ and every $y \in Sol(x)$ we require $|y| \leq p(|x|)$ for a fixed polynomial p;
iii) a function $m : \{(x, y)|x \in \mathcal{I}, y \in Sol(x)\} \to R_m$. The value $m(x, y)$ is called the measure *of the feasible solution y.*

In addition, the following has to hold:

iv) For a given $x \in \mathbb{R}^\infty$ it is decidable in polynomial time in $|x|$ in the BSS model if $x \in \mathcal{I}$;
v) for all $x \in \mathcal{I}$ and for any arbitrary $y \in R_s^\infty$ of size at most $p(|x|)$ it is decidable in polynomial time in $|x|$ in the BSS model if $y \in Sol(x)$;
vi) for all $x \in \mathcal{I}$ and for all $y \in Sol(x)$ the measure function m is computable in polynomial time in $|x|$ in the BSS model.

We are looking for the optimal solution measure $m^*(x) = \sup_{y \in Sol(x)} m(x,y)$.

b) The classes $NPO^{R_s,R_m}_{\mathbb{R},min}$ *of minimization problems is defined similarly.*

c) $NPO^{R_s,R_m}_{\mathbb{R}}$ *is the union of classes* $NPO^{R_s,R_m}_{\mathbb{R},max}$ *and* $NPO^{R_s,R_m}_{\mathbb{R},min}$. *Similarly,* $NPO_{\mathbb{R}}$ *denotes the union of all optimization problems defined above.*

Remark 1. a) In case the measure function takes values in $\mathbb{R}^{\geq 0}$ we just output a single real number. If the measure values are non-negative integers we output the number in binary representation.

b) Since a single real number has algebraic size 1 deciding if an arbitrary real number is integral cannot be done in polynomial time in its algebraic size. Thus, if some components of an instance are required to be integral or rational numbers we need to encode these in binary in order to satisfy condition a,iv) above.

b) $NPO^{\mathbb{R},\mathbb{R}^{\geq 0}}_{\mathbb{R},max}$ is the only class introduced in this paper for which problems can have instances with a non-empty set of feasible solutions and $m^*(x)$ undefined. If $R_s = \mathbb{Z}_2$ this follows from the finite search space; and if $R_m = \mathbb{Z}_2^\infty$ it is a consequence of part a,vi) of the definition.

Example 1. We illustrate the four variants of optimization problems by giving examples of problems in each class. $NPO^{\mathbb{Z}_2,\mathbb{Z}_2^\infty}_{\mathbb{R}}$ contains problems like Binpacking with real weights on the items or maximizing the number of real polynomials in a given system that share a zero in $\{0,1\}^n$. The TSP problem with real weights on the edges and the Knapsack optimization problem with real numbers as weights belong to $NPO^{\mathbb{Z}_2,\mathbb{R}^{\geq 0}}_{\mathbb{R}}$. Typical problems in $NPO^{\mathbb{R},\mathbb{Z}_2^\infty}_{\mathbb{R}}$ are the counterpart of the above problem for polynomial systems when solutions in $\mathbb{R}^n$ are searched. Similarly, the problem of finding the maximal solvable subsystem of a given system of linear inequalities is in this class. Finally, minimizing the function value of a multivariate real polynomial, minimizing the maximum norm of a root of such a polynomial, or the linear and quadratic programming problems with real data are members in $NPO^{\mathbb{R},\mathbb{R}^{\geq 0}}_{\mathbb{R}}$. We easily obtain the following relations between these classes:

$$NPO^{\mathbb{Z}_2,\mathbb{Z}_2^\infty}_{\mathbb{R}} \subset NPO^{\mathbb{Z}_2,\mathbb{R}^{\geq 0}}_{\mathbb{R}} \cap NPO^{\mathbb{R},\mathbb{Z}_2^\infty}_{\mathbb{R}} \subset NPO^{\mathbb{Z}_2,\mathbb{R}^{\geq 0}}_{\mathbb{R}} \cup NPO^{\mathbb{R},\mathbb{Z}_2^\infty}_{\mathbb{R}} \subset NPO^{\mathbb{R},\mathbb{R}^{\geq 0}}_{\mathbb{R}}$$

Definition 2. *An optimization problem* $\mathcal{P}$ *in class* $NPO_{\mathbb{R}}$ *is* polynomially bounded *if there exists a polynomial* q *such that for all* x *the value* $m(x,y)$ *is bounded by* $q(|x|)$ *for all feasible solutions* y.

2.2 Approximation Classes $APX_{\mathbb{R}}$, $PTAS_{\mathbb{R}}$ and $FPTAS_{\mathbb{R}}$

In order to study approximation properties of these types of optimization problems we define approximation classes similar to those well known in the classical setting. The definition, however, needs some care.

Definition 3. *Given an instance x and a feasible solution $y \in Sol(x)$ with measure v, the* ratio *of this feasible solution is defined as $R(x,v) = max(\frac{v}{m^*(x)}, \frac{m^*(x)}{v})$. The ratio is undefined if either $v = 0$, $m^*(x) = 0$ or $m^*(x)$ is undefined. This definition applies to both maximization and minimization problems.*

Definition 4. *An optimization problem $\mathcal{P}$ in class $NPO^{R_s,R_m}_{\mathbb{R},max}$ is in class $APX^{R_s,R_m}_{\mathbb{R},max}$ if there exists a real number $r > 1$ and polynomial time BSS machines M_1, M_2 and M_3 such that:*

i) For any input $x \in \mathcal{I}$ machine M_1 decides if $Sol(x) = \emptyset$;
ii) for any input $x \in \mathcal{I}$, $Sol(x) \neq \emptyset$ machine M_2 decides if $m^*(x) = 0$ or $m^*(x)$ is undefined;
iii) for any input $x \in \mathcal{I}$, $Sol(x) \neq \emptyset$, $m^*(x)$ is defined and $m^*(x) \neq 0$ machine M_3 computes a value $v \in R_m$ such that there exists a $y \in Sol(x)$ with $m(x,y) \geq v$ and $R(x,v) \leq r$.

Similarly for optimization problems in class $NPO^{R_s,R_m}_{\mathbb{R},min}$.

Remark 2. Most important we do not require M_3 to compute a feasible solution y. The output of the computation is only a bound for the measure of a feasible solution which *must* exist and this bound must be within a constant factor of the measure of the optimal solution. The reason for this is that though one might guarantee such a y to exist it is impossible to compute it by a BSS algorithm (f.e. if y has to satisfy an equation $y^2 - s = 0$ for some $\sqrt{s}$ not belonging to the algebraic closure of the machine constants).

$APX_{\mathbb{R}}$ is then defined as the union of all the subclasses. In a similar way we can define approximation classes *log-*$APX_{\mathbb{R}}$, *poly-*$APX_{\mathbb{R}}$ and *exp-*$APX_{\mathbb{R}}$. For these classes we require the *ratio* of the feasible solution y to be logarithmic, polynomial and exponential in $|x|$, respectively. Note that membership in *exp-*$APX_{\mathbb{R}}$ requires that we in polynomial time can decide if the set of feasible solutions is empty.

Finally, the class $PTAS_{\mathbb{R}}$ of problems allowing a polynomial time approximation scheme is defined by changing Definition 4 in the obvious way: The ratio r now is given as part of the input to M_1, M_2 and M_3. For approximation class $FPTAS_{\mathbb{R}}$ we additionally require the running time of M_1, M_2 and M_3 to be polynomial in $|x|$ and $\frac{1}{r-1}$.

Example 2. The following is an example of a problem in $APX^{\mathbb{R},\mathbb{Z}_2^\infty}_{\mathbb{R},max}$. Let $k \in \mathbb{N}$ be fixed and > 1. The problem MAX QUADRATIC POLYNOMIAL SYSTEM(k), for short MAX QPS(k), is defined as follows: Given polynomials $p_1, \ldots, p_m$ with real coefficients in n variables, each of degree at most 2, each depending on at most 3 of the variables $x_1, \ldots, x_n$ and each variable occurs in at most k of the polynomials, find the maximal number of polynomials p_i, $1 \leq i \leq m$, that have a common root in $\mathbb{R}^n$. For $k > 2$ MAX QPS(k) is an $NP_{\mathbb{R}}$-hard maximization problem due to [4]. It is not hard to see that MAX QPS(k) is in class $APX^{\mathbb{R},\mathbb{Z}_2^\infty}_{\mathbb{R},max}$. First decide for each p_i whether it has a root; this is polynomial time decidable

since only three variables occur in each single p_i. Then, for the solvable ones fixing the occurring variables will at most influence $3(k-1)$ other polynomials. Thus an approximation ratio $r := 3(k-1)+1$ can be guaranteed. □

2.3 Approximation Preserving Reductions

In order to show the existence of complete problems for the above classes of optimization problems we define approximation preserving reductions.

Definition 5. *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be maximization problems in class $NPO_{\mathbb{R}}$. $\mathcal{P}_1$ is AP-reducible to $\mathcal{P}_2$, denoted $\mathcal{P}_1 \leq_{AP} \mathcal{P}_2$, if functions f and g and a real constant $\alpha \geq 1$ exists such that:*

i) For any $x \in \mathcal{I}_1$ and any real number $r > 1$ it is $f(x,r) \in \mathcal{I}_2$;
ii) for any $x \in \mathcal{I}_1$ and any real number $r > 1$ if $Sol_1(x) \neq \emptyset$, then $Sol_2(f(x,r)) \neq \emptyset$;
iii) $f(x,r)$ is polynomial time computable in $|x|$;
iv) for any $x \in \mathcal{I}_1$ and any fixed real number $r > 1$ if there exists a $y \in Sol_2(f(x,r))$ with measure at least v such that $R_2(f(x,r),v) \leq r$, then there exists a $z \in Sol_1(x)$ with measure at least w such that $R_1(x,w) \leq 1+\alpha\cdot(r-1)$, and $w = g(x,r,v)$;
v) $g(x,r,v)$ is polynomial time computable in $|x|$.

A similar definition can be made for minimization problems.

Remark 3. a) In the classical setting we also require a function which computes a feasible solution to the instance of $\mathcal{P}_1$ given a feasible solution to the instance of $\mathcal{P}_2$ we reduced it to. As mentioned previously we do not require the actual solution to be computed; the fact that it exists is sufficient.

b) Most AP-reductions in this paper will use $\alpha = 1$, do not depend on r, and satisfy $g(x,r,v) = v$. We will explicitly state when this is not the case.

Definition 6. *Given a class $\mathcal{C}$ of optimization problems, $\mathcal{P}$ is* $\mathcal{C}$-hard *with respect to AP-reducibility if for all $\mathcal{P}' \in \mathcal{C}$, $\mathcal{P}' \leq_{AP} \mathcal{P}$. If furthermore $\mathcal{P} \in \mathcal{C}$, then $\mathcal{P}$ is* $\mathcal{C}$-complete *with respect to AP-reducibility.*

2.4 Relation of $NPO_{\mathbb{R}}$ to Decision Problems

Let us shortly justify our definition of $NPO_{\mathbb{R}}$ and its subclasses from another point of view, namely the relation of each class to a corresponding class of decision problems.

In the classical setting PO is the subset of NPO for which an optimal solution and its measure can be computed in polynomial time. It is $PO = NPO$ iff $P = NP$. Just as we have defined four subclasses of $NPO_{\mathbb{R}}$ we can define four subclasses of $PO_{\mathbb{R}}$.

In order for an optimization problem in $NPO_{\mathbb{R}}^{R_s,R_m}$ to be in the subclass $PO_{\mathbb{R}}^{R_s,R_m}$ we require that we in polynomial time can decide for any given instance

if the set of feasible solutions is empty. Then for each of the four classes $PO_{\mathbb{R}}^{R_s,R_m}$ introduced we add the following additional requirements necessary for a problem in order to belong to the class:

For $PO_{\mathbb{R}}^{\mathbb{Z}_2,\mathbb{Z}_2^\infty}$ and $PO_{\mathbb{R}}^{\mathbb{Z}_2,\mathbb{R}^{\geq 0}}$ an optimal solution and for $PO_{\mathbb{R}}^{\mathbb{R},\mathbb{Z}_2^\infty}$ the optimal measure value have to be computable in polynomial time. Finally, for $PO_{\mathbb{R}}^{\mathbb{R},\mathbb{R}^{\geq 0}}$ the following questions have to be decidable in polynomial time: given x, is $m^*(x)$ is defined; given $v \in \mathbb{R}^{\geq 0}$ is there a solution with measure v; given $v_1 < v_2 \in \mathbb{R}^{\geq 0}$, is there is a solution with measure in $]v_1, v_2[$.

Theorem 1. *The following relations hold between the classes of optimization problems and the corresponding classes of decision problems:*

a) $P_{\mathbb{R}} = NP_{\mathbb{R}} \iff PO_{\mathbb{R}}^{\mathbb{R},\mathbb{R}^{\geq 0}} = NPO_{\mathbb{R}}^{\mathbb{R},\mathbb{R}^{\geq 0}} \iff PO_{\mathbb{R}}^{\mathbb{R},\mathbb{Z}_2^\infty} = NPO_{\mathbb{R}}^{\mathbb{R},\mathbb{Z}_2^\infty}$

b) $P_{\mathbb{R}} = DNP_{\mathbb{R}} \iff PO_{\mathbb{R}}^{\mathbb{Z}_2,\mathbb{R}^{\geq 0}} = NPO_{\mathbb{R}}^{\mathbb{Z}_2,\mathbb{R}^{\geq 0}} \iff PO_{\mathbb{R}}^{\mathbb{Z}_2,\mathbb{Z}_2^\infty} = NPO_{\mathbb{R}}^{\mathbb{Z}_2,\mathbb{Z}_2^\infty}$

3 Completeness Results for Maximization

We turn to the main results of this paper. In this section we show the existence of complete problems for each of the classes of maximization problems $NPO_{\mathbb{R},max}^{\mathbb{Z}_2,\mathbb{Z}_2^\infty}$, $NPO_{\mathbb{R},max}^{\mathbb{Z}_2,\mathbb{R}^{\geq 0}}$, $NPO_{\mathbb{R},max}^{\mathbb{R},\mathbb{Z}_2^\infty}$ and $NPO_{\mathbb{R},max}^{\mathbb{R},\mathbb{R}^{\geq 0}}$ as well as the corresponding subclasses of $APX_{\mathbb{R}}$. Minimization will be treated in the next section.

3.1 $NPO_{\mathbb{R}}$-Completeness

For each of the four classes completeness of a particular problem will be shown. Let us start with

Example 3. The MAX WEIGHTED 4FEAS problem is defined as:
Input: $n \in \mathbb{N}$, a degree 4 polynomial with real coefficients in n variables. Each variable has a non-negative integer weight;
Feasible Solution: $x \in \mathbb{R}^n$ such that the polynomial evaluates to zero;
Measure: Sum of weights of all variables which have been assigned a value different from zero.

The problem is easily shown to belong to class $NPO_{\mathbb{R},max}^{\mathbb{R},\mathbb{Z}_2^\infty}$. Note that deciding existence of a feasible solution is $NP_{\mathbb{R}}$-complete so this problem does not belong to class *exp-APX*$_{\mathbb{R}}$ if $P_{\mathbb{R}} \neq NP_{\mathbb{R}}$.

Theorem 2. *MAX WEIGHTED 4FEAS is complete for* $NPO_{\mathbb{R},max}^{\mathbb{R},\mathbb{Z}_2^\infty}$ *under AP-reducibility.*

Proof. Let $\mathcal{P} := (\mathcal{I}, \{Sol(x)\}_{x\in\mathcal{I}}, m)$ be a problem in class $NPO_{\mathbb{R},max}^{\mathbb{R},\mathbb{Z}_2^\infty}$. We design an AP-reduction from $\mathcal{P}$ to MAX WEIGHTED 4FEAS.

Let $x \in \mathcal{I}$ be fixed. Consider the following decision problem: On input (y, v) decide if $y \in Sol(x)$, if v is the binary encoding of an integer, and if $m(x, y) = v$.

Since $\mathcal{P} \in NPO^{\mathbb{R},\mathbb{Z}_2^\infty}_{\mathbb{R},max}$ we can easily construct a BSS machine M_x which in polynomial time in $|x|$ decides this problem. On the other hand, deciding *if* an input (y, v) exists such that M_x accepts is a decision problem in class $NP_{\mathbb{R}}$.

An AP-reduction (f, g) is obtained as follows: On input $x \in \mathcal{I}$ f constructs M_x. It then uses the reduction described in [4] to generate a degree 4 polynomial p which has a real root iff there exists (y, v) which fulfills the above conditions. In p we identify the polynomially many variables which correspond to those registers that store the binary representation of v at the beginning of the computation performed by M_x on input (y, v). Each of these variables is assigned a weight 2^i corresponding to which part of the binary representation of v it represents. All other variables in p are assigned the weight 0. Adding weights to the variables in p produces a MAX WEIGHTED 4FEAS instance $q = f(x)$. By choosing $\alpha = 1$ and $g(x, r, v) = v$ we obtain an AP-reduction since:

- If $Sol(x) \neq \emptyset$, then there exists a (y, v) such that M_x on input (y, v) will accept. This implies q will have a real root so $Sol(q) \neq \emptyset$. Moreover, f is polynomial time computable.
- For any feasible solution z' to q we can identify the variables which correspond to the registers at the beginning of the computation performed by M_x on input (y, v). This y is in $Sol(x)$ and has measure v. The assignment of weights to the variables in q implies that $m(x, y)$ equals the measure of the feasible solution z' of q. This especially holds for optimal feasible solutions, so we have $m^*(x) = m^*(q)$. If there exists a feasible solution s' for q with measure w such that $R(q, w) \leq r$, then there exists a $y \in Sol(x)$ with measure w such that $R(x, w) \leq r$. □

Example 4. The MAX VARIABLE SUM problem is defined as:
INPUT: $n \in \mathbb{N}$, a degree 4 polynomial with real coefficients in n variables;
FEASIBLE SOLUTION: $x \in \mathbb{R}^n$ such that the polynomial evaluates to 0;
MEASURE: Sum $\sum_{i=1}^{n} x_i$ of all components of x.

Theorem 3. *MAX VARIABLE SUM is complete for* $NPO^{\mathbb{R},\mathbb{R}^{\geq 0}}_{\mathbb{R},max}$ *under AP-reducibility.*

Proof. Let $\mathcal{P} := (\mathcal{I}, \{Sol(x)\}_{x\in\mathcal{I}}, m)$ be a problem in class $NPO^{\mathbb{R},\mathbb{R}^{\geq 0}}_{\mathbb{R},max}$. We will show $\mathcal{P} \leq_{AP}$ MAX VARIABLE SUM. Let $x \in \mathcal{I}$ be a fixed instance for $\mathcal{P}$. We once more consider machine M_x which on input (y, v) decides if $y \in Sol(x)$ and if $m(x, y) = v$. Again the corresponding decision problem is in $NP_{\mathbb{R}}$. It can be reduced in polynomial time to a 4FEAS instance, i.e. the question whether a polynomial p of degree 4 has a real zero. Moreover one particular component of each root z^*, say z_1^*, represents the value v of the guess (y, v) accepted by M_x. Let $\hat{n}$ be the number of variables that p depends on. We introduce $\hat{n} - 1$ new variables denoted by $u_2, \ldots, u_{\hat{n}}$. Construct a MAX VARIABLE SUM instance as follows:

$$q(z, u) = p(z) + (z_2 + u_2)^2 + (z_3 + u_3)^2 + \ldots + (z_{\hat{n}} + u_{\hat{n}})^2$$

Clearly $q(z,u)$ has a real root iff $p(z)$ does. In order for $q(z,u)$ to evaluate to 0 we need $z_i = -u_i$ for $2 \leq i \leq \hat{n}$. Thus, for all feasible solutions to this MAX VARIABLE SUM instance the measure is equal to the value assigned to z_1. □

For the two remaining classes the following problems are complete:

Example 5. The MAX SUBCIRCUIT VALUE problem is defined as:
INPUT: $n \in \mathbb{N}$, an algebraic circuit (see [3]) with n input gates and two output gates; the latter are addressed as first and second output;
FEASIBLE SOLUTION: $x \in \{0,1\}^n$ such that the first output gate evaluates to 1;
MEASURE: Output from the second output gate.

Note that deciding whether the set of feasible solutions is empty is complete for the complexity class $DNP_{\mathbb{R}}$ [6]. The search space is finite, but of exponential size.

Example 6. The MAX WEIGHTED BINARY CIRCUIT is defined as:
INPUT: $n \in \mathbb{N}$, an algebraic circuit with n input gates, a non-negative integer weight for each of the input gates;
FEASIBLE SOLUTION: $x \in \{0,1\}^n$ such that the circuit evaluates to 1;
MEASURE: Sum of weights of all input gates which have been assigned value 1.

Theorem 4. *MAX SUBCIRCUIT VALUE is complete for* $NPO^{\mathbb{Z}_2,\mathbb{R}^{\geq 0}}_{\mathbb{R},max}$ *and MAX WEIGHTED BINARY CIRCUIT is complete in* $NPO^{\mathbb{Z}_2,\mathbb{Z}_2^{\infty}}_{\mathbb{R},max}$ *under AP-reductions.*

Proof. Let $\mathcal{P} := (\mathcal{I}, \{Sol(x)\}_{x\in\mathcal{I}}, m)$ be a problem in $NPO^{\mathbb{Z}_2,\mathbb{R}^{\geq 0}}_{\mathbb{R},max}$. We will show $\mathcal{P} \leq_{AP}$ MAX SUBCIRCUIT VALUE. Let $x \in \mathcal{I}$ be a fixed instance for $\mathcal{P}$. Once again we obtain a decision problem: On input $y \in \{0,1\}^{\infty}$ decide if $y \in Sol(x)$. This decision problem is in $DNP_{\mathbb{R}}$. So it reduces to the Binary Circuit Satisfiability problem in polynomial time. The output gate of this circuit is the first output gate in the instance we produce. Now add another circuit which as input gates has copies of the input gates in the above circuit. This circuit computes the function $m(x,y)$, where x is fixed and y comes from the input gates. The output gate of this circuit is the second output gate in the instance we produce.

For MAX WEIGHTED BINARY CIRCUIT apply ideas from Theorem 2 and Theorem 4. □

Many important problems are polynomially bounded, i.e. the numerical size of all solutions is polynomially bounded in the input size. For the related class $NPO^{\mathbb{R},\mathbb{Z}_2^{\infty}}_{\mathbb{R},max}$ the following problem turns out to be important:

Example 7. The MAX ZERO VARIABLES problem is defined as:
INPUT: $n \in \mathbb{N}$, a degree 4 polynomial with real coefficients in n variables;
FEASIBLE SOLUTION: $x \in \mathbb{R}^n$ such that the polynomial evaluates to 0;
MEASURE: Number of variables that are assigned the value 0.

The problem is in $NPO^{\mathbb{R},\mathbb{Z}_2^\infty}_{\mathbb{R},max}$; since the optimal solution has measure at most n the problem is polynomially bounded. Note that the corresponding problem for linear inequalities is related to finding non-degenerated bases in Linear Programming.

Theorem 5. *MAX ZERO VARIABLES is complete for the set of polynomially bounded problems in class $NPO^{\mathbb{R},\mathbb{Z}_2^\infty}_{\mathbb{R},max}$ with respect to AP-reducibility.*

Proof. Let $\mathcal{P} := (\mathcal{I}, \{Sol(x)\}_{x\in\mathcal{I}}, m)$ be a polynomially bounded problem in class $NPO^{\mathbb{R},\mathbb{Z}_2^\infty}_{\mathbb{R},max}$. Let $x \in \mathcal{I}$ be a fixed instance for this problem. Construct a BSS machine M_x which on input $(y, v) \in \mathbb{R}^\infty \times \mathbb{Z}_2^\infty$ decides if $y \in Sol(x)$ and if $m(x, y) = v$. As before, this decision problem reduces to the 4FEAS decision problem. The proof involves two steps: The first is to modify the reduction in [4] such that the degree 4 polynomial we produce does not have real roots involving 0-components. The second step is to construct an optimization problem by introducing new variables which are allowed to take the value 0 and are related to the value v given to M_x as part of its input.

We assume the reader is familiar with the reduction in [4]. As part of that reduction there occur variables $X_{t,r}$ representing the entry of a register r at step t of the computation of M_x. In the original reduction a variable $X_{t,r}$ can attain the value 0. We change the reduction in such a way that no solution of the resulting polynomial system has a 0-component. Towards this aim introduce additional variables $S_{t,r}$ which encode the value 0 in the following way: If in the computation performed by M_x on input (y, v) register r at time-step t holds a value $z \neq 0$, then we will enforce $S_{t,r} = 1$ and $X_{t,r} = z$. If register r at time-step t holds the value 0, then we will enforce $S_{t,r} = -1$ and $X_{t,r} = -1$.

Using a semi-algebraic description we both can simulate the computation of M_x and enforce avoidance of the value 0 in intermediate computations. The parts of the constructed formula that simulate algebraic operations are a bit more involved than in the usual proof. We have 8 different combinations of whether the 3 registers involved in the algebraic operation hold values different from or equal to 0. Nevertheless the total size of the produced formula is still polynomial in $|x|$. Each strict inequality can as usual be replaced by an equality by introducing one additional variable V. Some of these variables will end up in equalities which might not be fulfilled in an accepting computation of M_x. In such a case these variables can be assigned any value, including 0. This is resolved by introducing one more variable W, for each strict inequality and in addition requiring the equality $V \cdot W = 1$ to be satisfied.

For the second step of the proof we introduce new variables in order to obtain the desired optimization problem: Assume v as part of the input to M_x to be encoded in unary notation. For each register position i involved in this unary encoding we introduce a variable M_i and require $S_{1,i} + M_i = 1$. Note that $S_{1,i}$ represents the entry of register i at the start $t = 1$ of the computation; this entry according to our setting is given the corresponding component of (the unary notation of) v. Thus, for any real root of the constructed formula the number of variables among the M_i's that are assigned the value 0 equals the

value v of the guess (y, v) accepted by M_x. Moreover, since no other variables have been assigned the value 0 in a real root of the formula, this implies the total number of variables assigned the value 0 equals the value v. □

3.2 Completeness in $APX_{\mathbb{R}}$ and $PTAS_{\mathbb{R}}$

In this section we outline how both $APX_{\mathbb{R}}$-complete and $PTAS_{\mathbb{R}}$-complete problems will be constructed. This is done by exploiting a certain property of the previously introduced problems. This property leads in a canonical way to complete problems for both classes (where for the latter the reducibility notion has to be strengthened appropriately).

Definition 7. *We call* $\mathcal{P}$ measure preserving *with respect to* $NPO^{R_s,R_m}_{\mathbb{R},max}$ *under AP-reductions iff the following conditions are satisfied. First,* $\mathcal{P}$ *is* $NPO^{R_s,R_m}_{\mathbb{R},max}$*-complete under AP-reductions. Second, for every problem* $\mathcal{Q} \in NPO^{R_s,R_m}_{\mathbb{R},max}$ *there is a reduction algorithm which preserves the measure. The latter means that for every instance* x_1 *of* $\mathcal{Q}$*, if* x *is the* $\mathcal{P}$ *instance obtained when we apply the AP-reduction from* $\mathcal{Q}$ *to* $\mathcal{P}$ *on input* x_1*, then* x_1 *has a feasible solution with measure* t *iff* x *has a feasible solution with measure* t*.*

We show $APX^{R_s,R_m}_{\mathbb{R},max}$-completeness by using bounds on the measure for instances of a measure preserving problem. Since the proof is very similar to what is done in [5] for the classical setting we only include parts of it.

Definition 8. *Let* $\mathcal{P}$ *be in class* $NPO^{R_s,R_m}_{\mathbb{R},max}$*. We define a maximization problem* $\mathcal{P}'$ *as follows:*
Input: *A triple* $X := (x, k, b)$ *with* x *an instance of* $\mathcal{P}$*,* $k \geq 2$ *integral,* $b > 0$ *a real;*
Feasible Solution: $y \in \mathbb{R}^\infty$ *of size at most* $p(|x|)$ *for a polynomial* p*,* p *only depends on* $\mathcal{P}$*;*
Measure: *Let* $A(X) = (k-2) \cdot b$*. The measure is* $m_{\mathcal{P}'}(X, y) = A(X) + b$ *if either* $y \notin Sol(x)$ *or* $y \in Sol(x)$ *and* $m_{\mathcal{P}}(x, y) < b$*; and it is* $m_{\mathcal{P}'}(X, y) = A(X) + min(k \cdot b, m_{\mathcal{P}}(x, y))$ *if* $y \in Sol(x)$ *and* $m_{\mathcal{P}}(x, y) \geq b$*.*

The following two lemmas are crucial.

Lemma 1. *For all* $\mathcal{P}$ *in* $NPO^{R_s,R_m}_{\mathbb{R},max}$ *we have* $\mathcal{P}' \in APX^{R_s,R_m}_{\mathbb{R},max}$*.*

Lemma 2. *If* $\mathcal{P}$ *is measure preserving w.r.t.* $NPO^{R_s,R_m}_{\mathbb{R},max}$ *and AP-reductions* $\mathcal{P}'$ *is* $APX^{R_s,R_m}_{\mathbb{R},max}$*-hard with respect to AP-reductions.*

Proof. (of Lemma 2) Let $\mathcal{Q}$ be a problem in class $APX^{\mathbb{R},\mathbb{R}^{\geq 0}}_{\mathbb{R},max}$. Then there exists a real constant $k > 1$ such that for all instances of $\mathcal{Q}$ we can in polynomial time can compute a k-approximation. W.l.o.g. we assume $k \in \mathbb{N}$. Since k is fixed once $\mathcal{Q}$ is fixed we can have α and g in our AP-reduction from $\mathcal{Q}$ to $\mathcal{P}'$ depend on k. The AP-reduction works as follows:

Let x_1 be an instance of $\mathcal{Q}$. Compute a real number b such that there exists a feasible solution for x_1 with measure at least b and $R_{\mathcal{Q}}(x_1, b) \leq k$. By using x_1 as input to the AP-reduction from $\mathcal{Q}$ to $\mathcal{P}$ we obtain a $\mathcal{P}$ instance x with the property that x_1 has a feasible solution with measure t iff x has a feasible solution with measure t. We construct a $\mathcal{P}'$ instance $X := (x, k, b)$.

Consider the following two cases:

$k = 2$: We define $\alpha = 1$ and $g(w, r, v) = v$. We see that $m^*_{\mathcal{P}'}(X) = m^*_{\mathcal{P}}(x)$. This is not true for $\mathcal{P}'$ instances in general but it is true in this case since $b \leq m^*_{\mathcal{P}}(x) \leq k \cdot b$ (these inequalities are true because of the measure preserving property of $\mathcal{P}$ and because x was constructed from a $\mathcal{Q}$ instance x_1 for which we had a k-approximation).

Assume y is a feasible solution to X with $R_{\mathcal{P}'}(X, m_{\mathcal{P}'}(X, y)) \leq r$: If y is *not* a feasible solution to x, or y is a feasible solution to x with $m_{\mathcal{P}}(x, y) < b$, then $m_{\mathcal{P}'}(X, y) = b$. But there exists a feasible solution u to x with $m_{\mathcal{P}}(x, u) \geq b$ and this feasible solution u will have $R_{\mathcal{P}}(x, m_{\mathcal{P}}(x, u)) \leq r$. If y *is* a feasible solution to x with $m_{\mathcal{P}}(x, y) \geq b$ then we also have the $R_{\mathcal{P}}(x, m_{\mathcal{P}}(x, y)) \leq r$.

The measure preserving property of $\mathcal{P}$ then implies there exists a feasible solution to x_1 with measure at least $m_{\mathcal{P}'}(X, y)$ and $R_{\mathcal{Q}}(x_1, m_{\mathcal{P}'}(X, y)) \leq r$.

$k > 2$: We define $\alpha = (k - 1)$ and $g(w, r, v) = v - A(X)$. We have $m^*_{\mathcal{P}'}(X) = A(X) + m^*_{\mathcal{P}}(x)$ by the same argument as before. Assume there exists a feasible solution y to the $\mathcal{P}'$ instance X such that $R_{\mathcal{P}'}(X, m_{\mathcal{P}'}(X, y)) \leq r$: If y is *not* a feasible solution to x, or y is a feasible solution to x with $m_{\mathcal{P}}(x, y) < b$, then $m_{\mathcal{P}'}(X, y) = A(X) + b$. Since there exists a feasible solution u to x with measure at least b we obtain

$$\frac{m^*_{\mathcal{P}'}(X)}{m_{\mathcal{P}'}(X, y)} \leq r \Leftrightarrow \frac{m^*_{\mathcal{P}}(x)}{b} \leq (r - 1) \cdot \alpha + 1$$

If y *is* a feasible solution to x with $m_{\mathcal{P}}(x, y) \geq b$ it follows similarly:

$$\frac{m^*_{\mathcal{P}'}(X)}{m_{\mathcal{P}'}(X, y)} \leq r \Rightarrow \frac{m^*_{\mathcal{P}}(x)}{m_{\mathcal{P}}(x, y)} \leq (r-1)\cdot(k-2)+r = (r-1)\cdot\alpha+1 \qquad \square$$

Theorem 6. *All the four subclasses of $APX_{\mathbb{R},max}$ has complete problems under AP-reductions.*

Proof. According to the the previous two lemmas it is sufficient to establish the existence of measure preserving problems in each of the four related subclasses of $NPO_{\mathbb{R},max}$. The completeness proofs for the classes in $NPO_{\mathbb{R},max}$ in the previous section show that all the problems considered together with the given reductions are measure preserving. $\square$

$PTAS_{\mathbb{R}}$-completeness can be shown similarly. First, the notion of F-reductions needs to be transferred to the real number setting in order to establish a reduction that preserves membership in $FPTAS_{\mathbb{R}}$. The reductions in Section 3.1 are all F-reductions as well. Then a canonical problem similar to the one of Definition 8 is used to prove.

Theorem 7. *The above subclasses of $PTAS_{\mathbb{R}}$ do have complete problems under F-reductions.*

4 Completeness Results for Minimization

So far we have been dealing with maximization problems. Nevertheless, the result from the previous section can easily be used to show

Theorem 8. *All four subclasses of $NPO_{\mathbb{R},min}$ have complete problems under AP-reductions.*

Proof. For the problems studied in Examples 3, 4, 5, 6 and 7, respectively, the corresponding minimization versions are complete for their respective minimization classes. The above proofs can be adapted almost word by word. □

In this section we focus on completeness results for some further natural minimization problems. The results so far have all been for problems for which already a potential membership in class $exp\text{-}APX_{\mathbb{R}}$ implies $P_{\mathbb{R}} = NP_{\mathbb{R}}$. The next result deals with a problem for which feasible solutions easily are found. Consider the minimization problem MIN DIFFERENT FUNCTION VALUES: Let $n, m \in \mathbb{N}$, m degree polynomials of degree 4 with real coefficients in n variables be given. We ask for the minimal number of different function values obtained when evaluating the m polynomials in a point $x \in \mathbb{R}^n$. The related maximization problem is easily shown to be in class $PO^{\mathbb{R},\mathbb{Z}_2^\infty}_{\mathbb{R},max}$.

Theorem 9. *MIN DIFFERENT FUNCTION VALUES is complete for the class of polynomially bounded problems in poly-$APX^{\mathbb{R},\mathbb{Z}_2^\infty}_{\mathbb{R},min}$ with respect to AP-reducibility.*

Proof. Let $\mathcal{P} := (\mathcal{I}, \{Sol(x)\}_{x\in\mathcal{I}}, m)$ be polynomially bounded in class *poly-*$APX^{\mathbb{R},\mathbb{Z}_2^\infty}_{\mathbb{R},min}$. Let $x \in \mathcal{I}$ be a fixed instance for this problem and let p be a bound for the measure of all feasible solutions. For each $1 \le i \le p$ we consider a polynomial time BSS machine $M_{x,i}$ which on input y decides if $y \in Sol(x)$ and $m(x, y) \le i$. For each $M_{x,i}$, $1 \le i \le p$, we apply the reduction in [4] to obtain a 4FEAS instance which depends on variables y and additional variables Z_i. We obtain p polynomials $f_1, \ldots, f_p$ with the following properties:

$$\begin{aligned} \exists Z_1 \ \textit{such that} \ f_1(y, Z_1) = 0 &\Longleftrightarrow y \in Sol(x) \ \textit{and} \ m(x, y) \le 1 \\ &\vdots \\ \exists Z_p \ \textit{such that} \ f_p(y, Z_p) = 0 &\Longleftrightarrow y \in Sol(x) \ \textit{and} \ m(x, y) \le p \end{aligned}$$

By the construction of these p polynomials we know their function values always are non-negative. The next step is to obtain the desired instance of MIN DIFFERENT FUNCTION VALUES. The output x' of our AP-reduction is given as p further polynomials $g_1, \ldots, g_p$ obtained as follows:

$$\begin{aligned} g_p(y, Z_p) = f_p(y, Z_p) \ , \ g_{p-1}(y, Z_{p-1}, Z_p) &= f_{p-1}(y, Z_{p-1}) + f_p(y, Z_p) \\ &\vdots \\ g_1(y, Z_1, \ldots, Z_p) &= f_1(y, Z_1) + \ldots + f_p(y, Z_p) \end{aligned}$$

For all assignments of values to the variables of this set of polynomials the following inequalities will hold: $g_p \leq g_{p-1} \leq \ldots \leq g_1$. If the optimal solution to the instance x we reduced from has measure j, then we have for all assignments of values to the variables the following strict inequalities: $g_j < g_{j-1} < \ldots < g_1$. This implies that the optimal solution for x' has measure greater than or equal to the measure of the optimal solution for x, since there will be at least j different function values in x'. On the other hand, we can also find assignments to the variables of $g_p, \ldots, g_j$ such that all g_i's attain the same function value 0. So x and x' have the same optimal measure and the requirements for an AP-reduction are satisfied. □

Our final result deals with minimizing the norm of a root of a given polynomial. This and related questions are important in studying algorithms in semi-algebraic geometry. There, in many situations a first important task is to find good upper bounds on the norm of representatives in each connected component of a semi-algebraic set. For more on this see [10].

Example 8. The MIN L^1-NORM problem is defined as:
INPUT: $n \in \mathbb{N}$, a degree 4 polynomial with real coefficients in n variables;
FEASIBLE SOLUTION: $x \in (\mathbb{R}\backslash\{0\})^n$ such that the polynomial evaluates to 0;
MEASURE: The L^1-norm $\sum_{i=1}^{n} |x_i|$ of x.

Theorem 10. *MIN L^1-NORM is complete for $NPO^{\mathbb{R},\mathbb{R}^{\geq 0}}_{\mathbb{R},min}$ under AP-reducibility.*

Proof. Let $\mathcal{P} := (\mathcal{I}, \{Sol(x)\}_{x\in\mathcal{I}}, m)$ be a problem in class $NPO^{\mathbb{R},\mathbb{R}^{\geq 0}}_{\mathbb{R},min}$ and let $x \in \mathcal{I}$ be a fixed instance of this problem. Construct a BSS machine M_x which on input $(y, v) \in \mathbb{R}^\infty \times \mathbb{R}$ decides if $y \in Sol(x)$ and $m(x, y) = v$. As before, this reduces to a degree 4 polynomial q_x such that q_x has a real root iff an accepting guess for M_x exists. Moreover assume z_n to be the variable which for any root of q_x gets the measure v of the encoded feasible solution.

A straightforward idea now is to consider homogeneous polynomials. Let $\tilde{p}(z_0, z_1, \ldots, z_n)$ be the homogenization of the above q_x with $deg(\tilde{p}) = 4$ and z_0 as the homogenization variable. If $z^* \in \mathbb{R}^n$ is a root of q_x then for all $\lambda \neq 0$ we have $\tilde{p}(\lambda, \lambda \cdot z^*) = \lambda^4 \cdot \tilde{p}(1, z^*) = 0$ and thus the infimum of the L^1-norm for roots of $\tilde{p}$ with $\lambda \neq 0$ is 0. We finally have to increase the infimum to v to obtain the desired result. Define $p(z_0, \ldots, z_n, z_{n+1}) = \tilde{p}(z_0, \ldots, z_n) + (z_n - z_0 \cdot z_{n+1})^2$. Now for any root $(z_0^*, \ldots, z_{n+1}^*)$ of p with $z_0^* \neq 0$ we have $z_{n+1}^* = \frac{z_n^*}{z_0^*}$ and $q_x(\frac{z_1^*}{z_0^*}, \ldots, \frac{z_n^*}{z_0^*}) = 0$. Thus $v = z_{n+1}^*$ is always a lower bound for the L^1-norm of solution points obtained by varying z_0 above. It follows that the infimum of z_{n+1}^* (and thus the infimum of the L^1-norm) for all roots of p always equals the infimum of the optimal solution measure for x. □

Note that above the only variable we require to be different from 0 is the homogenization variable z_0. It would be interesting to see whether we can free ourselves from this restriction. The result as well holds for any other (computable) norm.

5 Conclusions

In the present paper we started the development of a structural analysis of approximation algorithms for real number optimization problems based on the BSS model of computation. Our goal was to set up a framework analogue to the study of combinatorial optimization problems in Turing complexity theory. We did so by introducing classes of real number optimization problems $NPO_{\mathbb{R}}$, $APX_{\mathbb{R}}$ and $PTAS_{\mathbb{R}}$ together with a completeness notion for these classes. It turned out that each of the above classes gives rise to consider four natural subclasses. For each of those we established the existence of complete problems. Given the many optimization problems being of immense interest it might be promising to further study the present approach in order to get a clearer view of the approximation properties of $NP_{\mathbb{R}}$-hard problems just as it is done since many years in combinatorial optimization.

References

1. G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, M. Protasi: Complexity and Approximation Springer (1999).
2. S. Basu, R. Pollack, M.-F. Roy: Algorithms in Real Algebraic Geometry. Springer (2003).
3. L. Blum, F. Cucker, M. Shub, S. Smale: Complexity and Real Computation. Springer (1998).
4. L. Blum, M. Shub, S. Smale: On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines. Bull. Amer. Math. Soc., vol. 21, 1–46 (1989).
5. P. Crescenzi, A. Panconesi: Completeness in Approximation Classes. Information and Computation 93, 241-262 (1991).
6. F. Cucker, M. Matamala: On digital nondeterminism. Mathematical Systems Theory 29, 635-647 (1996).
7. F. Cucker, J.M. Rojas (eds.): Foundations of Computational Mathematics, Festschrift on occasion of the 70th birthday of Steve Smale, World Scientific (2002).
8. T.Y. Li: Numerical solution of multivariate polynomial systems by homotopy continuation methods. Acta numerica, Cambridge University Press, vol. 6, 399–436 (1997).
9. K. Meer: On some relations between approximation and PCPs over the real numbers. To appear in: Theory of Computing Systems.
10. J. Renegar: On the Computational Complexity of Approximating Solutions for Real Algebraic Formulae. SIAM J. Comput. 21(6): 1008-1025 (1992).
11. J. Renegar, M. Shub, S. Smale (eds.): Mathematics of Numerical Analysis - Real Number Algorithms, Lecture Notes in Applied Mathematics vol. 32 (1996).

Physical Systems as Constructive Logics

Peter Hines*

York University, York YO10 5DD,
North Yorkshire, U.K.
peter.hines@cs.york.ac.uk

Abstract. This paper is an investigation of S. Wolfram's *Principle of Computational Equivalence'* – that (discrete) systems in the natural world should be thought of as performing computations. We take a logical approach, and demonstrate that under almost trivial (physically reasonable) assumptions, discrete evolving physical systems give a class of logical models. Moreover, these models are of *intuitionistic*, or *constructive* logics – that is, exactly those logics with a natural computational interpretation under the Curry-Howard 'proofs as programs' isomorphism.

1 Introduction

One of the more notable claims of [11] is that *physical systems* should naturally be thought of as *computational systems*. Although this claim is often backed up with reference to models of computation based on either cellular automata (as in [11]) or Turing machines [3], we address this claim from a *logical* perspective.

We consider a very broad class of physical systems evolving (in discrete steps) over time, and study them as though they are physical computers. We consider different descriptions of how they evolve over time, and introduce a partial ordering on this set of *machine evolutions*. Under assumptions about termination, this gives a familiar class of logical models – Heyting algebras. These are related to the *constructive* or *computational* logic known as intuitionistic logic, and play the same rôle for intuitionistic logic that Boolean algebras play for classical propositional logic.

2 Discrete Physical Systems

Our intuition of a discrete physical system is the following: we assume a set of *configurations* together with a rule $\mathcal{R}$ that describes how configurations change over time. The only assumptions we require are that configurations change in discrete steps (so we do not need to worry about Zeno's paradox [9]), and there are no 'hidden variables' — the current configuration unambiguously determines successive configurations.

* This research is part of QIP IRC www.qipirc.org (GR/S82176/01).

C.S. Calude et al. (Eds.): UC 2006, LNCS 4135, pp. 101–112, 2006.

This situation is easily formalised :

Definition 1. Abstract Computing Machines
An ACM, or **Abstract Computing Machine**, $\mathcal{M} = (X, \rightarrowtail)$ *is specified by a set* X *of* **configurations** *where every configuration* $x \in X$ *has at most one* **next configuration** y, *written* $x \rightarrowtail y$. *This is of course equivalent to the existence of a* partial *function* $Next : X \rightarrow X$. *The interpretation is that if an ACM is in configuration* x, *the next configuration (assuming this exists) it takes on is* $y = Next(x)$.

This definition, so far, is almost laughably simple. We merely have a (partial) function acting on a set, defining a binary relation via the usual representation of partial functions as relations. The interest arises in considering this partial function to be *iterated* — the intuition is that it describes a physical system with a simple rule for discrete evolution over time. The $Next$ function is (for example) the von Neumann architecture for a computer, the evolution rule for a cellular automata or Turing machine, the (discrete analogue of a) Hamiltonian for a physical system, or similar.

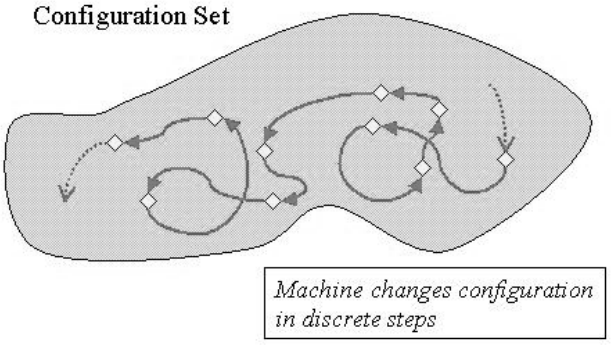


Fig. 1. An Abstract Computing Machine

Interpretation. A question of interpretation arises in that we allow for partiality in the $\rightarrowtail$ relation. Strictly, a physical system in a certain configuration will always have a next configuration under the application of a physically reasonable rule. We interpret a 'terminal configuration' (i.e. where $Next(x)$ is undefined) in one of two (related) ways:

- *Physical partiality* The next configuration is simply not accessible to us: the system evolves over time until it is outside of our domain of reference. The next configuration is beyond the scope of the observer — out of sight, perhaps.
- *A Halting Scheme* Partiality may also arise via the imposition of a **halting scheme**. Two possible examples are as follows:

– Given an ACM $(X, \rightarrowtail)$, we may specify a subset (or subspace, or whatever) of Halting configurations $H \subseteq X$, and restrict the next configuration relation to the complement of H. This gives a new relation $\rightarrowtail_H$, specified by

$$x \rightarrowtail_H y \quad \Leftrightarrow \quad x \rightarrowtail y \text{ and } x \notin H$$

– Alternatively, we may restrict the next configuration relation to situations where the configuration *changes* — ruling out 'halting configurations' where $x \rightarrowtail x \rightarrowtail x \rightarrowtail \ldots$

Examples. The definition of an Abstract Computing Machine is intentionally framed in as broad a manner as possible, to cover any discrete evolving physical system. We observe that those mentioned in discussions on the Principle of Computational Equivalence [11] — that is, digital computers, cellular automata, (discrete models of) weather systems, Turing machines, &c. — are covered by this definition.

We now analyse these systems using tools from theoretical computer science and logic, and demonstrate that even this simple definition gives rise to an interesting theory.

Definition 2. The 'Leads To' relation
Let $\mathcal{M} = (X, \rightarrowtail)$ be an ACM. We define the binary relation $\rightsquigarrow$ on the set X of configurations to be the transitive closure of the $\rightarrowtail$ relation. Hence, $\rightsquigarrow$ is the smallest binary relation satisfying :

– *$x \rightarrowtail y$ implies that $x \rightsquigarrow y$*
– *$a \rightsquigarrow b$ and $b \rightsquigarrow c$ implies that $a \rightsquigarrow c$*

We refer to this as the **leads to** *or* **subsequentness** *relation.*

Interpretation. The intuition of the 'leads to' relation is simple: a configuration describes the entire state of a computing machine, and $x \rightsquigarrow y$ simply means that if the machine is in configuration x it will, under whatever physical or logical rule $\mathcal{R}$ governs its evolution, be in configuration y at some later point.

We observe that the 'leads to' relation $\rightsquigarrow$ contains strictly less information about the evolution of an ACM than the 'next' relation, $\rightarrowtail$.

Proposition 1. *Let $\mathcal{M}_1 = (X, \rightarrowtail_1)$ and $\mathcal{M}_2 = (X, \rightarrowtail_2)$ be two distinct ACMs with the same configuration set, $X = \{a, b, c\}$, and with 'next' relations $\rightarrowtail_1$ and $\rightarrowtail_2$ specified by*

$$a \rightarrowtail_1 b \rightarrowtail_1 c \rightarrowtail_1 a$$

and

$$a \rightarrowtail_2 c \rightarrowtail_2 b \rightarrowtail_2 a$$

These two inequivalent ACMs give rise to the same subsequentness relation (the universal relation), where $x \rightsquigarrow x'$ for all $x, x' \in X$.

Proof. The proof of this is almost immediate, given the definition of $\rightsquigarrow$ as the transitive closure of $\rightarrowtail$. Of more interest is the observation that the subsequentness relation loses information about *causal ordering* — we cannot tell whether a is followed by b then c, or vice versa. □

Interpretations. Clearly, the subsequentness relation is important, computationally. However, the loss of information about the causal ordering is a non-trivial problem. There are three possibilities:

1. It is a simple observation that when two ACMs have the same subsequentness relation $\rightsquigarrow$ on a configuration set X, they may often be identified up to a permutation of X. However, we take a category-theorist's point of view that *isomorphism* should not be confused with *strict identity*, and note that the quotient induced by such a permutation will often identify all configurations — leading to trivial systems.
2. As a fairly 'blunt instrument' approach, we could introduce a *global clock* as an essential part of the structure of the ACM, together with the assumption that a single step $x \rightarrowtail y$ takes exactly one clock cycle. This would involve replacing the configuration space X with $X \times \mathbb{N}$ or $X \times \mathbb{Z}$, and replacing the transition $x \rightarrowtail y$ with the countable family of transitions $(x, n) \rightarrowtail (y, n+1)$. This will allow for the recovery of the causal ordering, albeit at the expense of a desperately expanded configuration set.
3. We could restrict the computational paths considered to those that *do not repeat configurations*. This prevents the sort of situation described in Proposition 1 from occurring, and allows causal orderings to be deduced from the subsequentness relation.

In what follows, we broadly follow 3. above, and restrict the computational paths considered. However, this is better done at the level of functions on the configuration set, rather than at the level of the configuration set itself. This also fits in with the category-theoretic approach that structure-preserving maps on a mathematical object are the correct level of discourse, rather than the elements of the mathematical object itself.

2.1 Evolutions and Semantics of Abstract Computing Machiness

Our intuition of an Abstract Computing Machine is that of a set X of configurations together with some rule $\mathcal{R}$ that describes – in a deterministic manner – how one configuration evolves into another.

In both physical and computational systems, we are often interested in studying systems at different levels of abstraction. Consider a *physical* computer, based on the *von Neumann architecture*, executing a *Java* program, using a *Java Virtual Machine*. We have a very different view of this according to whether we describe it at the level of machine language, interpreted byte code, high-level program code, or simply as a 'black box' that takes inputs to outputs.

In order to axiomatise this, we study the collection of partial functions on configuration sets that respect the 'leads to' relation, and introduce a partial ordering that corresponds to 'different levels of abstraction':

Definition 3. Evolutions, Machine Semantics, cycle-freeness
Let $\mathcal{M} = (X, \rightarrowtail)$ be an abstract computing machine. We define an **evolution** *of $\mathcal{M}$ to be a partial function $\eta : X \to X$ satisfying the following condition:*

$$\eta(x) = y \;\Rightarrow\; x \rightsquigarrow y \tag{1}$$

Of course, machine evolutions are far from unique — note that this definition even allows for the nowhere-defined partial function $0_X : X \to X$ as an evolution of $\mathcal{M}$. For an ACM $\mathcal{M}$, we define the **Machine Semantics** *of $\mathcal{M}$ to be the set of all evolutions, which we denote $[\mathcal{M}]$.*

When an evolution η of $\mathcal{M}$ satisfies the condition $\eta^K(x) \neq x$ for all $x \in X$ and $K \in \mathbb{N}^+$, we say that η is a **cycle-free evolution**. *We denote the set of cycle-free evolutions by $[\![\mathcal{M}]\!]$, and refer to this as the* **cycle-free semantics** *for $\mathcal{M}$. Note that nilpotent evolutions (i.e. evolutions $\eta \in [\mathcal{M}]$ where there exists some non-zero integer N such that $\eta^N = 0_X$) are always cycle-free; for finite configuration sets, the converse also holds. When the Next partial function is cycle-free it is clear that all machine evolutions are cycle-free. In this case, we say that $\mathcal{M}$ is a* **cycle-free ACM**.

Interpretation. Informally, an evolution is a partial function where, given a machine $\mathcal{M}$ in the configuration x, the machine will at some later point be in configuration $\eta(x)$ (provided $\eta(x)$ is defined). The term 'machine semantics' comes from the position that the meaning, or structure, of an ACM is best studied in terms of its set of evolutions.

We introduce a natural way of comparing cycle-free machine evolutions that has both a nice physical or computational interpretation, and well-behaved mathematical properties. The restriction to cycle-free evolutions is important, both in order to preserve the causal structure (as in Proposition 1), and in order to allow for the usual mathematical tools to be applied to the theory of machine semantics. Intuitively, we are restricting ourselves to considering computations that cannot 'get stuck in an infinite loop'.

Definition 4. The primitiveness relation
Let $\mathcal{M} = (X, \rightarrowtail)$ be an abstract computing machine, and let η, μ be cycle-free evolutions of $\mathcal{M}$, so $\eta, \mu \in [\![\mathcal{M}]\!]$. We say that η is **more primitive than** *μ, written $\mu \leq \eta$ exactly when, for all $\mu(c) = d$, there exists some integer $k \in \mathbb{N} = \{1, 2, \ldots\}$ such that $\mu(c) = d = \eta^k(c)$. (We emphasise that in this definition, k is not a fixed integer; it may vary as a function of both c, d and η, μ).*

Interpretation. The motivation for the primitiveness relation is, as stated above, that we wish to "study ACMs at different levels of generality". The primitiveness relation captures the informal idea that the description of a computer at the level of machine code is 'more primitive' than a description in terms of a compiled or interpreted language such as *Java*, which in turn is more fundamental than a description as a 'black box' that merely takes input to outputs.

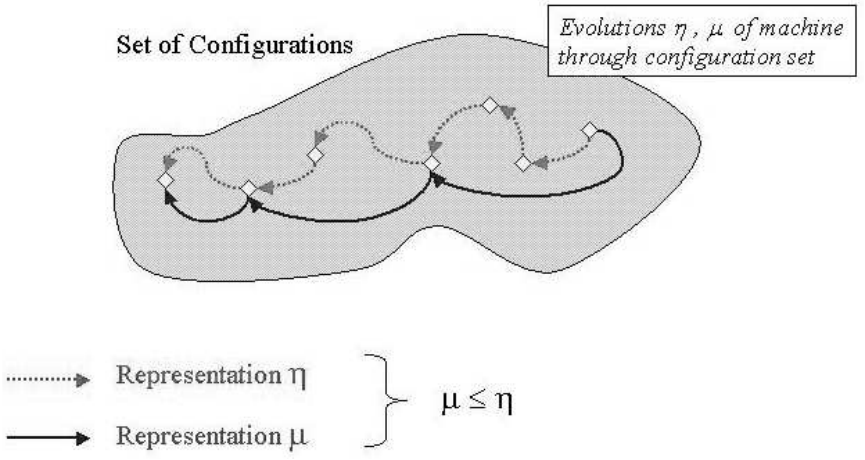


Fig. 2. The primitiveness partial order

3 The Mathematical Setting for the Primitiveness Relation

We demonstrate below (Theorem 1) that the correct mathematical setting for the primitiveness relation is in the theory of partial orders and (for cycle-free machines) lattices. The basics of this theory, as in [12] are as follows:

Definition 5. *Partial orders, Lattices, &c.*
A **partial order** *on a set P is a relation $\leq$ satisfying*

1. Reflexivity $a \leq a$ *for all* $a \in P$
2. Antisymmetry $a \leq b$ *and* $b \leq a$ *imples* $a = b$, *for all* $a, b \in P$
3. Transitivity $a \leq b$ *and* $b \leq c$ *implies* $a \leq c$, *for all* $a, b, c \in P$

A binary relation that is only required to satisfy 1. and 3. is called a **pre-order**, *instead of a partial order. Every set with a pre-order determines a partially ordered set by taking the minimal (order-preserving) equivalence classes determined by the congruence $x \sim y$ iff $x \leq y$ and $y \leq x$.*

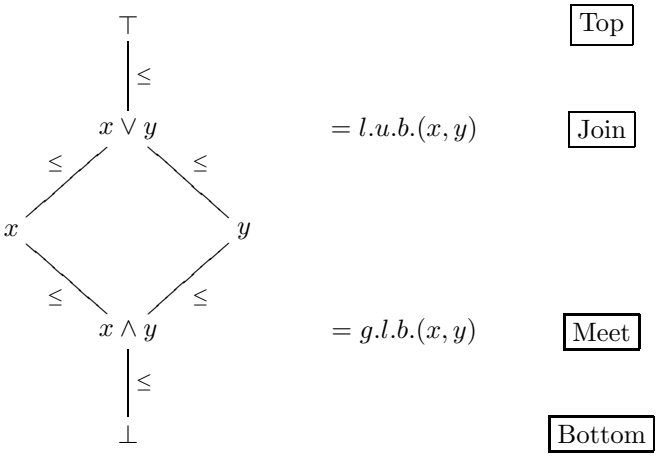
Given a pair of elements of a partially ordered set, $a, b \in P$, an **upper bound** *of a and b is an element z satisfying $a \leq z$ and $b \leq z$. The* **least upper bound** *or* **join** *(when it exists) of a and b, denoted $a \vee b$, is the unique upper bound satisfying $a \vee b \leq z$, for all upper bounds z of a and b.* **Lower bounds**, *and the* **greatest lower bound** *or* **meet** *(denoted $a \wedge b$), are defined dually.*

A **top element** *for a partially ordered set P is a unique element $\top \in P$ satisfying, $p \leq \top$, for all $p \in P$. Similarly, a bottom element $\bot \in P$ is a unique element satisfying $\bot \leq p$ for all $p \in P$.*

A **lattice** *is defined to be a partially ordered set where all pairs of elements (and hence by induction, all finite sets of elements) have both least upper bounds and greatest lower bounds. A lattice is called* **upper-complete** *when all sets of elements (i.e. not just finite sets) have a least upper bound. Similarly it is* **lower-complete** *when when all sets of elements have a greatest lower bound.*

By definition, all finite lattices are both upper-complete and lower-complete. Also, every upper-complete lattice has a top element given by $\top = \bigwedge L$ *and every lower-complete lattice has a bottom element given by* $\bot = \bigvee L$.

The term 'lattice' comes from the graphical representation for finite lattices as a Hasse Diagram, such as the following simple example:



Theorem 1. *Let* $\mathcal{M} = (X, \rightarrowtail)$ *be an ACM, with machine semantics* $[\mathcal{M}]$*, and cycle-free semantics* $[\![\mathcal{M}]\!] \subseteq [\mathcal{M}]$*. Then* $([\![\mathcal{M}]\!], \leq)$ *is a partial order, with* $0_X \in [\![\mathcal{M}]\!]$ *as bottom element.*

Proof.
To show that $([\![\mathcal{M}]\!], \leq)$ is a partial order, we need to demonstrate *reflexivity*, *anti-symmetry* and *associativity*:

- *Reflexivity* This is immediate from the definition : for all $\eta(c) = d$, trivially, we have that $\eta^K(c) = d$, where $K = 1$, and so $\eta \leq \eta$.
- *Anti-symmetry* Given $\eta \leq \mu$ and $\mu \leq \eta$, then for all $\eta(c) = d$, there exists $K > 0$ such that $\mu^K(c) = d$. Therefore, $dom(\eta) \subseteq dom(\mu)$. Similarly, $dom(\mu) \subseteq dom(\eta)$ and so $dom(\mu) = dom(\eta)$.
  We now we prove by contradiction that $\eta = \mu$. Given $d = \eta(c) = \mu^K(c)$, assume that $K \neq 1$. We then define

$$\mu(c) = e_1 \ , \ \mu^2(c) = e_2 \ , \ \ldots \ , \ \eta^k(c) = e_k = d$$

  As $\mu \leq \eta$, there exists $L_1, \ldots, \text{Ł}_K > 0$ such that

$$e_1 = \eta^{L_1}(c) \ , \ e_2 = \eta^{L_2}(e_1) \ , \ \ldots \ , \ e_K = d = \eta^{L_K}(e_{k-1})$$

This gives

$$\eta(c) = \eta^{L_k+L_{K-1}+\ldots+L_1}(c) \quad \text{where} \sum_{j=1}^{K} L_j \; > 1$$

a contradiction of the cycle-freeness of η. Therefore $K = 1$, and so $\eta(c) = \mu(c)$ for all $c \in dom(\eta) = dom(\mu)$.

– *Transitivity* Given $\eta \leq \mu$ and $\mu \leq \zeta$, then, for all $\eta(c) = d$, there exists $K \in \mathbb{N}^+$ such that $\eta(c) = d = \mu^K(c)$. Similarly, for all $\mu(c) = e$, there exists $L \in \mathbb{N}^+$ such that $\zeta^L(c) = e$. Therefore, for all $\eta(c) = d$, we deduce that there exists some finite series of positive integers $\{L_1, L_2, \ldots L_K\}$ such that $\zeta^{L_1+L_2+\ldots+L_K}(c) = d$. Therefore $\eta \leq \zeta$, and our result follows.

It is also trivial from the definition that the nowhere-defined partial function 0_X satisfies $0_X \leq \eta$ for all $\eta \in [\![\mathcal{M}]\!]$, and so is a bottom element for this partial ordering. □

4 The Order Theory of Cycle-Free Abstract Machines

As an indication that we may find interesting computational structures within the theory of evolutions and the primitiveness relation, we now analyse the cycle-free case in logical, or computational, terms. This is clearly the 'simplest possible' case — we briefly discuss the general case in Section 5. The intention in presenting this case is to show that we may derive non-trivial theories even from the simplest case of ACMs.

Interpretations. We are now restricting ourselves to the theory of cycle-free ACMs. The definition of a cycle-free ACM is that the $Next : X \to X$ partial function is cycle-free (and hence, trivially, all machine evolutions are cycle-free). In the finite case, this is exactly equivalent to the condition that $Next : X \to X$ is *nilpotent*. The intuition of this is not that we are considering computer programs without loops or control structure — rather, (in the finite case) we are considering programs with guaranteed *termination*, or *halting*. Although this is often difficult or impossible to prove, we refer to [5] for an decidedly non-trivial example of a computational system based on provably nilpotent maps.

Theorem 2. *Let $\mathcal{M} = (X, \rightarrowtail)$ be a cycle-free ACM – i.e. $Next(x) \neq x$, for all $x \in X$, and all evolutions are cycle-free, so $[\mathcal{M}] = [\![\mathcal{M}]\!]$. Then*
(i) *The partial order $([\mathcal{M}], \leq)$ has a top element.*
(ii) *$([\mathcal{M}], \leq)$ is closed under finite meets.*
(iii) *$([\mathcal{M}], \leq)$ is closed under arbitrary joins.*

Proof.
(i) The top element is quite simply the $Next : X \to X$ partial function. By definition of machine evolutions, $\eta(c) = d$ implies that $Next^N(c) = d$ for some $N > 0$. Also, as $\mathcal{M}$ is cycle-free, there does not exist any machine evolution $\zeta \in [\mathcal{M}]$ such that $\zeta^K(c) = Next(c)$ for $K > 1$, so $Next$ is the top element

of this partial order. For the remainder of this proof, we use lattice-theoretic notation, and write $\top(c)$ for $Next(c)$.
(ii) Having established the existence of a top element, we explicitly exhibit the join of a set of elements $\{\eta_i\}_{i\in I}$, again in terms of the top element. Given a configuration $x \in X$, then:

- $\left(\bigvee_{i\in I} \eta_i\right)(x)$ is undefined exactly when $\eta_i(x)$ is undefined for all $i \in I$.
- Consider the subset $\{\eta_j\}_{j\in J\subseteq I}$ of evolutions where $\eta_j(x)$ is defined. For all η_j, there exists some minimal n_j such that $\eta_j(x) = \top^{n_j}(x)$. We then define an integer n in terms of the greatest common divisor, $n = gcd(\{n_j\}_{j\in J})$ and take $\left(\bigvee_{i\in I} \eta_i\right)(x) = \top^n(x)$.

It is then immediate by the definition of the primitiveness ordering and the top element that this is the least upper bound of $\{\eta_i\}_{i\in I}$.
(iii) We explicitly exhibit the meet of two elements η, μ, again in terms of the top element:

- Given $x \in X$, then $(\eta \wedge \mu)(x)$ is *undefined* when either $\eta(x)$ is undefined or $\mu(x)$ is undefined, or both.
- Assuming that both $\eta(x)$ and $\mu(x)$ exist, then by (i) above, there exist least integers p, q such that $\top^p(x) = \eta(x)$, and $\top^q = \mu(x)$ (note that p and q are not fixed, but vary with x). The meet of η and μ is defined in terms of the least common multiple of p, and q, by $(\eta \wedge \mu)(x) = \top^r(x)$ where $r = lcm(p, q)$.

It is again immediate, by the definition of the primitiveness ordering, that this is the greatest lower bound of η and μ. □

4.1 Cycle-Free Abstract Machines Describe Constructive Logics

We now observe that there is a close connection between partially ordered sets (with additional 'closure' properties), and both theoretical computer science and logic. We refer to [1,6] for good expositions, and restrict the following exposition to one example that is relevant to the theory of abstract computing machines — that of Heyting algebras.

Heyting algebras play the same rôle for intuitionistic logic that Boolean algebras play for classical logic. Unfortunately, there is no space here to present an overview of intuitionistic logic, so we refer to [4] for a good introduction. Very broadly, intuitionistic logic is a restriction of classical logic that exactly captures the constructive fragment of logic (the original intention was that a proof of the existence of a mathematical object is exactly a construction of that object). Because of this constructive aspect, it is perhaps most familiar as the logic used in the PROLOG programming language [7] — it is also the logic required for the "Proofs - as Programs" correspondence given by the Curry-Howard isomorphism [10] between (intuitionistic) logics and typed lambda calculii.

Definition 6. Heyting algebras, (relative) pseudocomplements
A **Heyting algebra** *is a lattice L with (distinct) top and bottom elements, where for every pair of elements g, f the set $\{h : f \wedge h \leq g\} \subseteq L$ is bounded above. The (unique) upper bound of this set is called the* **relative pseudocomplement** *of f with respect to g, and denoted $f \Rightarrow g$. Clearly, the relative pseudocomplement of f with respect to g is used to model (intuitionistic) implication. There is also a similar concept for (intuitionistic) negation:*

The **pseudocomplement** *of $f \in L$, denoted $\neg f$ is defined in terms of the bottom element of the lattice, as $\neg f = f \Rightarrow \bot$.*

The precise connection between intuitionistic logic and Heyting algebras is then the following:

Definition 7. Logical equivalence, the Lindenbaum-Tarski algebra
Consider a logical theory $\mathcal{T}$, consisting of formulæ, connectives, and rules of implication. Formulæ p, q are called **logically equivalent**, *when p can be deduced from q and q can be deduced from p. We denote this equivalence relation by $p \sim q$.*

The **Lindenbaum-Tarski algebra** *of T has as elements equivalence classes of formulæ under this equivalence relation. The operations of the Lindenbaum-Tarski algebra are those inherited from the logical theory $\mathcal{T}$ (such as conjunction, disjunction, negation, &c.), provided they are well-defined under this quotient (— which is a reasonable assumption for logical theories).*

Although this is a very general concept, for intuitionistic logics there is a good characterisation of the corresponding Lindenbaum-Tarski algebras:

Proposition 2. *Heyting algebras exactly the Lindenbaum-Tarski algebras of intuitionistic logics.*

Proof. This is a standard result of mathematical logic — we refer to [2] for a good exposition. □

We now demonstrate that machine semantics for Abstract Computing Machines are Heyting algebras, and hence by Proposition 2, are intuitionistic logics, with logically equivalent terms identified.

Proposition 3. *For an arbitrary cycle-free ACM $\mathcal{M}$, the primitiveness partial order gives a Heyting Algebra structure to the machine semantics.*

Proof. We have seen that the machine semantics for an ACM has (distinct) top and bottom elements, is closed under finite meets and arbitrary joins. The existence of the relative pseudocomplement follows almost trivially from the closure under arbitrary joins, by

$$f \Rightarrow g \;=\; \bigvee \{h \;:\; f \wedge h \;\leq\; g\} \tag{2}$$

□

Corollary 1. *Let $\mathcal{M}$ be a cycle-free abstract computing machine. Then the machine semantics of $\mathcal{M}$ is a model of an intuitionistic logic, under the logical equivalence relation.*

5 Commentary

We have only scratched the surface of the theory of Abstract Computing Machines, and machine semantics. Notably, all substantial results have been about the very special case of cycle-free machines. The following comments are more speculative, but hopefully provide more intuition:

Mathematically: For an ACM $\mathcal{M}$, the lattice of machine evolutions $[\mathcal{M}]$ is closed under composition, and hence is a *semigroup*. However, in general, the set of cycle-free evolutions is *not* closed under composition. Even for cycle-free machines, the composition is not order-preserving (since $a \leq b$ does *not* imply that $ax \leq bx$, so $[\mathcal{M}]$ is not – for example– a quantale). The interaction of the composition and the partial ordering is a non-trivial subject and presumably related to the domain-theoretic notion of 'computation by the calculation of least fixed points'. We also observe that the order-theory of arbitrary ACMs remains to be studied. Although it is relatively easy to show that we may recover domain-theoretic notions such as directed-completeness, compactness, &c. (we refer to [1] for a good exposition of these notions) the full theory has not been given.

Physically: Although the definition of Abstract Computing Machines was framed very widely, there were nevertheless certain underlying assumptions. We may think of the partial functions (i.e. the machine evolutions) as describing possible *observations* of an evolving system. This is the underlying assumption that observing a system is not a physical change to it; similarly, the use of partiality is an implicit acceptance that either the halting scheme used is physically reasonable, or our mathematical tools are appropriate to analyse a system where we do not have access to all possible configurations. These assumptions (any many others) will need to be considered in more detail if we try to analyse quantum-mechanical systems in a similar way.

Computationally. It is interesting to observe that the cycle-free machines (i.e. those that guarantee not to enter closed loops – unconditional termination in the finite case) are exactly those that provide models of intuitionistic logic (also very closely related to unconditional termination via Robinson's unification Algorithm [7] and various decidability results [4]). In the general case, the set of cycle-free evolutions is not a lattice, because it has no top element (the $Next : X \to X$ partial function is *not* cycle-free). Preliminary studies suggest that the correct setting is domain theory, and computational interpretations may be found in areas such as models of functional programming or untyped lambda calculus [1,8].

The Principle of Computational Equivalence? No claim has been regarding Abstract Computing Machines and computational universality. However, there are a number of approaches to computationally universal systems via order theory; undoubtedly the most famous of these is Dana Scott's semantics for the

pure untyped lambda calculus [8]. As for interpretations, or whether this paper supports Wolfram's principle, the computationally significant structure in our theory arises from comparing distinct ways of observing (subproperties of) an evolving physical system. One possible conclusion is that if we are free to look at (subsections of) a physical system in any way we wish, we can quite easily see significant computational structures. However, the actual computational structure arises from our perceptions of evolving systems, rather than being intrinsic to the systems themselves.

References

1. S. Abramsky, A. Jung: Domain Theory, in *S. Abramsky, D.M. Gabbay, T.S.E. Maibaum, editors, Handbook of Logic in Computer Science. III* Oxford University Press, 1994.
2. F. Borceux: Handbook of Categorical Algebra 3, In *Encyclopedia of Mathematics and its Applications, Vol. 53*, Cambridge University Press, 1994.
3. L. Bunimovich: Many Dimensional Lorentz Cellular Automata and Turing Machines, *Int Jour of Bifurcation and Chaos 6* (1996), 1127–1135.
4. M. Dummett: *Elements of Intuitionism*, Oxford University Press, 2000.
5. J.-Y. Girard: Geometry of Interaction 1, *Proceedings Logic Colloquium '88*, North-Holland (1989), 221–260.
6. P. Johnstone: The point of pointless topology, *Bulletin American Mathematical Society*, 8(1) (1983), 41–53.
7. J. Robinson: A machine-oriented logic based on the resolution principle, *Communications of the ACM*, 5 (1965), 23–41.
8. D. Scott: Outline of a mathematical theory of computation, In *4th Annual Princeton Conference on Information Sciences and Systems*, (1970), 169-176.
9. Z. Silagadze: Zeno meets modern Science, *Acta Phys. Polon.*, B36 (2005), 2886–2930.
10. P. Urzyczyn, M. Sorensen: *Lectures on the Curry-Howard Isomorphism* Elsevier, 2006.
11. S. Wolfram: *A New Kind of Science*, Wolfram Media, 2002.
12. S. Vickers: *Topology via Logic*, Cambridge Tracts in Theoretical Computer Science, 5 (1998).

On Spiking Neural P Systems and Partially Blind Counter Machines*

Oscar H. Ibarra[1], Sara Woodworth[1], Fang Yu[1], and Andrei Păun[2]

[1] Department of Computer Science
University of California, Santa Barbara, CA 93106, USA
{ibarra, swood, yuf}@cs.ucsb.edu
[2] Department of Computer Science/IfM
Louisiana Tech University, Ruston, LA 71272, USA
apaun@latech.edu

Abstract. A k-output spiking neural P system (SNP) with output neurons, $O_1, ..., O_k$, generates a tuple $(n_1, ..., n_k)$ of positive integers if, starting from the initial configuration, there is a sequence of steps such that during the computation, each O_i generates exactly two spikes a a (the times the pair a a are generated may be different for different output neurons) and the time interval between the first a and the second a is n_i. After the output neurons generate their pairs of spikes, the system eventually halts. We give characterizations of sets definable by partially blind multicounter machines in terms of k-output SNPs operating in a sequential mode. Slight variations of the models make them universal.

1 Introduction

Neurons are arguably one of the most interesting cell-types in the human body. A large number of neurons working in a cooperative manner are able to perform tasks that are not yet matched by the tools we can build with our current technology. Some such tasks are thought, self-awareness, intuition, etc. Coming closer to computer science, even "simple" tasks that have been studied extensively, such as pattern matching, are performed much faster and more reliably by our brains using the "technology" of neurons than our computers which are several orders of magnitude faster in their information processing capabilities.

We believe the distributed manner in which the brain processes information is important in obtaining better performance, thus we are interested in the emerging area of Spiking Neural P systems (SNPs) defined as a computational model in [4], and investigated in a series of papers:[9], [10], [3]. SNPs incorporate ideas from spiking neurons into membrane computing [8], see, e.g., [1], [5], [6].

We would like to stress that the current work is not intended as a straight simulation of neurons as many features/details of neurons are omitted/abstracted.

* The research of O. H. Ibarra and S. Woodworth was supported in part by NSF Grants CCF-0430945 and CCF-0524136. The research of A. Păun was supported in part by NSF Grant CCF-0523572.

C.S. Calude et al. (Eds.): UC 2006, LNCS 4135, pp. 113–129, 2006.

For example, the different elements of neurons such as soma, dendrite, axon, axon terminal, Schwann cells etc. are abstracted. We also do not consider the inhibitory synapses in a direct manner, but rather model them indirectly by 'forgetting rules' (defined below). It is also worth mentioning that we consider the spikes of the neurons to be of the same type, thus the actual changes in the electric potential of spikes are assumed to encode no information. Finally, in the basic model, the output of such a system of neurons is considered to be the time elapsed between the only two spikes of the output neuron (one specific, pre-set neuron). The reason behind this choice was two-fold: first we want to have a model that is consistent (all the components have the same types of rules), thus the output neuron is a "regular" neuron in all respects. Second, this definition is very close to bio-implementation as one could observe the specific output neuron, detecting the two spikes, and determining the time elapsed between the spikes. It is our hope that once technology gives way and we are able to "manipulate" neurons, the constructions presented in this paper could see a practical implementation.

We now pass to a more detailed description of the SNP; such a system is represented as a directed graph consisting of a set of neurons (nodes of a graph) connected by synapses (directed edges of the graph). The neurons send signals (spikes) along these synapses by means of firing rules, which are of the form $E/a^c \to a; t$, where E is a regular expression, c is the number of spikes consumed by the rule that spikes a single a, and t is the delay between firing the rule and emitting the spike. A rule can only be used if the number of spikes in the neuron are "covered" by expression E, in the sense that the current number of spikes in the neuron, n, is such that a^n is contained in the set $L(E)$. In the time interval between firing a rule and emitting the spike, the neuron is closed/blocked – it does not receive other spikes and cannot fire. After the time interval, the neuron is again open and can again fire and receive other spikes. There are also rules for forgetting spikes, of the form $a^s \to \lambda$ (s spikes are just removed from the neuron). *In this paper, for convenience, we will also refer to the forgetting rules as firing rules.* Starting from a fixed initial distribution of spikes in the neurons (initial configuration) and using the rules in a synchronized manner (a global clock is assumed), the system evolves. A computation is a sequence of transitions starting from the initial configuration. A transition is maximally parallel in the sense that all neurons that are fireable must fire. However, in any neuron, at most one rule is allowed to fire. Details can be found in [4].

An SNP can be used as a computing device in various ways. Here, as in previous papers, we will use them as generators of numbers. We will only consider SNPs with three types of neurons:

1. A neuron is *bounded* if every rule in the neuron is of the form $a^i/a^j \to a; t$, where $j \leq i$, or of the form $a^k \to \lambda$, provided there is no rule of the form $a^k/a^j \to a; t$ in the neuron. Note that there can be several such rules in the neuron. These rules are called *bounded rules*. (For notational convenience, we will write $a^i/a^i \to a; t$ simply as $a^i \to a; t$.)

2. A neuron is *unbounded* if every rule in the neuron is of the form $a^3(a^2)^*/a^2 \rightarrow a;t$, $a^3(a^2)^*/a^3 \rightarrow a;t$, or $a(a)^*/a \rightarrow a;t$. (Again, there can be several such rules in the neuron.) These rules are called *unbounded rules*.
3. A neuron is *general* if it can have *general rules*, i.e., bounded as well as unbounded rules.

An SNP is bounded if all the neurons in the system are bounded. If, in addition, there are unbounded neurons then the SNP is said to be unbounded. A general SNP has general neurons.

It was recently shown in [3] that a set $Q(\Pi) \subseteq N^1$ is recursively enumerable if and only if it can be generated by a 1-output unbounded SNP Π all of whose unbounded neurons have only one rule – either $a(a)^*/a \rightarrow a;0$ or $a(a)^*/a \rightarrow a;2$.

We generalize the SNP by allowing it to produce k outputs. A *k-output SNP Π* has k output neurons, $O_1, ..., O_k$. We say that Π generates a k-tuple $(n_1, ..., n_k) \in N^k$ if, starting from the initial configuration, there is a sequence of steps such that each output neuron O_i generates exactly two spikes a a (the times the pair a a are generated may be different for different output neurons) and the time interval between the first a and the second a is n_i. Moreover, after all the output neurons have generated their pair of spikes, the system eventually *halts*, in the following sense:

> *Π halts* if it reaches a configuration where all neurons are open but no neurons are fireable. In fact, for the constructions in this paper, this will correspond to the configuration in which all neurons, except for a specified subset R of neurons, have zero spikes, and those in R have exactly two spikes.

The set of all k-tuples generated is denoted by $Q(\Pi)$.

In this paper, we study SNPs operating in sequential mode. Informally, this means that at every step of the computation, if there is at least one neuron with at least one rule that is fireable, we only allow one such neuron and one such rule (both nondeterministically chosen) to be fired. There are two interesting cases:

1. *Case 1*: At every step, there is at least one neuron with a fireable rule. We show that:
   (a) A k-output sequential unbounded SNP Π can generate a set $Q(\Pi) \subseteq N^k$ if and only if it can be generated by a partially blind multicounter machine (PBCM). Since PBCMs are not universal [2], it follows that these SNPs are not universal.
   (b) In contrast to 1(a), sequential general SNPs are universal.
2. *Case 2*: Not every step has at least one neuron with a fireable rule. (Thus, the system might be dormant until a rule becomes fireable. However, the clock will keep on ticking.) In contrast to 1(a), we show that sequential unbounded SNPs are universal.

2 Sequential Spiking Neural P Systems

We will investigate the computational power of SNPs whose behavior is controlled to operate in a sequential manner (i.e. asynchronously). More precisely, the SNP is restricted in its operation as follows:

1. As before, the system starts from a fixed initial configuration and is synchronized, i.e., there is a global clock (so all the neurons use this clock).
2. However, a step consists of nondeterministically choosing a "fireable" rule. (For convenience, a forgetting rule is classified as a "fireable" rule.) If there is no fireable rule, then the system is dormant until a rule becomes fireable. However, the clock will keep on ticking.
3. The convention for halting is like before, i.e., all neurons, except for a specified subset R of neurons, have zero spikes and those in R have two spikes (and, of course, all the neurons are open, but none are fireable).

For convenience we will refer to SNPs operating in a sequential mode as *sequential SNPs.*

2.1 Strongly Sequential SNPs and Partially Blind Counter Machines

We will give a characterization of partially blind multicounter machines. But first we recall the definition of a multicounter machine[7] .

A nondeterministic multicounter machine (CM) $\mathcal{M}$ is a nondeterministic finite automaton with a finite number of counters (it has no input tape). Each counter can only hold a nonnegative integer. The machine starts in a fixed initial state with all counters zero. During the computation, each counter can be incremented by 1, decremented by 1, or tested for zero. A distinguished set of k counters (for some $k \geq 1$) is designated as the output counters. The output counters are nondecreasing (i.e., cannot be decremented). A k-tuple $(n_1, ..., n_k) \in N^k$ is generated if $\mathcal{M}$ eventually halts in an accepting state, all non-output counters zero, and the contents of the output counters are $n_1, ..., n_k$, respectively. We will refer to a CM with k output counters (the other counters are auxiliary counters) as a k-output CM.

A *partially blind k-output* CM (k-output PBCM) [2] is a k-output CM, where the counters cannot be tested for zero. (Again the output counters are nondecreasing.) The counters can be incremented by 1 or decremented by 1, but if there is an attempt to decrement a zero counter, the computation aborts (i.e., the computation becomes invalid). Note that, as usual, the output counters are nondecreasing. Again, by definition, a successful generation of a k-tuple requires that the machine enters an accepting state with all non-output counters zero.

It is known that partially blind k-output CMs can be simulated by vector addition systems, and vice-versa [2]. (Hence, such counter machines are not universal.) In particular, a partially blind k-output CM can generate the reachability set of a vector addition system. We can characterize partially blind k-output multicounter machines (PBCMs) in terms of "strongly sequential unbounded" SNPs.

> A *strongly sequential unbounded SNP* is a sequential unbounded SNP which is further restricted in its operation in that an accepting computation is valid only if *every step of the computation has at least one fireable rule.* Otherwise (i.e., there is a step in which there is no fireable rule), the computation is viewed as invalid and no output from such a computation is included in the generated set.

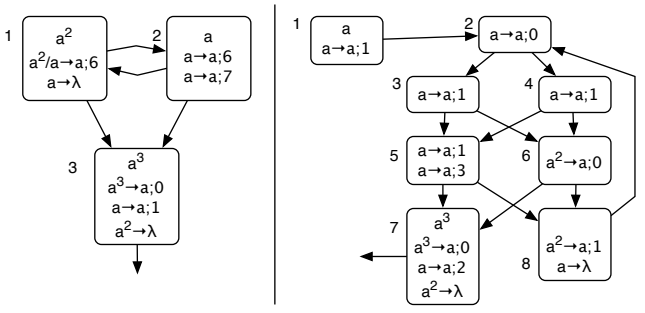


Fig. 1. SNP (Left) and Strongly Sequential SNP (Right) Generating $\{7n+1 \mid n \geq 1\}$

To illustrate how a strongly sequential SNP operates, in Figure 1 we give an example of a SNP generating $Q = \{7n+1 \mid n \geq 1\}$ along with a strongly sequential SNP generating the same set. The SNP operates by having all three neurons fire in parallel during the first time step sending a spike to the environment beginning the output. Neuron 2 nondeterministically picks to fire one of its rules. If the first rule is chosen, the system will repeat. If the second rule is chosen, the system will trigger neuron 3 to fire sending a second spike to the environment at $7n+1$ steps after the first spike (where n is the number of times the system looped plus 1).

The operation of the strongly sequential SNP is more complicated. A valid computation for this system fires the neurons in the following sequential manner: $1, 2, ((3,4)+(4,3)), (5,6,8,7,2,((3,4)+(4,3)))^*, 5, 6, 7, 8, 8$, where $((3,4)+(4,3))$ is interpreted to mean that either 3 is followed by 4 or 4 is followed by 3. Changing the firing order of other neurons creates an invalid computation by forcing the system into a non-halting configuration where no neuron is fireable.

We now consider the case when each neuron is either bounded or unbounded. In fact, as we shall see, we may assume that in an unbounded neuron, the only rule present is $a^3(a^2)^*/a^3 \to a; 0$.

Lemma 1. *If $Q(\Pi) \subseteq N^k$ is generated by a k-output strongly sequential unbounded SNP Π, then it can be generated by a k-output PBCM $\mathcal{M}$. Hence, such SNPs are not universal.*

Proof. Given a k-output strongly sequential unbounded SNP Π, we construct a k-output PBCM $\mathcal{M}$ to simulate Π. $\mathcal{M}$ has several counters, which are initially zero. There are k output counters, $G_1, ..., G_k$. For $1 \leq i \leq k$, when output neuron O_i of the Π has generated the first spike a, output counter G_i starts incrementing at every step of the simulation. When O_i has generated the second spike a, G_i stops incrementing. When all the output counters have stopped incrementing, $\mathcal{M}$ enters the *Ending Phase*, which we will describe below.

We describe how the other counters of M are used to keep track of the numbers of spikes in the neurons during the computation. A bounded neuron is easy to simulate and does not need a counter. Since the regular expression in a bounded

neuron represents a finite set, the simulation of the neuron can be done in the finite control.

Consider an unbounded neuron. First consider the case when the only rule in the unbounded neuron is $a^3(a^2)^*/a^3 \rightarrow a; 0$. (In fact, as we shall see, in the converse of this lemma, only this rule is used in every unbounded neuron.) To simulate this unbounded neuron, $\mathcal{M}$ uses a partially blind counter C to keep track of the number of spikes of this neuron during the computation. Let m be the initial number of spikes in this neuron. At the start of the simulation, $\mathcal{M}$ increments C (which starts at zero) to value m. During the simulation, the finite-state control of M will keep track of the parity of C. If C is even, the neuron cannot fire. If C is odd (we don't know whether it is $1, 3, 5, ...$) and the neuron is "open", then when we fire the neuron, we decrement C by 3.

1. If C had value 1, the machine aborts.
2. If C had value $2i + 1$ $(i = 1, 2, ...)$, then the the new value of C (after decrementing) is $2(i-1)$, which is even. The next time it will be odd is when a spike comes into the neuron, and the counter value will then be $2(i-1)+1$. If $i = 2, ...$, then the neuron is fireable. If $i = 1$, then when we try to fire, the machine will abort.

Thus, M can simulate the rule $a^3(a^2)^*/a^3 \rightarrow a; 0$. In general, if the neuron has other rules of the forms $a^3(a^2)^*/a^2 \rightarrow a; t$, $a^3(a^2)^*/a^3 \rightarrow a; t$, or $a(a)^*/a \rightarrow a; t$, a new counter is needed to simulate each such rule. Thus, if there are s different unbounded rules in the neuron, $\mathcal{M}$ needs s counters to simulate the neuron.

Ending Phase: This phase is entered only when the k output counters have k values generated by the k output neurons of the SNP. $\mathcal{M}$ continues the simulation until at some time, nondeterministically chosen, it guesses that Π has halted. By definition, this happens when all the neurons, except for a specified subset R of neurons, have zero spike and those in R have two spikes. $\mathcal{M}$ then decrements by 2 the counters corresponding to the neurons in R and enters an accepting state. It follows that $\mathcal{M}$ generates $Q(\Pi)$. □

Lemma 2. *Let $Q(\mathcal{M}) \subseteq N^k$ be generated by a k-output PBCM $\mathcal{M}$. Then we can construct a k-output strongly sequential unbounded SNP $\Pi_{11}^{(o_1,\cdots,o_k)}$ which generates the set $Q(\Pi_{11}^{(o_1,\cdots,o_k)}) = \{(11x_1 - o_1, \cdots, 11x_k - o_k) \mid (x_1, \cdots, x_k) \in Q(\mathcal{M}), x_i \geq 2$ for $1 \leq i \leq k\}$ for some set tuple $(o_1, \cdots, o_k)$ where $0 \leq o_i \leq 10$.*

Proof. Given a k-output PBCM $\mathcal{M}$, we construct a k-output strongly sequential unbounded SNP $\Pi_{11}^{(o_1,\cdots,o_k)}$ for some tuple $(o_1, \cdots, o_k)$ where $0 \leq o_i \leq 10$. $\Pi_{11}^{(o_1,\cdots,o_k)}$ simulates $\mathcal{M}$ by simulating each of $\mathcal{M}$'s instructions with a strongly sequential unbounded SNP module. The instructions of $\mathcal{M}$ are of the forms $l_i = (ADD(r_n), l_j, l_k)$, $l_i = (SUB(r_n), l_j)$, and $l_i = HALT$. We assume that $\mathcal{M}$'s output counters (counters $r_1, \cdots, r_k$) are never decreasing. In the simulation, we use a single neuron to represent each counter which stores some count x as $2x$ spikes. The initial configuration of the system has the initial configuration of each module along with a single spike in neuron l_0 (representing the initiating

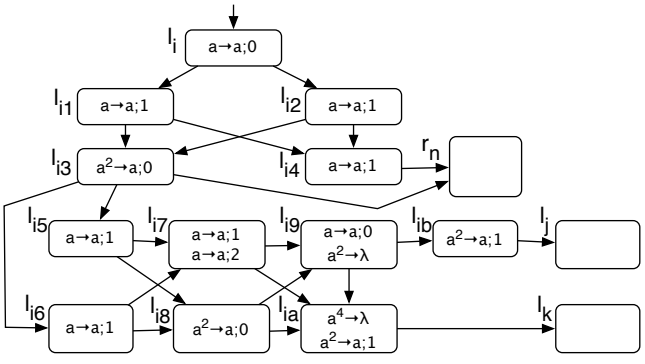


Fig. 2. Strongly Sequential Unbounded SNP Addition Module

neuron for instruction l_0). When the computation halts, a correct computation will leave the system with all neurons empty except neurons $r_1, \cdots, r_k$ which each contains two spikes. Due to the sequential nature of the machine, the output module requires 11 steps to decrement each neuron $r_{k'}$ for $1 \leq k' \leq k$ and determine if this is an initial or final decrement. Hence the numbers generated by $\Pi_{11}^{(o_1,\cdots,o_k)}$ are eleven times the numbers generated by $\mathcal{M}$ (minus some offset value of ten or less). Also due to the nature of the output module we will describe later in the proof, the strongly sequential nature of the system only holds if each tuple $x_{k'}$ for $1 \leq k' \leq k$ is ≥ 2.

To simulate each ADD instruction we create the module shown in Figure 2. The module is initiated by a single spike to neuron l_i which fires sending a spike to both neurons l_{i1} and l_{i2} (during time $t+1$ where time t is the time the initiating spike is sent to neuron l_i). Since the system is sequential, one of these two neurons fires during the next step (time $t+2$) and the other neuron fires in the following step (time $t+3$) with both sending a spike to neurons l_{i3} and l_{i4}. (Since both rules have a delay of a single step, neurons l_{i3} and l_{i4} are not fireable until both neurons l_{i1} and l_{i2} fire.) At time $t+4$, neuron l_{i4} spikes with a delay of 1. (Since the neuron is closed at time $t+4$, the second spike sent to neuron l_{i4} is lost.) Neuron l_{i3} spikes and fires at time $t+5$ so neuron r_n receives two spikes during one step. This increments the value of counter r_n by one.

When neuron l_{i3} fires, a spike is also sent to neurons l_{i5} and l_{i6}. Now one of these two neurons is chosen nondeterministically to spike during time $t+6$ and the other neuron spikes during time $t+7$. (Again, this is guaranteed because of the delay associated with both rules.) These spikes guarantee that neuron l_{i7} will spike at time $t+8$ (nondeterministically applying either of its rules) and neuron l_{i8} will spike at time $t+9$. If neuron l_{i7}'s second rule is applied, neurons l_{i9} and l_{ia} both receive a single spike at time $t+9$ (from neuron l_{i8}). This causes l_{i9} to spike at time $t+10$ which is also the time step where neuron l_{i7} fires sending spikes to neurons l_{i9} and l_{ia}. (So in the previous step neuron l_{ia} contained one spike and now after one step it contains three spikes.) Neuron l_{i9} will again spike at time $t+11$ causing both neuron l_{ia} (with 4 spikes) and neuron l_{ib} (with 2

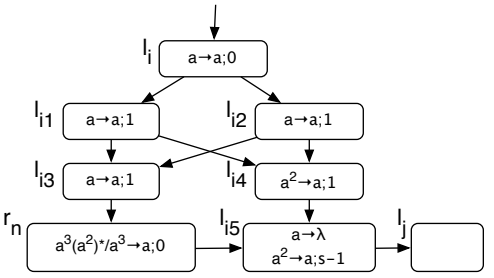


Fig. 3. Strongly Sequential Unbounded SNP Subtraction Module

spikes) to become fireable. If neuron l_{ia} forgets at time $t + 12$ (causing neuron l_{ib} to fire at time $t + 13$) we will not have a fireable rule at time $t + 14$ creating an invalid computation. If neuron l_{ib} fires at time $t + 12$ (causing neuron l_{ia} to forget at time $t + 13$), instruction l_j will begin to be simulated at time $t + 14$.

If neuron l_{i7} had chosen its first rule to fire (at time $t + 8$), both spikes from neurons l_{i7} and l_{i8} are received at time $t + 9$ by neurons l_{i9} and l_{ia} making both fireable. If neuron l_{i9} forgets at time $t + 10$, then neuron l_{ia} will fire at time $t + 11$ causing no neuron to be fireable at time $t + 12$ (an invalid computation). If neuron l_{ia} fires at time $t + 10$, then neuron l_{i9} will forget at time $t + 11$ and initiating instruction module l_k at time $t + 12$.

The SUB instruction is easier to simulate (since no checking for zero is necessary) and is shown in Figure 3. This module is again initiated by a single spike in neuron l_i which sends a spike to both neuron l_{i1} and l_{i2} at time $t + 1$. These two neurons spike during the next two time steps ($t+2$ and $t+3$) causing neuron l_{i3} to spike at time $t + 4$ (sending a spike to neuron r_n) and l_{i4} to spike at time $t + 5$. Due to the delay, the second spike sent to neuron l_{i3} is not received. If the counter r_n is storing a positive count, neuron r_n will fire during time $t + 6$ causing neuron l_{i5} to receive two spikes at the same time. If this occurs, neuron l_{i5} fires and initiates instruction module l_j. If neuron r_n contains only one spike, no neuron will be fireable at time $t + 6$ creating an invalid computation.

Since neuron r_n may be shared with other instructions we must guarantee additional neurons are not initiated when r_n spikes. Neuron r_n only has outgoing edges to neurons $l_{i'5}$ where $l_{i'} = (SUB(r_n), l_{j'})$. None of these neurons will fire since no additional spike is present. However, these neurons must be cleaned-up to guarantee no spike lingers causing problems later. To achieve this, when neuron l_{i5} fires, it has a delay of $s - 1$ time steps where s is the number of instructions of the form $l_{i'} = (SUB(r_n), l_{j'})$ in $\mathcal{M}$. This delay allows every $l_{i'5}$ to forget the extra spike it received before the next instruction executes.

The $HALT$ instruction is simulated by the output module shown in Figure 4. The $HALT$ instruction initiates the output neurons and then halts the computation. The module is initiated by a single spike sent to neuron r_1 which causes the value stored in neuron r_1 (x_1) to be output. (After the first output x_1 is sent to the environment, the neuron r_2 is triggered to output x_2.) For each

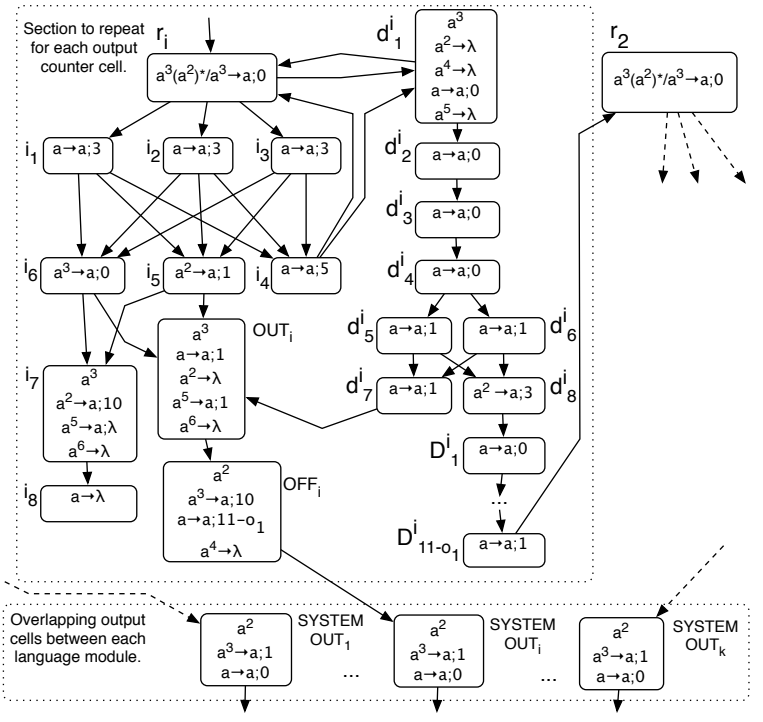


Fig. 4. k-Output Strongly Sequential Unbounded SNP Module

neuron r_i, when it is triggered with an additional spike it will contain an odd number of spikes. (This never happens previously in the computation since the output counters are non-decrementing counters.) This causes the neuron to fire sending spikes to neurons i_1, i_2, i_3, and d_1^i making all four neurons fireable. Now neuron d_1^i contains 4 spikes which can be forgotten. If the forgetting rule is applied before all neurons i_1, i_2, and i_3 have fired, there will be no fireable rule at some time slot between $t+6$ and $t+8$ causing an invalid computation. A valid computation nondeterministically fires neurons i_1, i_2, and i_3 at times $t+2$, $t+3$, and $t+4$. Then neuron d_1^i forgets its spikes at time $t+5$. This causes neurons i_4, i_5, and i_6 to fire sequentially during the next three time steps ($t+6$ to $t+8$) with delays respectively.

Neuron i_4 fires sending a spike to neurons r_i and d_1^i at time $t+6$ which will nondeterministically guess that either neuron r_i has been fully decremented (causing neuron d_1^i to spike at time $t+7$) or that it hasn't (causing neuron r_i to spike at time $t+7$). Neurons i_5 and i_6 work together to send two spikes to neurons i_7 and OUT_i simultaneously. Thus at time $t+9$ both neurons are fireable. The first time this occurs, neuron OUT_i will fire at time $t+9$ sending a spike to neuron OFF_i with a delay of 1 time step. During this delay, neuron i_7 will forget its spikes (time $t+10$) and then neuron OFF_i will spike (time $t+11$) with a delay of 10 time steps causing neuron SYSTEM OUT_i to receive a spike at time $t+21$.

(If neurons i_7 and OUT_i fire in the reverse order, there will be no fireable rule at time $t+11$ causing the computation to become invalid.) This loop is repeated identically for the next decrement of r_i until neurons i_7 and OUT_i again become fireable. (So for time steps $t+12, \cdots, t+19$ a valid computation will cause the neurons $r_i, i_1, i_2, i_3, d_1^i, i_4, i_5, i_6$ to spike sequentially where the order of i_1, i_2, i_3 can be interchanged.) Remember that each $x_i \geq 2$ so a valid computation must make this loop at least twice. Now neuron i_7 will spike at time $t+20$, neuron OUT_i will forget at time $t+21$, and neuron SYSTEM OUT_i will spike at time $t+22$ with a delay of one time step. This initiates the output value at time $t+23$. (Again, if neuron OUT_i is fired before neuron i_7 there will be no fireable rule at time $t+30$ causing the computation to become invalid.) If counter r_i contains a count greater than 2, then this decrementing loop must be rerun $x_i - 2$ more times. For each of these loops, when both neurons i_7 and OUT_i are fireable, a valid computation will have neuron i_7 fire at each time $t+20+11x$ and neurons OUT_i and i_8 fire at times $t+21+11x$ and $t+22+11x$ (nondeterministically) for $0 \leq x \leq x_i - 2$.

When the system guesses that r_i no longer can be decremented, neuron d_1^i fires at some time $t+23+11x$ (which sends a spike to neuron r_i making it no longer fireable and leaving it with 2 spikes if the guess was correct). This is followed by neurons $d_2^i, \cdots, d_8^i$ firing sequentially during times $t+24+11x, \cdots, t+30+11x$. Neuron d_7^i sends a spike to neuron OUT_i making it fireable during time $t+31+11x$. At time $t+31+11x$ neuron OUT_i fires followed by neuron i_8 and then neuron OFF_i with a delay of $11 - o_i$ steps. (If neuron i_8 fires first, the computation becomes invalid.) During this delay, neurons D_1^i to $D_{11-o_1}^i$ fire (time $t+34+11(x_i-2) = t+12+11x_i$ to time $t+22+11x_i - o_i$). (Since $0 \leq o_i \leq 10$ there is guaranteed to be at least one of these delay neurons.) Once neuron $D_{11-o_i}^i$ fires, neuron SYSTEM OUT_i receives the delayed spike and spikes at time $t+11x_i+23-o_i$. Therefore, the output generated by neuron SYSTEM OUT_i is $11x_i - o_i$. Finally, if $i < k$ (i.e. this is not our last output), neuron r_{i+1} receives a spike and then fires during the next time step.

This process is repeated for each r_i for $1 \leq i \leq k$ allowing every generated output x_i to be sent to the environment as a multiple of 11 with an offset of o_i by the corresponding neuron SYSTEM OUT_i. □

Lemma 3. *If $Q(\mathcal{M}) \subseteq N^k$ is generated by a k-output PBCM $\mathcal{M}$, then it can be generated by a k-output strongly sequential unbounded SNP Π.*

Proof. Let $\mathcal{M}$ be a k-output PBCM $\mathcal{M}$ which generates the set $Q(\mathcal{M})$. Define $11^k * 2^k$ new k-output PBCMs $\mathcal{M}^{(o_1,\cdots,o_k,c_1,\cdots,c_k)}$ such that $Q(\mathcal{M}^{(o_1,\cdots,o_k,c_1,\cdots,c_k)}) = \{((x_1+o_1)/11, \cdots, (x_k+o_k)/11) \mid (x_1, \cdots, x_k) \in Q(\mathcal{M}), (x_i+o_i)$ is divisible by 11 for $1 \leq i \leq k$, if $c_i = 1$ then $(x_i+o_i)/11 \geq 2$, if $c_i = 0$ then $(x_i+o_i)/11 = 1\}$ where $0 \leq o_i \leq 10$ and $c_i \in \{0,1\}$. This can easily be done by simulating $\mathcal{M}$ with k new output counters. Once the computation halts, the old counters are decremented by there appropriate offset (o_i for counter i) respectively and transferred to the new output counters. The transfer is done by decrementing each old output counter and incrementing the new corresponding output counter

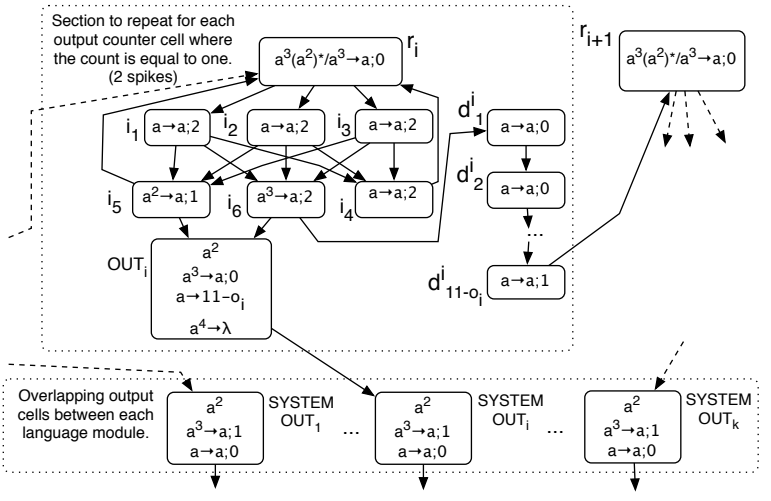


Fig. 5. k-Output Strongly Sequential Unbounded SNP Module with Offsets for Counters Containing One

for every eleven decrements. For each counter i where $c_i = 0$, we decrement the original output by exactly 11 and increment the new output counter by 1. For each counter i where $c_i = 1$, we decrement the original output $22 + 11x$ times (where $x \geq 0$) and increment the new output counter by 1 for each 11 decrements. The machine nondeterministically guesses when the old counter has reached zero.

We construct a k-output strongly sequential unbounded SNP $\Pi_{11}^{(o_1,\cdots,o_k,c_1,\cdots,c_k)}$ which generates the set $Q(\Pi_{11}^{(o_1,\cdots,o_k,c_1,\cdots,c_k)}) = \{(11x_1 - o_1, \cdots, 11x_k - o_k) \mid (x_1, \cdots, x_k) \in Q(\mathcal{M}^{(o_1,\cdots,o_k,c_1,\cdots,c_k)})\}$ by simulating each $\mathcal{M}^{(o_1,\cdots,o_k,c_1,\cdots,c_k)}$. We use the construction given in Lemma 2 with a slight modification to the output module. In the output module of Lemma 2 (Figure 4) we duplicate the neurons given for each output. We use this structure only for the set of neurons $\{r_i \mid c_i = 1, 1 \leq i \leq k\}$ (meaning $x_i \geq 2$). To output the remaining neurons $\{r_i \mid c_i = 0, 1 \leq i \leq k\}$ we use a new output module given in Figure 5. This new module allows these neurons which store the value 1 (2 spikes) to be decremented only once in a strongly sequential manner.

We output the value stored in neuron r_i by sending an additional spike to neuron r_i. This causes neuron r_i to fire consuming 3 spikes at time $t+1$ (where t is the time that the initial spike is sent to neuron r_i) sending a spike to neurons i_1, i_2, and i_3. These three neurons fire nondeterministically during time steps $t+2$, $t+3$, and $t+4$ causing neurons i_4, i_5, and i_6 to fire sequentially during the next three time steps. These spiking neurons send a spike to neuron OUT_i and two spikes to neuron r_i at time $t+7$. The two spikes sent to neuron r_i allow it to halt with the appropriate two remaining spikes (assuming neuron r_i initially did store a correct count of one). Neuron OUT_i will spike at time $t+8$ causing neuron SYSTEM OUT_i to spike at time $t+9$ (with a delay of one). Also at time

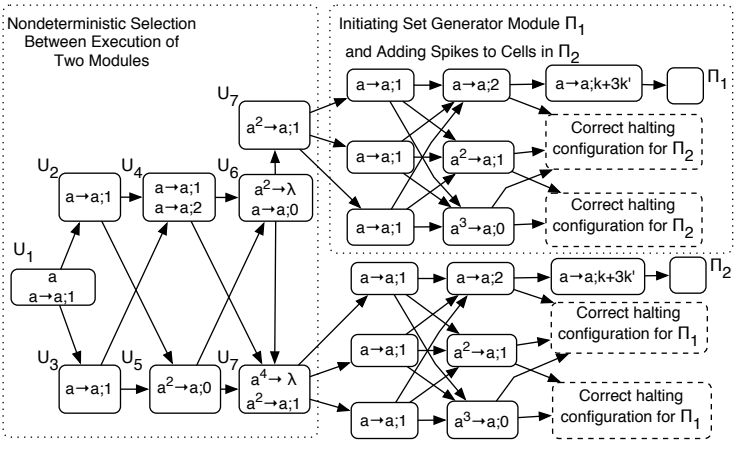


Fig. 6. Strongly Sequential Unbounded SNP Union Module

$t + 9$ neurons OUT_1 and d_1^i receive spikes from neuron i_6. If neuron d_1^i spikes before neuron OUT_i spikes, time $t + 15$ will have no fireable rule causing an invalid computation. A valid computation will have neuron OUT_i spike followed by neurons $d_1^i, \cdots, d_{11-o_i}^i$. When neuron $d_{11-o_i}^i$ fires (with a delay of one) the spike sent by neuron OUT_i is received by neuron SYSTEM OUT_i. This will cause SYSTEM OUT_i to spike at time $t + 10 + 11 - o_i$ (generating the number $11 - o_i$). If $i \neq k$, the spike triggering neuron r_{i+1} is received during this last step causing neuron r_{i+1} to spike at time $t + 22 - o_i$. After outputting r_i, all neurons in this module will contain zero spikes with neuron r_i containing two spikes (unless r_i did not contain the correct count of one).

To regain $Q(\mathcal{M})$, we take the union of all of these sets. Figure 6 shows a union module which operates in a strongly sequential manner and nondeterministically executes either machine Π_1 or Π_2. The set R (containing neurons which will contain exactly two spikes in a halting computation) is defined as neurons $r_1, \cdots, r_k$ of both set modules. The set module that is picked nondeterministically will leave it's neurons in the correct configuration when the system halts (assuming a correct computation occurred), but the alternate set module will still have its neurons left in the initial configuration. Therefore, we will use the union module to guarantee that the alternate set generating module is left in a correct halting configuration. If the machine Π_1 is picked to be executed, two spikes are sent to neurons $r_1, \cdots, r_k$ in Π_2 to leave them with two spikes. Two spikes are also sent to neurons $OUT_1, \cdots, OUT_k$ in Π_2 which allows the initial spikes to be forgotten in k steps. The neurons d_1^i in Π_2 where $c_i = 1$ are also sent two spikes while neurons i_7 and OUT_i (where $c_i = 1$) are sent three spikes. This allows these initial spikes to be forgotten in $3k'$ steps where k' is the number of $c_{i'}$'s $= 1$. Since the computation of Π_1 is initiated after a delay of $k + 3k'$ steps, all of the initial spikes in Π_2 are forgotten before the execution of Π_1 can occur. (If machine Π_2 is non deterministically selected for execution, the same

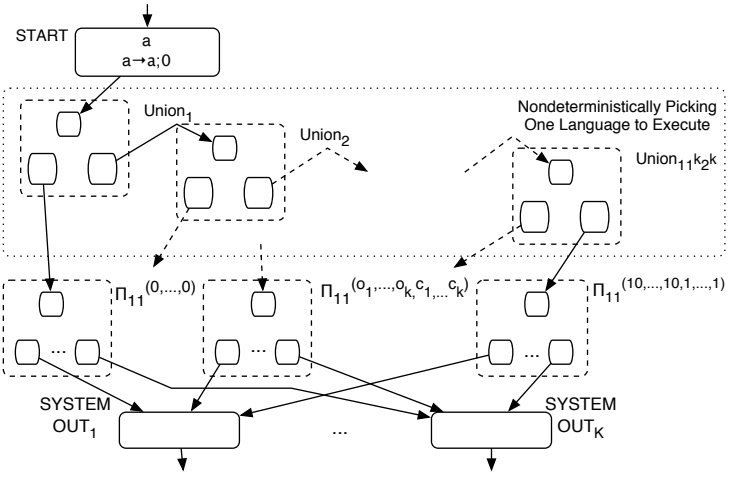


Fig. 7. k-Output Strongly Sequential Unbounded SNP Generating the Union of Sets with Offsets

procedure occurs to leave Π_1 in a correct halting configuration.) The resulting generated set is $Q(\Pi_1) \cup Q(\Pi_2)$. Figure 7 shows how each of these set generator modules can be linked (with the system output neurons of each set generator module combined) so that at the start of computation, one and only one set generator module is executed which generates some k-tuple. This entire system will now generate the set $Q(\Pi) = Q(\mathcal{M})$. □

From Lemmas 1, 2, and 3 we get the following result.

Theorem 1. *A set $Q(\mathcal{M}) \subseteq N^k$ is generated by a k-output PBCM $\mathcal{M}$ if and only if it can be generated by a k-output strongly sequential unbounded SNP Π. Hence, such SNPs are not universal.*

2.2 Relaxing the Strongly Sequential Operation

If we relax our *strongly sequential unbounded SNP* model to allow time steps with no fireable rules, we can show universality. We will call such models simply *sequential unbounded SNPs*. Universality is shown by giving a construction such that any 1-output CM $\mathcal{M}$ can be simulated by a sequential unbounded SNP Π. It is well known that 1-output CMs are universal.

Theorem 2. *Sequential unbounded SNPs are universal.*

Proof. Given some multicounter machine $\mathcal{M}$ generating a set $Q(\mathcal{M})$, we can simulate $\mathcal{M}$ with a sequential unbounded SNP Π_{11} which generates the set $Q(\Pi_{11}) = \{11x \mid x \in Q(\mathcal{M})\}$. (We only need one non-decrementing output counter r_1 in $\mathcal{M}$.) The SNP Π_{11}'s initial configuration will again start with

the initial configuration for each module along with a single spike in neuron l_0. When the computation halts, all Π_{11}'s neurons are empty except neuron r_1 which contains exactly two spikes.

To create a sequential unbounded SNP generating exactly $Q(\mathcal{M})$ we use the same ideas and methods given in Lemma 3. First we create 11 new multicounter machines $\mathcal{M}^o$ for $0 \leq o \leq 10$ generating the sets $Q(\mathcal{M}^o) = \{(x+o)/11 | x \in Q(\mathcal{M}), (x+o)$ is divisible by 11$\}$. We then simulate each $\mathcal{M}^o$ with a sequential unbounded SNP Π^o_{11} which generates the set $Q(\Pi^o_{11}) = \{11x - o \mid x \in Q(M^o), 0 \leq o \leq 10\}$. The union of these 11 set generators (using the union module in Figure 6) creates one large sequential unbounded SNP Π' such that $Q(\mathcal{M}) = Q(\Pi')$. Again the union module is used to both select which set generator module to execute and to guarantee a correct halting configuration (by sending two spikes to the unexecuted neuron OUT which are forgotten and one spike to the unexecuted neuron r_1 which remains along with a second spike sent at the end of computation by neuron SYSTEM OUT).

To show that this SNP model can simulate any 1-output CM $\mathcal{M}$ by a 1-output sequential unbounded SNP we must simulate instructions of the forms $l_i = (ADD(r_n), l_j, l_k)$, $l_i = (SUB(r_n), l_j)$, and $l_i = HALT$. For each of these instruction types, we again create an SNP module. We use our previous technique given in Lemma 2 (Figure 2) to simulate each ADD instruction.

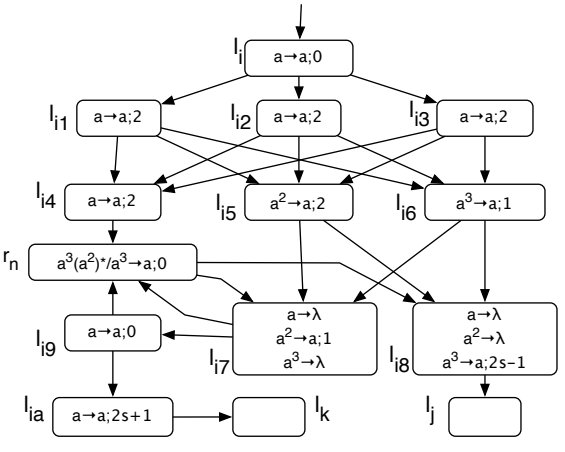


Fig. 8. Sequential Unbounded SNP Subtraction Module

To simulate a SUB instruction, the previous subtraction module was unable to test a counter for zero so we must create a new module for this instruction type. The new module is shown in Figure 8. It is initiated with a single spike in neuron l_i which immediately sends a spike to neurons l_{i1}, l_{i2}, and l_{i3} during time $t+1$ (where t is the time the initial spike is sent to neuron l_i). These three neurons nondeterministically spike during the next three steps (time $t+2$, $t+3$, and $t+4$). This causes neurons l_{i4}, l_{i5}, and l_{i6} to spike sequentially during the

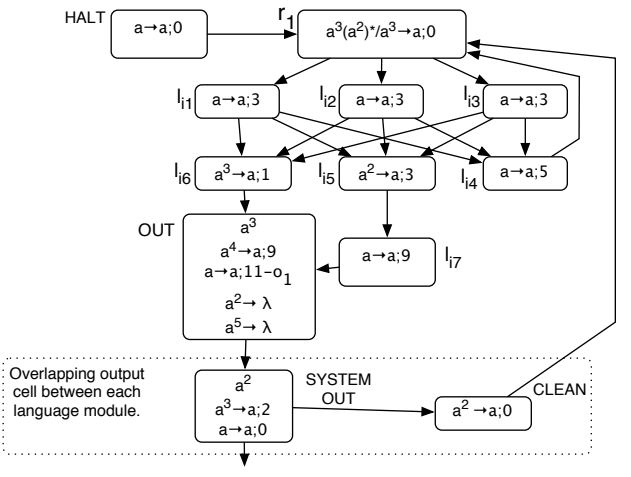


Fig. 9. Sequential Unbounded SNP Output Module

following three time steps (time $t+5$, $t+6$, and $t+7$). The firing of neuron l_{i4} sends a spike to neuron r_n causing it to contain an odd number of spikes. If the counter is non-zero, neuron r_n will fire during time step $t+8$, otherwise no neuron will fire at time $t+8$. (This step with a possibility of no fireable rule which allows us to test the counter for zero.)

If neuron r_n fired, neurons l_{i7} and l_{i8} receive three spikes simultaneously. This causes neuron l_{i8} to fire initiating instruction l_j after a delay. The delay is necessary to guarantee all the remaining spikes are 'cleaned-up'. Since neuron r_n sends spikes to all neurons $l_{i'7}$ and $l_{i'8}$ where $l_{i'} = (SUB(r_n), l_{j'}, l_{k'})$, these neurons receive a single spike during the computation of instruction l_i. These spikes must be forgotten before the next instruction executes. (None of these neurons will be fireable.) We denote the number of these instructions by s so l_{i8} will delay firing for $2s-1$ steps to guarantee that everything is cleaned-up.

If neuron r_n does not fire, neurons l_{i7} and l_{i8} receive only two spikes simultaneously. This causes neuron l_{i7} to fire (at time $t+9$ or $t+10$ with neuron l_{i8} forgetting its spike during the alternate time slot) sending a spike to neurons l_{i9} and r_n. During the next time step neuron l_{i9} fires sending a spike to neurons r_n and l_{ia}. Neuron l_{ia} is now fireable initiating instruction l_k after a delay to allow the system to 'clean-up'. Now neuron r_n fires removing the spikes added by the current subtract module. This sends spikes to all neurons $l_{i'7}$ and $l_{i'8}$ where $l_{i'} = (SUB(r_n), l_{j'}, l_{k'})$ so $2s$ steps are needed to guarantee that these spikes are all removed. (The delay for neuron l_{ia} allows $2s+1$ time steps to pass to allow neuron r_n to spike and guarantee all neurons $l_{i'7}$ and $l_{i'8}$ forget their spikes.)

To simulate $l_i = (HALT)$ we create the module given in Figure 9. This module decrements neuron r_1 and sends three spikes to neurons l_{i1}, l_{i2}, and l_{i3} which fire nondeterministically during the next three time steps. Neurons l_{i4}, l_{i5}, and l_{i6} now spike sequentially. Neuron l_{i4} initiates the next loop (if counter

neuron r_1 is not yet zero) after a delay of 5 time steps. Meanwhile, neurons l_{i6} and l_{i5} operate together to trigger neuron OUT which triggers the SYSTEM OUT neuron. Neuron l_{i7} spikes one loop behind neuron 6 so that neuron OUT receives a single spike both the first time r_1 is decremented and the last time r_1 is decremented. During intermediate loops, neuron OUT receives two spikes together which are then forgotten. After neuron SYSTEM OUT fires for a second time (generating the output $11x - o$), neuron CLEAN fires sending a spike to neuron r_1. At this point, the computation halts with all neurons empty except neuron r_1 which contains two spikes (assuming a correct computation). □

2.3 Strongly Sequential General SNPs

If we keep the strongly sequential requirement, but now allow general neurons, the model becomes universal. In fact, for universality we only need three general neurons which allow the rules $a^3(a^2)^*/a^3 \to a; 0$ and $a \to a; 2$ to coexist. (This is due to the fact that 3 counters is sufficient for universality in a CM.)

Theorem 3. *Strongly sequential general SNPs are universal.*

Proof. Again this can be shown in the same manner as previous proofs and follows from the proof of Theorem 2 with a change to the subtraction module. The new subtraction module is shown in Figure 10.

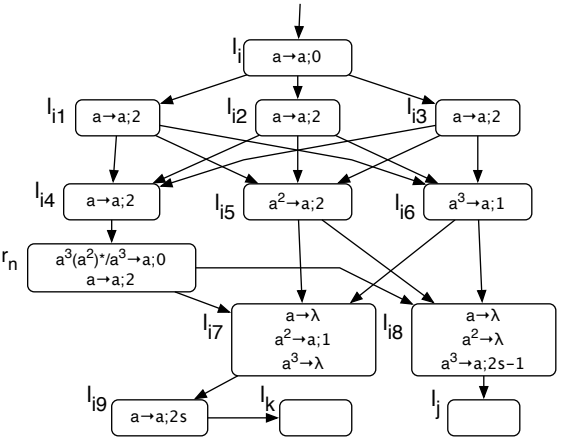


Fig. 10. Strongly Sequential General SNP Subtraction Module

The operation of this module occurs identically to the subtraction module in the proof of Theorem 2 up until a spike is sent to neuron r_n. Here, if r_n contains a non-zero count, the first rule will be applied otherwise the second rule will be applied which allows a strongly sequential computation. The remainder of the computation also follows from the proof of Theorem 2 except that there is

no need to remove a spike from neuron r when it contains a zero count. (This module allows many computations which are invalid based on our definition, but there exists a computation which correctly subtracts one (two spikes) from neuron r and has a neuron spike during each time step.) □

References

1. W. Gerstner, W Kistler: *Spiking Neuron Models. Single Neurons, Populations, Plasticity*. Cambridge Univ. Press, 2002.
2. S. Greibach: Remarks on blind and partially blind one-way multicounter machines. *Theoretical Computer Science*, 7, 3 (1978), 311–324.
3. O. Ibarra, A. Păun, G. Păun, A. Rodriguez-Paton, P. Sosik, and S. Woodworth: Normal forms for spiking neural P systems. *Proc. Fourth Brainstorming Week on Membrane Computing*, vol. II, Fenix Editora, Sevilla, 2006 (also available at [11]).
4. M. Ionescu, Gh. Păun, T. Yokomori: Spiking neural P systems. *Fundamenta Informaticae*, 71, 2-3 (2006), 279–308
5. W. Maass: Computing with spikes. *Special Issue on Foundations of Information Processing of TELEMATIK*, 8, 1 (2002), 32–36.
6. W. Maass, C. Bishop, eds.: *Pulsed Neural Networks*. MIT Press, Cambridge, 1999.
7. M. Minsky: *Computation – Finite and Infinite Machines*. Prentice Hall, Englewood Cliffs, NJ, 1967.
8. Gh. Păun: *Membrane Computing – An Introduction*. Springer-Verlag, Berlin, 2002.
9. Gh. Păun, M.J. Pérez-Jiménez, G. Rozenberg: Spike trains in spiking neural P systems. *Intern. J. Found. Computer Sci.*, to appear (also available at [11]).
10. Gh. Păun, M.J. Pérez-Jiménez, G. Rozenberg: Infinite spike trains in spiking neural P systems. Submitted, 2006.
11. The P Systems Web Page: `http://psystems.disco.unimib.it`.

Chemical Information Processing Devices Constructed Using a Nonlinear Medium with Controlled Excitability

Yasuhiro Igarashi[1], Jerzy Gorecki[1,2], and Joanna Natalia Gorecka[3]

[1] Institute of Physical Chemistry, Polish Academy of Sciences,
Kasprzaka 44/52, 01-224 Warsaw, Poland
[2] Faculty of Mathematics and Natural Sciences,
Cardinal Stefan Wyszynski University
Dewajtis 5, Warsaw, Poland
gorecki@ichf.edu.pl
[3] Institute of Physics, Polish Academy of Sciences,
Al. Lotnikow 36/42, 02-668 Warsaw, Poland

Abstract. Chemical signals composed of excitation pulses can be processed in a medium with an appropriate geometrical arrangement of excitable and non-excitable regions. In this paper we consider two types of signal processing devices: a binary logic gate and a four input, neuron like structure. Using numerical simulations, we demonstrate that small local changes in the excitability level of the medium can completely change the function executed by the device and can thus be used to program it.

1 Introduction

Recently, a growing interest in unconventional methods of information processing can be observed. This interest has been motivated by a common expectation that unconventional computers will be able to solve specific problems faster than the classical ones, characterized by the von-Neumann-type architecture. A significant number of studies are concerned with reaction-diffusion computing, i.e. the information processing with the medium described by a set of parabolic partial differential equations [1]. The reaction-diffusion computing includes very different systems like arrays of quantum dots acting as coupled single electron oscillators, or far-from-equilibrium chemical systems like Belousov-Zhabotinsky (BZ) reaction. However, due to similarities in the mathematical description many results concerned with information processing in a specific type of reaction-diffusion medium are generic and they can be easily adopted to other systems. Having this in mind, one can find numerous studies on information processing in chemical systems because both experiments and simulations are relatively easy. On the other hand, the experiments with especially prepared semiconductors in which the carrier density evolves according to reaction-diffusion equations [2] are very difficult, but it is expected that these systems will have an important impact on industrial applications of unconventional computers in the near future.

C.S. Calude et al. (Eds.): UC 2006, LNCS 4135, pp. 130–138, 2006.

In this paper we are concerned with information processing in an excitable chemical reaction-diffusion medium. In a spatially distributed medium, a local perturbation can develop into a propagating pulse of excitation in which the concentration of the activator is high. Spatio-temporal evolution of the medium can be interpreted in the language of logic operations when one associates the excitation pulse with the logical "true" state and the non-excited medium with the logical "false". Within such an interpretation, a propagating pulse of excitation corresponds to a single bit of information moving in space. A chemical signal is formed by a number of pulses. For information processing applications, it is convenient to consider a nonhomogeneous medium characterized by different excitability levels at various points of space. Let us assume that we are able to fix the local excitability level from a high value (a pulse propagates retaining its shape) to a low one where excitations rapidly decay and signals die. In such nonhomogeneous medium, one can create excitable channels surrounded by non-excitable medium, so pulses of excitation can propagate within channels without interfering one with another.

Logical operations are executed through interaction of pulses. By selecting the proper geometry of excitable and non-excitable (inhibitory) regions, one can force the required type of interaction between pulses and construct devices executing given logical functions, like for example binary logical operations [3]. Using the concepts of pulse based logic, a number of devices performing specific information processing operations have been designed [4,5,6,7,8,9,10]. The majority of such devices should be classified as instant machines able to perform a single specific function. Typical instant machines, even if they are linked together, do not give us the flexibility required to perform multiple tasks. In systems composed of instant machines the change of connections between signal processing elements seem to be the only method to modify the function performed by the device. However, it is quite easy to design chemical signal processing devices in which the interaction between signals strongly depends on external factors regulating the local excitability level like: temperature, inflow of reactants or light intensity. Such factors can be used to modify the performed signal processing operations and so they can be used for programming [11]. In this paper, we discuss two devices that are able to perform different functions depending on the local excitability levels of the medium they are composed of. One of them is a logical gate that can execute different functions depending on the excitability of its junctions. Another is a neuron-like device for which the excitation threshold can be controlled by the parameters of medium. We demonstrate that significant changes in functionality of these devices can be introduced with relatively small changes in the parameters. Both considered devices can be incorporated into programmable networks in which the functionality is controlled by an external factor. Such control is an important step towards realization of reaction-diffusion devices able to perform complex operations. Our studies are based on numerical simulations of information processing in Ru-catalyzed BZ reaction. In such a case the excitability can be easily controlled by the external illumination. We have selected such system because the experimental verification of our ideas does

not seem difficult. Of course the presented results seem generic and they can be applied to the other types of excitable medium.

2 Different Operations at a Single Binary Gate

The device that performs different binary logic operations depending on local illumination levels is shown in Fig. 1. Two input channels (A and B) are linked with the output channel O through a number of excitable channels and non-excitable gaps separating them. All junctions in the device illustrated in Fig. 1 are non-symmetrical and for a moderate illumination at the gap they work as chemical signal diodes [3,12] transmitting the signals from a flat to a triangular excitable region only. Of course, for a low illumination level the junction transmits chemical signals in both directions and for a high illumination level it is unpenetrable.

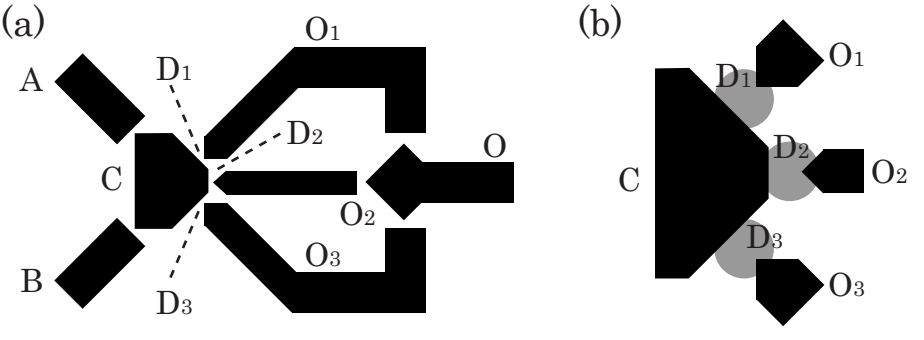


Fig. 1. A logic gate that executes different binary operations on inputs A and B depending on the illumination level at the diodes D_1, D_2 and D_3. (a) the geometry of excitable(black) and non-excitable(white) regions in the device. (b) the position of regions with controlled excitability (grey).

We have studied the behaviour of the device in numerical simulations based on 2-variable Oregonator model of Ru-catalyzed BZ reaction :

$$\begin{aligned} \frac{\partial u}{\partial t} &= \frac{1}{\epsilon}\left[u(1-u) - (fv+\phi)\frac{u-q}{u+q}\right] + D_u\nabla^2 u \\ \frac{\partial v}{\partial t} &= u - v \ . \end{aligned} \tag{1}$$

where u and v are dimensionless concentrations of $HBrO_2$ and $Ru(4,4'-\mathrm{dm}-\mathrm{bpy})_3^{3+}$, respectively. The diffusion of the ruthenium catalytic complex is neglected because it is much smaller than that of other reagents. In the model the parameter ϕ represents the rate of bromide production caused by illumination and it is proportional to the applied light intensity. Bromine is an inhibitor of

Ru-catalyzed BZ reaction so the non-illuminated areas are excitable and the illuminated ones are not. Adjusting the local value of ϕ the regions with required level of excitability can be created. The results presented in Fig. 2 has been obtained from a numerical solution of Eqs. (1) for a section of the system covering inputs, the central area and outputs O_1, O_2 and O_3. The right part of the device just groups O_1, O_2 and O_3 into a single output signal. The square grid of 150×150 points was considered with no flux boundary conditions at its ends. The free flow boundary conditions between excitable and non-excitable regions have been assumed. The calculations have been performed for $\epsilon = 5.0\times10^{-2}$, $q = 1.5\times10^{-4}$, $f = 1.0$ and $D_u = 1.0$. We have used $\Delta x = 0.18$ and $\Delta t = 8.1\times10^{-5}$. The illumination in the excitable regions is $\phi = 7.0\times10^{-3}$ and in the inhibitory ones 2.0×10^{-1}. The excitability of the gaps in diodes D_1, D_2 and D_3 (cf. Fig. 1B) have been individually controlled. In Fig. 2, pulses of excitation are marked as light areas on the structure.

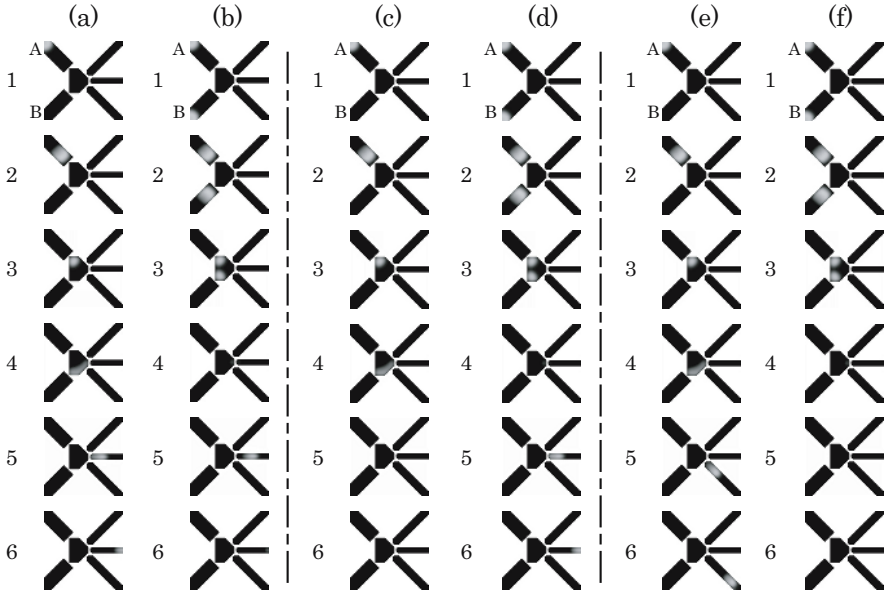


Fig. 2. (a) and (b) - an OR gate, (c) and (d) - an AND gate, (e) and (f) - an XOR gate. The columns (a),(c),(e) show the time evolution of pulses for the input $A = 1$ and $B = 0$, (b),(d),(f) correspond to $A = 1$ and $B = 1$. The time difference between successive figures in a column is 1 unit.

Figs. 2(a),(b) show the evolution of excitation pulses coming from inputs A and B, when the gap illuminations are $\big(\phi(D_1), \phi(D_2), \phi(D_3)\big) = (0.20, 0.18, 0.20)$. In such a case the device works as an OR gate. The excitability at D_2 is high enough to transmit excitation of the central area C generated by a single input pulse. When two input pulses come to C, the excitations are

integrated and transmitted as a single output. The excitabilities of D_1 and D_3 are low, so no pulse is transmitted through these gaps. Fig. 2(c),(d) correspond to $\big(\phi(D_1), \phi(D_2), \phi(D_3)\big) = (0.20, 0.195, 0.20)$. In such case the device works as an AND gate. The excitability of D_2 is now a bit lower than before and output pulse appears only when two input pulses come to C simultaneously. Fig. 2(e),(f) correspond to $\big(\phi(D_1), \phi(D_2), \phi(D_3)\big) = (0.19, 0.20, 0.19)$ and such case the device works as an XOR gate. The excitability in D_2 is so low that an output pulse does not appear in O_2 even if two input pulses come to C simultaneously. The device shown in Fig. 1 can work as NOT gate when one of input pulses in XOR gate is considered as the reference signal [3].

3 Chemical Neuron Controlled by External Illumination

Another information processing device that can be easily controlled by illumination is illustrated in Fig. 3. It has a neuron like shape. We can recognize four identical excitable input channels (dendrites), the cell body C, and the excitable output channel A (the axon). All input channels have the same length, width $d = 1.2213 a.u.$ and the same excitability characterized by Φ_a. The widths of input channels have been chosen such that the channels are subexcitable which means that the amplitude of excitation pulse decays as it travels along the channel. Choosing the proper lengths of input channels we are able to obtain the required amplitude of activator at the gap separating an input channel from C after the input excitation is applied at the end of input channel (dark gray areas in Fig. 3). The output channel A (Φ_a) is significantly wider ($d = 1.7641 a.u.$) than the input channels and it is just excitable. An output pulse propagates in this channel without changing its shape. The input channels are separated from excitable central area C ($\Phi_c = \Phi_a$) by passive gaps with the same width. The properties of the gaps are controlled by the illumination (Φ_p) of the surrounding nonexcitable medium. The response of neuron like structure to pulses of excitation is calculated using 3-variable Oregonator model for photosensitive Belousov-Zhabotynsky reaction [13]:

$$\begin{aligned} \varepsilon_1 \frac{\partial u}{\partial t} &= u\,(1-u) - w\,(u-q) + D_u \Delta u \\ \frac{\partial v}{\partial t} &= u - v \\ \varepsilon_2 \frac{\partial w}{\partial t} &= \Phi + f v - w\,(u+q) + D_w \Delta w\ . \end{aligned} \tag{2}$$

Here w is the dimensionless concentrations of Br^-. The dimensionless units of space and time has been chosen to scale the reaction rates and the diffusion coefficient $D_u (= 1)$. Eqs. (1) can be obtained from Eqs. (2) if we assume that diffusion and relaxation of Br^- are much faster than the other variables ($\varepsilon_2 << \varepsilon_1$). In numerical simulations we use $D_u = D_w = 1$, $\varepsilon_1 = 0.08$, $\varepsilon_2 = 9.7 \cdot 10^{-4}$, $f = 1.12$ and $q = 0.002$. The calculations have been performed using the explicit Euler algorithm for the diffusion combined with the 4-th order Runge-Kutta

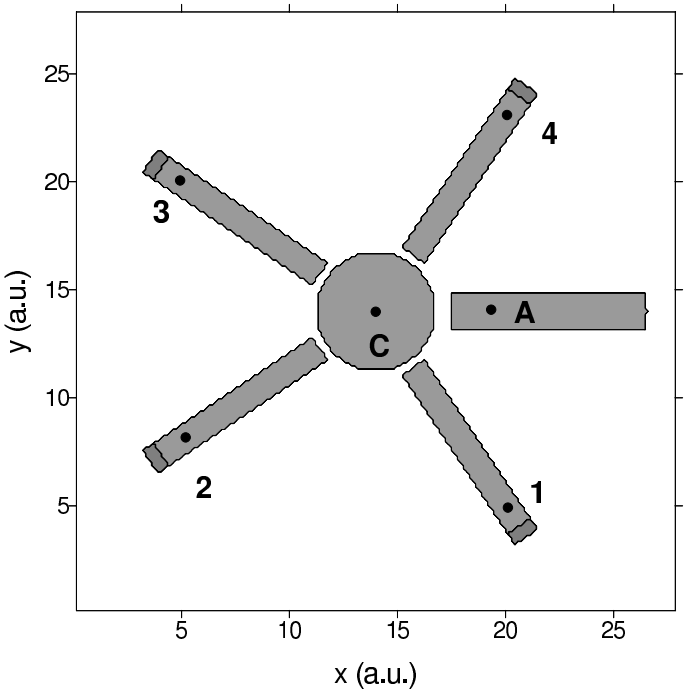


Fig. 3. A chemical neuron-like signal processing element. The white area is passive, the gray parts active. Regions where input channels 1,2, 3,4 are excited are marked dark gray.

method for chemical kinetics with spatial step $\Delta x = 0.14$ and temporal step $\Delta t = 0.0001$. We have studied the response of the neuron for excitation of a chosen number of input channels for different values of Φ_a and Φ_p. For example, Fig.4 shows the concentration of u at the marked points of the device (cf. Fig. 3) for the case when $\Phi_a = 0.00700336$ and $\Phi_p = 0.0454$. Fig. 4a shows the case when channels 2 and 3 are excited and Fig. 4b corresponds to the case when channels 1, 2 and 4 are excited. In the first case there is no input signal, in the second the input signal appears. The response of the chemical neuron to the correlated input signals for different values of Φ_a and Φ_p is summarized in Fig.5. It is interesting that the behavior of the neuron can be controlled by small changes in one of these parameters. If illumination is low (excitability high) the output signal appears if only one of the inputs is activated so the neuron works as a multiple OR gate ($A = I_1 \vee I_2 \vee I_3 \vee I_4$). For higher illuminations two of the inputs have to be activated to get an output signals so now $A = (I_1 \wedge I_2) \vee (I_1 \wedge I_3) \vee (I_1 \wedge I_4) \vee (I_2 \wedge I_3) \vee (I_2 \wedge I_4) \vee (I_3 \wedge I_4)$. For yet lower excitability three input signals activate the neuron and $A = (I_1 \wedge I_2 \wedge I_3) \vee (I_1 \wedge I_2 \wedge I_4) \vee (I_1 \wedge I_3 \wedge I_4) \vee (I_2 \wedge I_3 \wedge I_4)$. Further increase of illumination changes the executed function into multiple AND: $A = I_1 \wedge I_2 \wedge I_3 \wedge I_4$. Of course for yet lower excitability, the neuron never gets excited. The response of the device to a simultaneous excitation of specific input channels as a function of Φ_p or of Φ_a is summarized in Fig.5. For

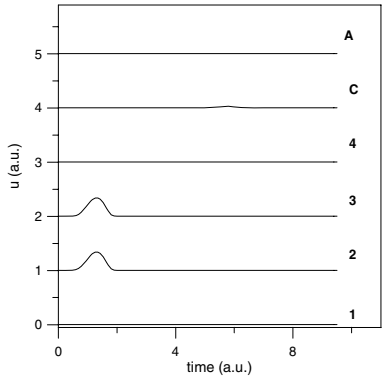


Fig. 4.a

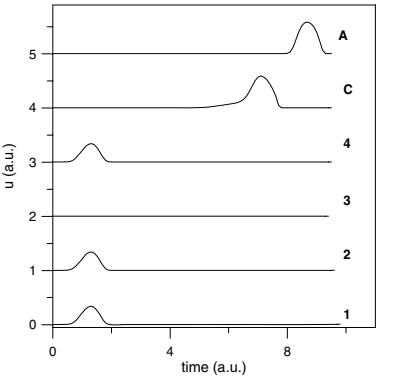


Fig. 4.b

Fig. 4. The time evolution of $u(r)$ at the selected points of the device (these points are marked by dots on Fig.3) for $\Phi_p = 0.0454$ and $\Phi_a = 0.00700336$. Fig 4a - channels 2 and 3 are initially excited. Fig 4b - channels 1, 2 and 4 are initially excited.

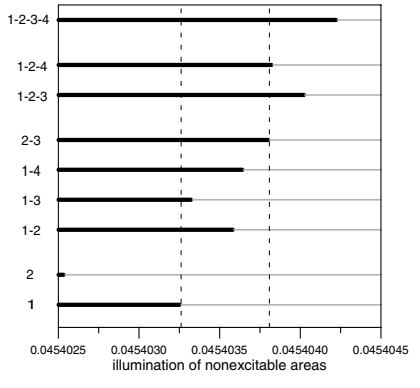


Fig. 5.a

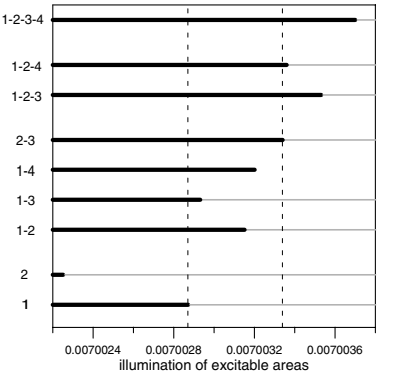


Fig. 5.b

Fig. 5. The response of the chemical neuron with respect to all combinations of the input signals. The thick black line indicates illuminations at which the output signal appears. The gray lines correspond to the cases where output is not excited. Two vertical dashed lines illustrate the minimum illumination at which no single input creates output excitation and the minimum illumination at which no combination of two input excitations generates an output signal. Fig. 5a — $\Phi_a = 0.007$, Fig. 5b — $\Phi_p = 0.0454$.

example (cf. Fig 5b), when $\Phi_p = 0.0454$, the neuron gets excited by a single excited input channel if $\Phi_a < 0.00700226$. For $0.00700288 \leq \Phi_a < 0.00700293$ a simultaneous excitation of any two (or more) channels gives an output pulse. When $0.00700335 \leq \Phi_a < 0.00700337$ the device does not produce the output

signal if it is excited by two channels only. Such signal appears after excitation of any three input channels. Finally, for $0.00700354 \leq \Phi_a < 0.00700370$ the device can be excited only by a simultaneous excitation of all inputs, so it works as "all or nothing" neuron. Between the intervals of illumination mentioned above, the device performs yet more complex logical functions that can be also important in special applications.

4 Discussion

In this paper we have discussed two examples of chemical signal processing devices that can be controlled by an external factor that determines the excitability of the medium. We have demonstrated that the functions performed can be completely changed by small changes in values of parameters controlling the medium. Our results can be seen as a generalization of the studies on propagation of wave fragments in sub-excitable medium [14,15] and their use for collision-based logic [16]. In the devices discussed above the required changes in illumination level are much smaller than in the case of homogeneous medium considered in the papers mentioned above, because the architecture of excitable and nonexcitable regions has been already designed for information processing.

We have adopted two strategies of chemical neuron control. According to the first of them, the properties of gaps separating signal channels are properly adjusted. Having in mind analogies with biological systems, they can be seen as a modification of synapses. The second strategy controls the excitability of signal channels that corresponds to the properties of nerve connections. In both cases (cf. Fig. 5) we have found that tiny changes in excitability significantly modify the functions executed by a neuron, so they can be related to minor changes in concentrations of reagents forming a complex network of biochemical reactions within a nerve cell.

The spatial scale of the considered devices is determined by the diffusion speed and the reaction rates. For example, in Ru-catalyzed BZ reactions the channels are about 1 mm wide and gap widths are measured in hundreds of μm. If the signal channels are too narrow then the reagents diffuse away and a pulse of excitation dies fast. Therefore, the spatial scale of the devices can be scaled down only if the medium used allows for it.

In the paper we have described the response of signal processing devices to a single set of input signals, which corresponds to the simplest binary information coding. It seems more interesting to consider another types of coding, based on trains of spikes, as it looks more suitable for excitable medium. Such coding may be linked with a multiple-valued logic. The fact that signals can be significantly changed after propagation through gaps [17] gives us another level of complexity that can be used in signal processing with reaction-diffusion medium. We hope to discuss some aspects of this problem in near future using numerical simulations. Experiments with trains of pulses corresponding to logical states using Ru-catalyzed BZ reaction become difficult because the processes in the chemical medium are quite slow (the speed of excitation pulses on a membrane is of the

order of millimeters per minute) and it is hard to keep the system at a stationary condition for a long time. In this respect the semiconductor information processing medium with reaction-diffusion dynamics of carriers [2,18] looks more promising for the experimental realization of the considered devices.

References

1. Adamatzky, A., Costello, D. B., Asai, T.: *Reaction-Diffusion Computers*. Elsevier Science (2005)
2. Asai, T., Nishiyama, Y., Amemiya, Y.: A CMOS reaction-diffusion circuit based on cellular-automaton processing emulating the Belousov-Zhabotinsky reaction. IEICE Trans. Fund. **E85-A** (2002) 2093–2096
3. Motoike, I., Yoshikawa, K.: Information Operations with an Excitable Field. Phys. Rev. E. **59** (1999) 5354–5360
4. Motoike, N. I., Yoshikawa, K., Iguchi, Y., Nakata, S.: Real-Time Memory on an Excitable Field. Phys. Rev. E. **63** (2001) 036220(1–4).
5. Gorecki, J., Yoshikawa, K., Igarashi, Y.: On chemical reactors that can count. J. Phys. Chem. A **107** (2003) 1664–1669
6. Gorecka, J., Gorecki, J.: T-shaped coincidence detector as a band filter of chemical signal frequency. Phys. Rev. E. **67** (2003) 067203(1–4)
7. Motoike, N. I., Yoshikawa, K.: Information operations with multiple pulses on an excitable field. Chaos, Solitons & Fractals. **17** (2003) 455–461
8. Nagahara, H., Ichino, T., Yoshikawa, K.: Direction detector on an excitable field : Field computation with coincidence detection. Phys. Rev. E. **70** (2004) 036221(1–5)
9. Motoike, N. I., Adamatzky, A.: Three-valued logic gates in reaction–diffusion excitable media. Chaos, Solitons & Fractals. **24** (2005) 107–114
10. Gorecki, J., Gorecka, J., Yoshikawa, K., Igarashi, Y., Nagahara, H.: Sensing the distance to a source of periodic oscillations in a nonlinear chemical medium with the output information coded in frequency of excitation pulses. Phys. Rev. E. **72** (2005) 046201(1–7)
11. Adamatzky, A.: Programming Reaction-Diffusion Processors. Banatre J.-P. et al. (Eds.): LNCS 3566 (2005) 31–44, Springer, Berlin.
12. Agladze, K., Aliev, R. R., Yamaguchi, T., Yoshikawa, K.: Chemical diode. J. Phys. Chem. **100** (1996) 13895–13897
13. Brandstadter, H., Braune, M., Schebesch, I., Engel, H.: Experimental study of the dynamics of spiral pairs in light-sensitive Belousov-Zhabotinskii media using an open-gel reactor. Chem. Phys. Lett. **323** (2000) 145–154
14. Sendina-Nadal, I., Mihaliuk, E., Wang, J., Perez-Munuzuri, V. and Showalter, K.: Wave propagation in subexcitable media with periodically modulated excitability. Phys. Rev. Lett. **86** (2001) 1646–1649.
15. Sakurai, T., Mihaliuk, E., Chirila, F., Showalter, K.: Design and control of wave propagation patterns in excitable media. Science **296** (2002) 2009–2012.
16. Adamatzky, A.: Collision-based computing in Belousov–Zhabotinsky medium Chaos. Solitons & Fractals **21** (2004) 1259–1264.
17. Sielewiesiuk, J., Gorecki, J.: Complex transformations of chemical signals passing through a passive barrier. Phys. Rev. E. **66** (2002) 016212(1–9).
18. Gossen, C., Niedernostheide, J. F., Purwins, G. H.: Pattern Formation of the Electroluminescence in AC ZnS:Mn Devices. *Nonlinear Dynamics and Pattern Formation in Semiconductors and Devices, Springer Proceedings in Physics* **79** (1995) 112–132 Springer, Berlin.

Flexible Versus Rigid Tile Assembly

Nataša Jonoska and Gregory L. McColm

Department of Mathematics University of South Florida Tampa, FL 33620
{jonoska, mccolm}@math.usf.edu

Abstract. DNA molecules have been assembled in rigid DX and TX molecules, arrayed in assemblies similar to Wang tiles, and, as flexible branched junction molecules with flexible arms have been used in assemblies representing arbitrary graphs. This paper considers both models of rigid and flexible tiles. A model representing complexes assembled out of rigid tiles based on tile displacements is presented. This presentation is used to simulate computations obtained from (bounded) rigid tile self-assembly by corresponding assemblies of flexible tiles.

1 Introduction

While most work in DNA nanotechnology of array assemblies and computation has been done with tiles with rigid arms [16,10], a variant using "flexible tiles" with flexible arms has recently appeared in construction of arbitrary graphs [12,5], with somewhat different properties and computational power. Very roughly, rigid tile computation is associated with all recursive functions [15], while flexible tile computation is associated with nondeterministic polynomial time computable functions [8].

In order to make a direct comparison of their computational power and their ability to directly simulate each other, we need mathematical descriptions of the assembled complexes. In this note, we use flexible tile assembly to simulate rigid tile assembly, and in order to do this, we devise a mathematical formalization of rigid tile assemblages. The rigid tile model presented here is different than the models considered in [14,11], where the bonding strength of each joining port is essential. In this paper we concentrate on the geometry of the final product of the assembly as the shape and architecture of the (possible) complete complex emerges. The order of assembly of the tiles is not of essence.

Molecules and other nanostructures have been modeled with tinkertoy-like assemblies for a century now, but there is still little formal apparatus to describe these assemblages. Both the connections between individual components and the macroscopically observable phenomena that these structures generate have been modeled using highly sophisticated mathematics, but the structure of the assembly itself has been largely ad hoc. Now that complex (nano-) assembly has become one of the pre-eminent areas of scientific research ("chemical self-assembly" was one of the top 25 open questions listed recently in *Science* [13]), there should be considerable demand for a formal system describing the complexes experimentalists construct.

C.S. Calude et al. (Eds.): UC 2006, LNCS 4135, pp. 139–151, 2006.

The method developed in this note for identifying the complete complexes is based on displacement and rotation transformations. It describes paths in a complex from one rigid tile to another. This description can be encoded into flexible tiles so that, within certain restrictions, a (sufficiently small) complete flexible tile complex is possible if and only if a (sufficiently small) rigid tile complex is possible.

Since the vehicle for this paper is the representation of rigid tile assembly by flexible tile assembly, we first outline the origins and interest behind the latter model. The flexible tile model was introduced in [7] with the goal of more readily generating complexes in the form of arbitrary graphs. Most extant models were based on tiling the planes, and thus were usually regular or domino-like planar tiles, and thus could not be used to, say, construct copies of non-planar graphs. With non-planarity in mind, the flexible tile model was used to model the assembly of non-planar constructs. One way to look at the flexible tile model is as a variant of the rigid tile model, as follows. While a (rigid) tile would be a starfish-shaped object, with rigid arms extending in various directions from the center, and connecting to other rigid tiles at ports (often called "sticky ends") on the tips of the arms, a flexible tile is more like an octopus, with flexible tentacle-like arms attached to the center, and connecting to other flexible tiles via ports at the tips of the tentacles. A graph might be difficult to construct out of starfish – one must worry about the geometry of the arms – but it is not difficult to arrange complementary tentacles to connect up in an arbitrary graph structure. This is illustrated in Figure 1.

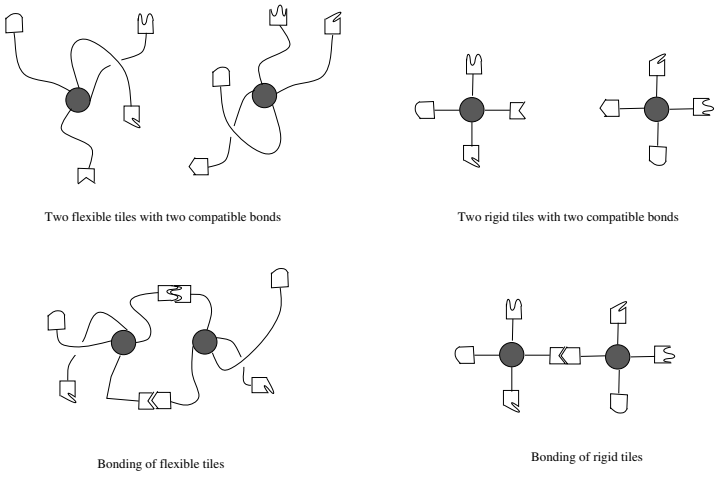


Fig. 1. Flexible vs rigid tile connections

There is one important distinction between the rigid tile and flexible tile models, as currently extant. (We use the term "query" to represent the formalization of the question that a given computation is to answer; in this paper, this answer

is encoded in the existence and structure of (appropriate) complete complexes.) In the rigid tile model, for any one query, one uses a fixed set of types of tiles, and the input is used to generate a seed that grows into a complete complex, which represents the output. In the flexible tile model, the query has an associated program which takes an input and generates a set of types of tiles, and the output is represented by a "sufficiently small" complete complex (if any).

In this paper, we present a method that describes a complete complex of rigid tiles by giving the relative positions and orientations of each tile, starting from a given tile in a "standard position." We prove that within certain restrictions, a flexible tile assembly can simulate a rigid tile assembly, provided that the latter is not "too large"; this size restriction arises from the different computational powers of these two models. To prevent superpositions of tiles in rigid tile assembly, we prevent the centers of two rigid tiles from being in the same position. For simplicity, and following contemporary lines of research, as the pre-eminent algorithmic DNA array assembly uses Wang or Wang-like tiles which align in a plane, we restrict our attention to the two-dimensional case, although we should remember that a main motivation for developing the flexible tile model was to model three-dimensional structures.

2 The Flexible Tile Model

In this section we describe the basics of the flexible tile model that is used for simulating the rigid tile model. For more information on this model, see [8]. As this paper deals with models of rigid tiles and of flexible tiles, we attach the prefix "flex" to the words "tile," "complex," "pot," etc., to distinguish the flexible versions from the rigid tile versions. The port types are the same for rigid and flexible tiles.

Definition 1. *Fix a set H of* port types*; this can be regarded as a set of words or codes that we use for tiles or flex tiles. Fix a function $h \mapsto \hat{h}$ on H so that $h \neq \hat{h}$ and $\hat{\hat{h}} = h$ for all $h \in H$.*

A flex-tile has ports at the ends of its flex-arms, labeled by port types so that two flex-arms can join at ports if and only if one of the arms terminates in a port of type h and the other terminates in a port of type $\hat{h}$. For the rest of this article, take H and $\hat{}$ to be fixed as above.

Definition 2. *A* flex-tile type *is a function $\tau\colon H \to \mathbb{N}$ such that for any $h \in H$, $\min\{\tau(h), \tau(\hat{h})\} = 0$. A* flex-pot type *over H is a set of flex-tile types on H.*

Thus a flex-tile type is determined by the number of ports of each type the tile has; note that a flex-tile type does not admit flex-arms with complementary port types. And a flex-pot type is determined by the types of flex-tiles that are present.

A *flex-tile* is a flexible tile of some flex-tile type; that type determines how many ports of each port type the flex-tile has. A pot is a (presumably large) set

of tiles, and is of some flex-pot type that determines what types of flex-tiles are present. These flex-tiles can combine to form flex-complexes.

We represent flex-complexes with petri graphs. Recall that a petri graph is a graph with two kinds of vertices, primary or "place" vertices and secondary or "transition" vertices, the latter of which always has at most two incident edges. We use a petri graph to represent a complex of flex-tiles, with flex-tiles represented by primary (place) vertices, and junctions (between two ports on ends of flex-arms) represented by secondary (transition) vertices.

Definition 3. *Given a flex-pot type* $\mathbf{P}$*, a* flex-complex over $\mathbf{P}$ *is a tuple* $\langle T, J, E\lambda\rangle$*,* $(T \cap J = \emptyset)$ *such that:*

- *The tuple* $\langle T, J, E, \rangle$ *is a connected petri graph, where:*
  - T *is the set of primary vertices, which we call* tile vertices*, and*
  - J *the set of secondary vertices of degree at most two, which we call* junction vertices*, and*
  - $E \subseteq \{\{t, j\}: t \in T \ \&\ j \in J\}$ *is the set of* edges*.*
- *The function* λ *assigns types to vertices and edges as follows:*
  - *For each* $t \in T$*,* $\lambda(t) \in \mathbf{P}$*, thus assigning flex-tile types to tile vertices, and*
  - *For each* $e \in E$*,* $\lambda(e) \in H$*, and*
  - *For each* $t \in T$ *and* $h \in H$*, there are exactly* $\lambda(t)(h)$ *edges* e *incident to* t *such that* $\lambda(e) = h$*, and*
  - *For each* $j \in J$*,* $\lambda(j) = \{\lambda(e): e \in E \wedge j \in e\}$*.*

We say that a junction vertex of degree one represents a free port. If a complex has no free ports, call it complete.

The question is: given a flex-pot type, can one construct a complete flex-complex, not too large, out of flex-tiles of types from that flex-pot? More precisely, from [8]:

Definition 4. *We define* Flexible Tile Assembly Polynomial time *(or* FTAP*) as the set of all queries* Q *that can be computed as follows. There exists a polynomial-time algorithm which, given input* I *of size* $|I|$*, can enumerate the flex-tile types of a flex-pot type* $\mathbf{P}(I)$ *and compute a (polynomial) bound* $b = b(|I|)$ *such that* $I \in Q$ *if and only if there exists a complete flex-complex of at most* b *flex-tiles over* $\mathbf{P}(I)$*.*

Fix a polynomial f so that the polynomial time algorithm that converts I into a flex-pot type is such that the flex-pot has no more than $f(|I|)$ tiles, each with no more than $f(|I|)$ tentacles. It was shown in [8] that FTAP captures NPTIME.

3 The Rigid Tile Model

A rigid tile is centered at a single point in space, with rigid arms extending from its center to ports at the ends of the arms. Thus each arm fixes the vector from the center of the tile to the port at the end of the arm. The tile may be moved by displacing (translating) the center (and thus the entire tile) by a vector; this

motion can be obtained by applying the affine operator for that displacement. The tile can also be rotated about its center through some angle about that central point. This motion can be obtained by applying the corresponding rotation operator for that angle. These operators, and their compositions, are *isometries*, i.e., operators that describe strictly rigid motions. For brevity, we make three simplifications:

- We consider assembly of two-dimensional tiles into complexes in the plane.
- We have a simple superposition restriction: we require that no two tiles have their centers in the same position.
- We have a simple articulation restriction: two tiles can connect at a junction of two arms only if they are attached at the tips, with the two arms aligned.

Relaxing the restrictions usually results in a more complicated but similar situation. For example, we would get the same results if we permitted tiles to articulate via non-aligned arms – but the resulting nomenclature would be more involved.

Similarly as with flex-tiles, rigid tiles have a center and arms that end with ports. If a port type is denoted as h, its complement is $\hat{h}$. The set of port types is H. We start by describing what a tile looks like in a "standard position." It has several arms, where an arm is described as a pair, a vector from the center to its tip, and the type of the port at its tip. To move a tile one can imagine that one has rotated it by an angle, and displaced it by a vector $\mathbf{x}$; thus in Figure 2, a tile in standard position has been rotated and displaced in order to attach to another tile.

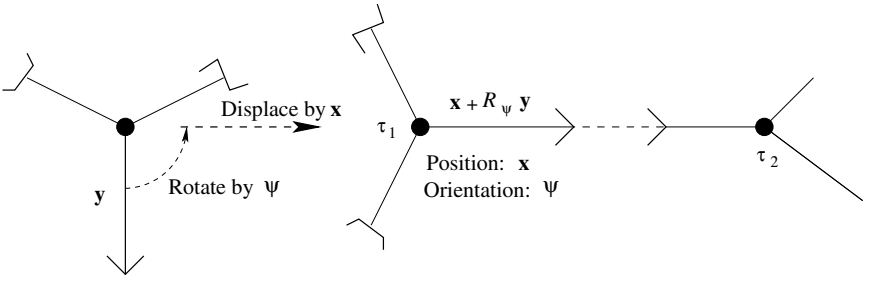


Fig. 2. To connect, the tile in a standard position (left) is rotated by ψ and displaced by a vector $\mathbf{x}$. The arm originally pointing downwards is aligned with an arm of the other tile (right); the two arms can attach as their ports are of complementary port types (indicated by the shape). Here, R_ψ is the rotation operator by angle ψ.

Thus the tile itself can be characterized by the vectors for its arms, and types of the ports at the tips of those arms.

Definition 5. *An* arm type *is a pair* $(\mathbf{y}, h)$*, where* $\mathbf{y}$ *is a vector and* h *is a port type. A* tile *with* k *arms is a set* $\tau = \{(\mathbf{x}, \psi), (\mathbf{y}_1, h_1), \ldots, (\mathbf{y}_k, h_k)\}$ *where* $(\mathbf{x}, \psi)$ *is called* location *of* τ *and each* $(\mathbf{y}_i, h_i)$ *is an arm type such that no two arm types have vectors* $\mathbf{y}, \mathbf{y}'$ *with* $\mathbf{y} = \alpha\mathbf{y}'$ *for* $\alpha > 0$*. A tile is* in standard position *if* $(\mathbf{x}, \psi) = (\mathbf{0}, 0)$*. A* tile type *is a tile in a standard position. A* pot type *is a set of tile types.*

Note that in this case a tile can have ports of complementary types h and $\hat{h}$. This is not permitted in flex-tiles as the two complementary ports would (presumably) join, obviating their utility. But as two complementary ports on a rigid tile cannot join (their arms being rigid), this restriction is omitted.

The location $(\mathbf{x}, \psi)$ of a tile τ says that the tile was moved from the standard position, by rotating counterclockwise through angle ψ, and then translating by vector $\mathbf{x}$. If R_ψ is the vector operation of the rotation, and the tile of type τ is at location $(\mathbf{x}, \psi)$, then the port on an arm of type $(\mathbf{y}, h)$ is at the position $\mathbf{x} + R_\psi \mathbf{y}$.

Example 1. Consider a pot type $P = \{\tau\}$ with one tile-type $\tau = \{(\mathbf{0}, 0), E = (\langle 1, 0\rangle, h_1), W = (\langle -1, 0\rangle, \hat{h}_1), N = (\langle 0, 1\rangle, h_2), S = (\langle 0, -1\rangle, \hat{h}_2)\}$. Here we name the arms of τ: E, W, N, S. Then tiles from P assemble into an infinite grid as in Figure 3(a), and if one tile is in standard position, with its center at the origin, the other tiles can be taken at positions $(2i, 2j)$, where i and j are integers.

In Example 1, if we had an infinite complete complex generated from a tile in standard position, then each of the other tiles is of the same type, but displaced by a vector $\langle 2i, 2j\rangle$. Note that a tile rotated by 90°, 180°, or 270° will not fit in the grid; thus all tiles are rotated by integral multiples of 360°.

Example 2. Suppose that we have two kinds of tiles, both consisting of two arms of the same length, where the angle between the arms is 240°. The ports of the arms within one tile are the same: in one tile type, both ports are of type h, while for the other of tile type, both ports are of port type $\hat{h}$. When they assemble, they alternate between one type of tile and the other, with all tiles of the second type rotated 120° with respect to the second as in Figure 3(b).

The complete complex in Example 2(b) has the form of a hexagon. We fix the standard positions for these tiles. Let $\mathbf{a} = \langle \frac{1}{2}, \frac{\sqrt{3}}{2}\rangle$ and $\mathbf{b} = \langle \frac{1}{2}, -\frac{\sqrt{3}}{2}\rangle$. Then the two tile types are: $\tau = \{(\mathbf{0}, 0), a = (\mathbf{a}, h), b = (\mathbf{b}, h)\}$ and $\tau' = \{(\mathbf{0}, 0), a' = (-\mathbf{b}, \hat{h}), b' = (-\mathbf{a}, \hat{h})\}$. The arms of τ are denoted a and b and the arms of τ' are a' and b'. Then if the tile at the left-most of the hexagon (indicated in Figure 3(b) as t_0) is in standard position, with its center at the origin, since the sides of the hexagon are of length 2 the two tiles at the bottom have their centers at $(1, -\sqrt{3})$ and $(3, -\sqrt{3})$ respectively. Hence the "complete complex" will have "tile vertices" at positions $(0, 0)$, $(1, -\sqrt{3})$, $(3, -\sqrt{3})$, $(4, 0)$, $(3, \sqrt{3})$, and $(1, \sqrt{3})$.

The (rigid) complex structure with a base tile t_0 is defined as follows.

Definition 6. *Given a pot type* $\mathbf{P}$*, a* complex over $\mathbf{P}$ *is a tuple* $\langle T, t_0, J, E, \lambda\rangle$*,* $(T \cap J = \emptyset)$ *such that:*

- *The tuple* $\langle T, J, E\rangle$ *is a connected petri graph, where:*
  - T *is the set of primary vertices, called* tile vertices*,*
  - J *is the set of secondary vertices of degree at most 2, called* junction vertices*,*
  - $E \subseteq \{\{t, j\} : t \in T \ \&\ j \in J\}$ *is the set of* edges*.*
- *The function* λ *is a labeling of the vertices and the edges:*

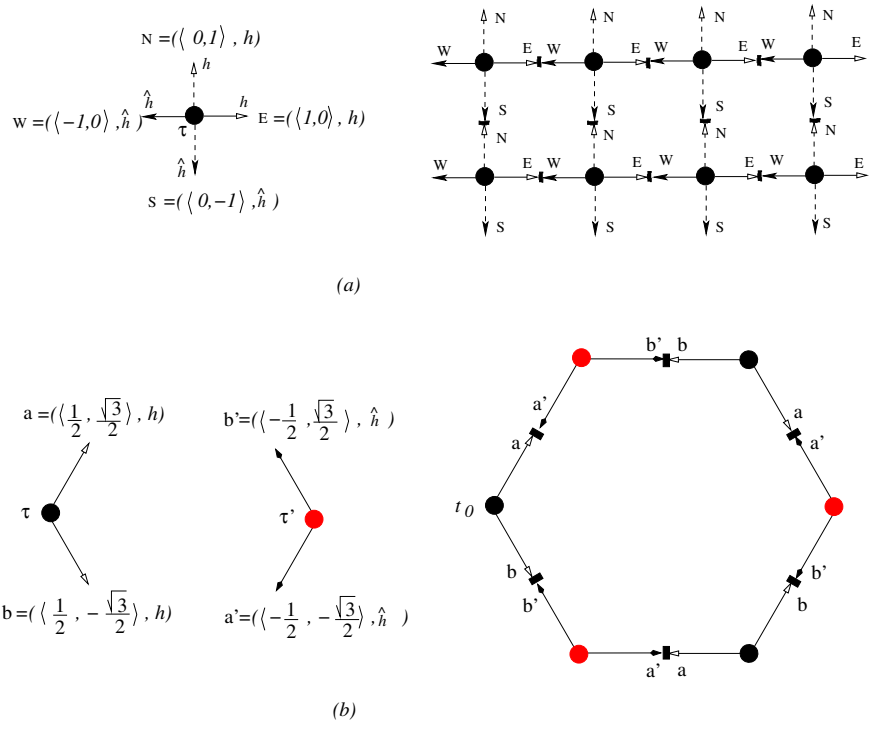


Fig. 3. Two examples. (a) Square tiles forming a grid, and (b) angled tiles forming a hexagon. The port types on the tips of the arms of the angled types are represented by solid (for type $\hat{h}$) versus unfilled (for port type h) arrows.

- *For each $t \in T$, let $\lambda(t) = (\mathbf{x}, \varphi, \tau)$, $\tau \in \mathbf{P}$ and $(\mathbf{x}, \varphi)$ is a location. There is exactly one edge incident to t for each arm type in τ.* [each tile vertex corresponds to a tile type; the label indicates the location and the tile type of the vertex]
- *For each $e = \{t, j\} \in E$, if $\lambda(t) = (\mathbf{x}, \varphi, \tau)$ then $\lambda(e) \in \tau$*
- *For each $j \in J$, if j is incident to two edges $e = \{t, j\}$ and $e' = \{t', j\}$ with labels $\lambda(t) = (\mathbf{x}, \varphi, \tau)$, $\lambda(t') = (\mathbf{x}', \varphi', \tau')$, $\lambda(e) = (\mathbf{y}, h)$ and $\lambda(e') = (\mathbf{y}', h')$ then $h' = \hat{h}$ $\mathbf{x}' - \mathbf{x} = R_\varphi \mathbf{y} - R_{\varphi'} \mathbf{y}'$* [the arms form two edges connecting the tile vertices via the junction vertex j] *and for some $\alpha < 0$, $R_\varphi \mathbf{y} = \alpha R_{\varphi'} \mathbf{y}'$* [the arms are antiparallel, and so aligned].

– *The tile t_0 is called the* base *tile and it has label $\lambda(t_0) = (\mathbf{0}, 0, \tau)$, where $\tau \in \mathbf{P}$*

Similarly as for flex-tiles, the junction vertices of a complex with degree one are called *free* and the junction vertices with degree two are *complete*. A complex is *complete* if all junction vertices are complete.

In Example 1, the infinite grid has, for each $t \in T$, $\lambda(t) = (\langle 2i, 2j\rangle, 0, \tau)$, where i and j are integers and τ is the sole tile type. The labels of the edges are N, S, W or E as defined through τ.

Two complexes $\mathcal{C}$ and $\mathcal{C}'$ over the same pot type are *isomorphic* if there is a one to one correspondence $T \to T'$, $J \to J'$, $E \to E'$ that preserves the labeling of the tile vertices and the edges.

To each complex we associate a set of words in the following way. Given a pot type $\mathbf{P}$ with tile types $\tau_1, \ldots, \tau_k$, for each tile type τ_i denote the arms $a_{i1}, \ldots, a_{ks_k}$ $(i = 1, \ldots, k)$ and put all these arm types into a set A_i. Consider the alphabet $\Sigma = \Sigma_{\mathbf{P}} = \cup_{i=1}^{k} A_i$. Let $\mathcal{C} = \langle T, t_0, J, E, \lambda\rangle$ be a complex over $\mathbf{P}$. Then for each edge $e \in E$, $\lambda(e)$, the label of e, is in Σ. Consider the labels of all paths in $\mathcal{C}$ that start at vertex t_0. This set of words in fact uniquely defines $\mathcal{C}$. Denote with $L(\mathcal{C})$ the set of all words that are labels of paths in $\mathcal{C}$ and start at t_0.

Theorem 1. *Let $\mathcal{C}$ and $\mathcal{C}'$ be two complexes over a pot type $\mathbf{P}$, and let t_0 and t_0' be base tiles of the same type in $\mathcal{C}$ and $\mathcal{C}'$, respectively. Suppose that each complex has at most n tile vertices. Then $\mathcal{C}$ is isomorphic to $\mathcal{C}'$ through an isomorphism that preserves vertex and edge labels if and only if $L(\mathcal{C})^{\leq 2(n-1)} = L(\mathcal{C}')^{\leq 2(n-1)}$.*

Proof. (sketch) Note that a path that visits n tile vertices in a complex, visits $n-1$ junction vertices and hence traverses $2(n-1)$ edges. Since a complex is connected, the set of all paths of length $2(n-1)$ includes all paths such that the n tile vertices are visited at least once. Let $\mathcal{C}$ with base tile t_0 and $\mathcal{C}'$ with base tile t_0' be two complexes with n vertices over the same pot type $\mathbf{P}$. The labels of the edges belong to the same alphabet $\Sigma_{\mathbf{P}}$. We show that $\mathcal{C}$ and $\mathcal{C}'$ contain identically labeled arrays of tiles with isomorphic displacements. We proceed by induction on n. If $n = 0$ then by assumption t_0 and t_0' are tiles of the same type. If $\mathcal{C}$ and $\mathcal{C}'$ contain $n+1$ tile vertices, let $t \neq t_0$ be a boundary tile vertex in $\mathcal{C}$. Consider $\bar{\mathcal{C}}$ a complex obtained from $\mathcal{C}$ with t removed. Consider $K = L(\bar{\mathcal{C}})^{\leq 2(n-1)}$. By assumption $K \subseteq L(\mathcal{C}')$ and hence it defines a subcomplex $\bar{\mathcal{C}}'$ of $\mathcal{C}'$ with n tile vertices isomorphic to $\bar{\mathcal{C}}$. Extension of words in K by two symbols that are labels of paths reaching t are in both $L(\mathcal{C})$ and $L(\mathcal{C}')$ and hence define the extension isomorphism map from $\mathcal{C}$ to $\mathcal{C}'$.

In Example 2, the hexagon is uniquely defined with $\mathrm{Pref}(w, w^R)$ for $w = aa'b'baa'b'baa'b'b = (aa'b'b)^3$ where Pref indicates the set of all prefixes. Note that subwords aa, ab, ba, bb are forbidden in any complex over this pot, as arms denoted a and b have the same port h and cannot meet. Similarly with arms a' and b'. Thus in every complex over $\mathbf{P}$ the words alternate between primed symbols and non primed symbols.

In Example 1 the complex is potentially infinite and there is no complete complex over any finite pot. The four-by-two grid indicated in Figure 3(a) can be defined with the set $\mathrm{Pref}(w)$ where w is a label of a path starting from the bottom left corner and having at most length $14 = 2(8-1)$ (for ex. $w = EWEWEWNSWEWEWE$).

The rigid tile model for computation is formalized as follows.

Definition 7. *(Rigid tile computation) For a query Q construct a set of tile types $\mathbf{P}$ and a polynomial time algorithm such that for any input I, the algorithm produces a (description of) a partial complex – a seed $s(I)$ encoding the input I. The query Q is true if and only if the seed extends to a complete complex.*

It is known (in case of DNA tile assembly see [15]) that even in the simple case of Wang tiles (four armed tiles in the plane) the queries computable by the above procedure are precisely the classically computable queries. In the next section we describe how flex-tile assemblies can simulate the rigid tile assemblies, given that the rigid tile assemblies generate complexes within a certain bound.

4 The Simulation

Most contemporary DNA assembly concerns algorithmic assembly of rigid Wang tiles (for ex. [10,2,6]) that are designed as double or triple cross-over DNA molecules. These simple square-shaped tiles define models capable of universal computation and their mathematical description within the above described concept is straightforward. In this section we restrict attention to assemblies (of polynomial size) of Wang tiles and show that such assemblies can be simulated as assemblies of flex-tiles. Wang tiles are unit squares in the plane with colored sides such that same colored sides can be aligned. Wang tiles are not allowed to rotate, and can only translate. Each type of Wang tile comes with arbitrary large number of copies of itself. It is well known that the question of whether a finite set of Wang tile types can be aligned such that they tile the whole plane is undecidable. and it is precisely from this fact that their universal computational power can be proven.

Each Wang tile can be represented as a four-armed rigid tile (North, South, East, West). As rotations are not allowed, East can connect only with West and South only with North.

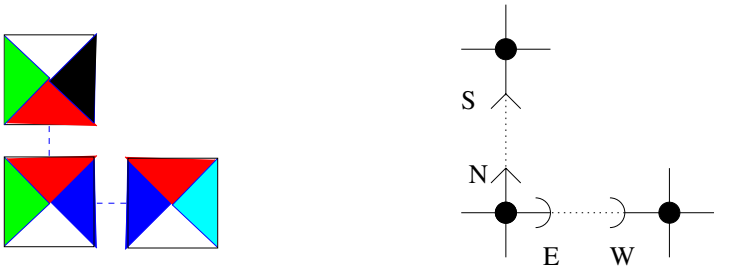
A Wang tile with left color l, right color r, top color t and bottom color b is represented with a rigid tile type

$$\tau = \{(\mathbf{0}, 0), (\langle -1, 0\rangle, lW), (\langle 1, 0\rangle, rE), (\langle 0, 1\rangle, tN), (\langle 0, -1\rangle, bS)\}$$

where $\hat{S} = N$ and $\hat{W} = E$.

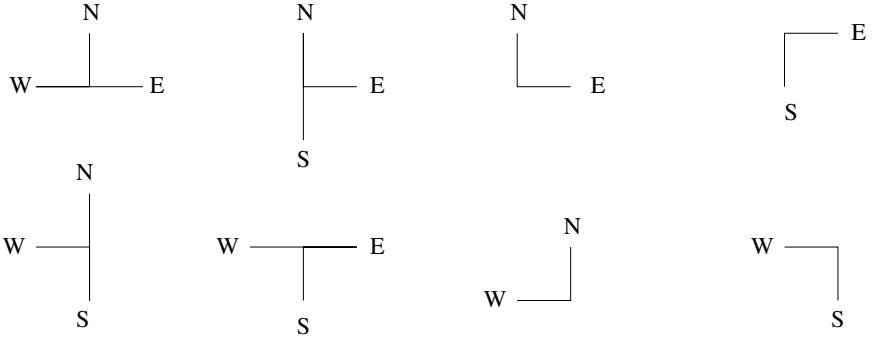
In order to obtain complete complexes, additional border tiles are needed. To form boundaries, we allow Wang tiles with one or two (adjacent) uncolored sides, and require that Wang tiles do not join across uncolored sides. This proviso corresponds to adding eight types of rigid tiles, as in Figure 5.

Theorem 2. *Given a pot type of rigid Wang tiles $\mathbf{P}_r$, there is a polynomial time algorithm and a polynomial q such that given an input seed of rigid tiles with n tiles, the algorithm can construct a flex-pot type $\mathbf{P}_f$ with the property that $\mathbf{P}_f$ admits a complete complex with $p(n) = (2q(n) - 1)^2$ flex-tile vertices if and only if the seed extends to a complete complex over $\mathbf{P}_r$ with at most $q(n)$ tile vertices.*



(a) Wang tiles (b) Square tile representation

Fig. 4. (a) Wang tiles assembly, (b) representing Wang tile assembly using four armed rigid tiles similar as in Example 1



(a) Side boundary tiles (b) Corner boundary tiles

Fig. 5. The boundary tiles

Proof. (sketch) As flexible tiles do not follow an underlying geometry and their connections may result in a complex that lives in the 3-dimensional space instead of the plane like Wang tiles, the flex-tiles should be indexed with their intended positions, and ports encoded such that only flex-tiles labeled with adjacent positions can join. We use the technique introduced in [8]: index the positions of the flex-tiles, and order the indices so that no two tiles of the same position index can appear in the same "sufficiently small" flex-complex. Note that the shape of the rigid complete complex might not be rectangular. We enclose a flex-tile representation of this shape into a rectangular grid of no more than $(2q(n) - 1) \times (2q(n) - 1)$ flex-tiles; see Figure 6.

Since we are only interested in extending the seed of n tiles to a rigid complex of at most $q(n)$ tiles, if we take one of the tiles in the seed to be in the standard position, then as the resulting complex is connected, every other tile in the ultimate complex must be at a position $(2i, 2j)$ in which $2i + 2j < 2q(n)$. Thus we order the positions to that every tile of the ultimate complex will be at a position enumerated during the first $p(n) = 4q(n)^2 - 4q(n) - 1$ positions as in Figure 6.

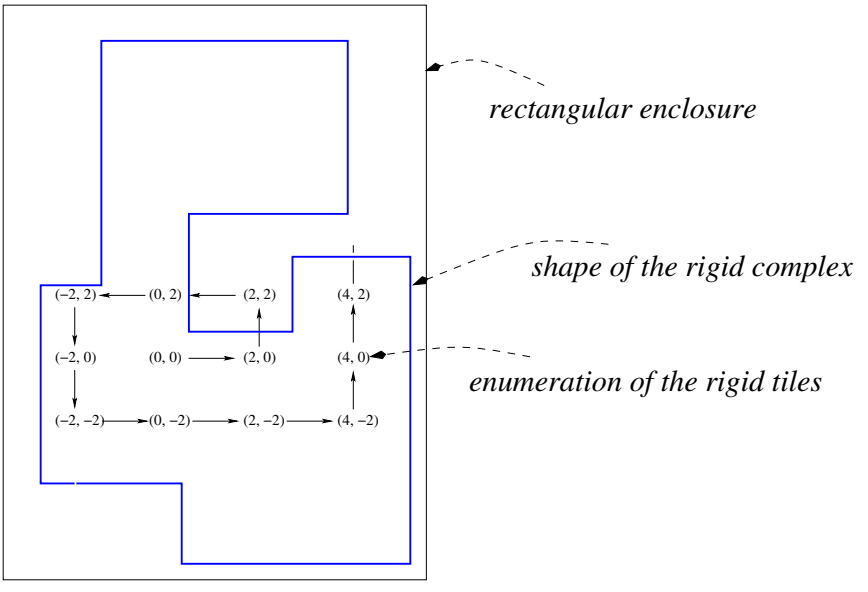


Fig. 6. The ordering of positions

Notice that in Figure 6, $\mathrm{succ}(0,0) = (2,0)$, $\mathrm{succ}^2(0,0) = (2,2)$, $\mathrm{succ}^3(0,0) = (0,2)$, and so on, and all positions within $q(n)$ displacements from the origin are among $(0,0)$, $\mathrm{succ}(0,0)$, $\mathrm{succ}^2(0,0)$, ..., $\mathrm{succ}^{p(n)-1}(0,0)$. Thus any complex of at most $q(n)$ Wang tiles, in which one of those tiles is in standard position, will have all of its tiles at positions among $(0,0)$, ..., $\mathrm{succ}^{p(n)-1}(0,0)$.

The flex-tile types for the flex-pot that simulates the given (rigid tile) pot are constructed as follows. There are two kinds of flex-tile types: those that represent (rigid) tiles in particular positions, and those that represent unoccupied positions, positions in the rectangular enclosure outside the shape of the complex.

Consider flex-tile types representing (rigid) tile types of tiles at particular positions, both inside and outside the seed. For brevity, we explain a flex-tile type representing a typical (four-armed) square tile at a particular position, with particular port types. The cases of the eight (kinds of) boundary tiles can be described similarly. As in Figure 7(a), suppose that a four-armed tile is to be placed at $(2i, 2j)$ within the rigid complex. This tile is simulated with a flex-tile of 6 flex-arms as in Figure 7(b). Four of the arms encode the port types the position $(2i, 2j)$; the additional two arms follow the spiral ordering of the positions; this spiral is used to count the number of rigid tiles represented in the shape on the flex-tile complex, as follows.

As the "shape" of the flex-complex representation of the rigid tile complex (see Figure 6) takes up at most $q(n)$ flex-tiles, the flex-complex will include at least $p(n) - q(n)$ "blank" flex-tiles, representing unoccupied positions. In the flex-complex, there will be exactly one flex-tile for each position up to $\mathrm{succ}^{p(n)-1}(0,0)$. As the rigid tile complex being simulated has at most $q(n)$ tiles,

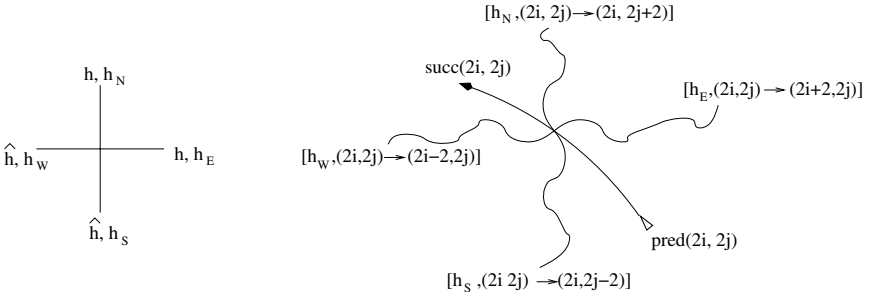


Fig. 7. Simulating rigid tiles. The rigid tile at left is at position $(2i, 2j)$, so that its four neighbors are at $(2i, 2j+2)$, $(2i+2, 2j)$, $(2i, 2j-2)$, and $(2i-2, 2j)$. The flex-tile which simulates this rigid tile has those four articulating positions built into its port codes. It will only join with flex tiles labeled with those positions. Here "$(i, j) \to (i', j')$" means that the port is on a flex-arm of a flex-tile presumably at (i, j), and it wants to join to a port on a flex-arm of a flex-tile presumably at (i', j'). In addition, the two extra arms connect to arms for flex tiles at $\text{pred}(2i, 2j)$ and $\text{succ}(2i, 2j)$.

they occur within the rectangle of size $(2q(n) - 1) \times (2q(n) - 1)$. The flex-tiles representing unoccupied positions occur in the parts outside the "shape" of the rigid tile complex. Such a "blank" flex-tile for the position $(2i, 2j)$ has only two flex-arms, one to connect to the spiralling arm of the flex-tile at $\text{pred}(2i, 2j)$, and the other to the spiralling arm of the flex-tile at $\text{succ}(2i, 2j)$.

Thus starting at the flex-tile labeled with the base tile's position $(0, 0)$, the spiral traverses the entire flex-complex, and the (labels on the) successive spiraling arms count the number of flex-tiles representing rigid tiles in the "shape," as opposed to blank flex-tiles representing unoccupied positions outside the "shape." Thus, if all possible final $p(n)$th flex-tile types for the position $\text{succ}^{p(n)-1}(0, 0)$ have a single (predecessor) spiral arm that connects only if the count is at most $q(n)$, the flex-complex can be completed if and only if the (rigid tile) complex represented has at most $q(n)$ tiles.

5 Excelsior

The rigid tile model here is intended to describe finished structures, and this paper being a first iteration, we are not concerned with the various kinds, strengths, and properties of bonds. Our concern is geometric, for while most extant DNA assembly has relied on a limited array of (largely two-dimensional) tiles, other fields of chemical assembly has explored problems involving a variety of two and three-dimensional ... tiles ... (or, as a chemist might prefer, "molecular building blocks" (MBBs): see, e.g., [17], [3], [4], and especially [1] on "crystal engineering"). (Hence the motivation for the geometric generality of our model: there is already a large range of building block shapes to accomodate.)

But in this project, our concern is primarily computational: we have two related models of computation, and we have a class of queries of the form "given

building blocks of such-and-such form, a completed complex exists." We have shown that any such query on the two-dimensional rigid tile model can (within polynomial restrictions) have its assembly modeled by the flexible tile model. The converse holds as well: given any such query, the flexible tile assembly can be simulated by a rigid tile assembly [9].

Acknowledgement. This work has been supported in part by NSF grants CCF #0432009 and CCF#0523928.

References

1. *Cambridge Structural Database*, Cambridge Crystallographic Data Centre, on-line at <http://www.ccdc.cam.ac.uk/>.
2. A. Carbone, N.C. Seeman, *Molecular tiling and DNA self-assembly*, in *Aspects of Molecular Computing* (N. Jonoska, Gh. Păun, G. Rozenberg eds.), Springer LNCS **2950** (2004) 61–84.
3. G.R. Desiraju, *Crystal Engineering: the Design of Organic Solids* (Elsevier, 1989).
4. M.D. Foster, M.M J. Treacy, J.B. Higgins, I. Rivin, E. Balkovsky, K.H. Randall, *A systematic topological search for the framework of ZSM-10*, *J. Appl. Crystallography* **38** (2005), 1028–1030. This group maintains an on-line searchable database at <http://www.hypotheticalzeolites.net/>.
5. N. Jonoska, P. Sa-Ardyen, N.C. Seeman, *Computation by self-assembly of DNA graphs*, *Genetic Programming and Evolvable Machines* **4** (2003) 123–137.
6. N. Jonoska, S. Liao, N.C. Seeman, *Transducers with Programmable Input by DNA Self-assembly*, in: *Aspects of Molecular Computing* (N. Jonoska, Gh. Păun, G. Rozenberg eds.), Springer LNCS **2950** (2004) 219–240.
7. N. Jonoska, S. Karl, M. Saito, *Three dimensional DNA structures in computing, BioSystems* **52** (1999) 143–153.
8. N. Jonoska, G.L. McColm, *A Computational Model for Flexible Self-Assembling DNA Tiles*, *Proc. 4th Internat'l Conf. Unconv. Comp.* (Spring LNCS 3699, 2005), 142–156.
9. N. Jonoska, G.L.McColm, *From rigid tiles to flexible and back*, (in preparation).
10. P. Rothemund, N. Papadakis, E. Winfree, *Algorithmic Self-assembly of DNA Sierpinski Triangles*, PLoS Biology **2**(12) (2004) available at http://biology. plosjournals.org/
11. P.W.K. Rothemund, E. Winfree, *The Program-Size Complexity of Self-Assembled Squares*, *Proceedings of 33rd ACM meeting STOC 2001*, Portland, Oregon, May 21-23 (2001) 459–468.
12. P. Sa-Ardyen, N. Jonoska, N. Seeman, *Self-assembling DNA graphs*, *Natural Computing* **2** No.4 (2003) 427-438.
13. R.F. Service, *How Far Can We Push Chemical Self-Assembly? Science 309:5731* (1 July, 2005), 95.
14. E. Winfree, *Self healing tile sets* in *Nanotechnology: Science and Computation* (J. Chen, N. Jonoska, G. Rozenberg eds.) Springer-Verlag, 55–74, 2005.
15. E. Winfree, X. Yang, N. C. Seeman, *Universal computation via self-assembly of DNA: some theory and experiments*, in: L. Landweber & E. Baum, eds., *DNA based computers II* (AMS DIMACS series 44, 1998), 191–214.
16. E. Winfree, F. Liu, L.A. Wenzler, N.C. Seeman, *Design and self-assembly of two-dimensional DNA crystals*, *Nature* **394** (1998), 539–544.
17. M. J. Zaworotko, *Superstructural diversity in two dimensions: crystal engineering of laminated solids*, *Chemical Communications* (2001, Issue 1), 1–9.

On Pure Catalytic P Systems

Shankara Narayanan Krishna

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Powai, Mumbai, India 400 076
krishnas@cse.iitb.ac.in

Abstract. Catalytic P systems is one of the basic classes of P systems. The number of catalysts required for optimal universality results (both in pure catalytic systems and catalytic systems) has been a problem of extensive research [3], [5], [6], [7], [12]. The differences that can give universality/non-universality are very small in these systems, and finding this borderline is one of the 'jewel' problems in P systems [12]. In this paper, we try to figure out this borderline and have obtained some interesting results. We have proved that with 2 catalysts, if λ-rules are not used, then universality cannot be obtained. We also consider two restricted variants of pure catalytic systems and prove that they are also not universal. Finally, we look at mobile catalytic systems and solve two open problems.

1 Introduction

P systems using catalysts has come a very long way [3], [5], [6], [7], [14], [15]. Initially, catalysts were introduced in P systems as a natural variant. It was shown that to obtain universality, catalysts can be used instead of using priorities. The interest has since shifted to finding the minimal number of catalysts required for universality. Several papers [3], [4], [5], [6], [14], [15] have looked at this problem as well as restricted versions of catalytic systems, and systems with different notions of acceptance while using catalysts. Some of these include P systems with catalysts using an extended alphabet [3], pure catalytic P systems, deterministic P systems, [5], [6], P systems with catalysts that function like P automata [3] and P systems with mobile catalysts [7]. All these different versions were introduced in the search for universality results with minimal number of catalysts. Due to the vast amount of work devoted towards this basic problem, this has been called one of the jewel problems in membrane computing [12].

We quickly recall some of the most recent results in this direction. It has been shown in [3] that (i) pure catalytic systems with three catalysts and two membranes (acceptance by halting) and using λ-rules are universal, (ii) using a terminal alphabet, 3 catalysts (using λ-rules) and one membrane give universality. However, it can be easily seen that with 3 catalysts, 2 membranes, and allowing only λ-free rules, we can still achieve computational completeness. A small modification [8] of Corollary 8 [3] would do this. Thus, the power of 3 catalysts with one membrane in the non-extended case and the power of 2 catalysts

C.S. Calude et al. (Eds.): UC 2006, LNCS 4135, pp. 152–165, 2006.

with one/two membranes is still open. Similarly, in the case of mobile catalysts, it is known [7] that if rules of the kind $a \to v$ are used in conjunction with catalytic rules, then universality can be obtained with (i) two membranes and two mobile catalysts and (ii) one mobile catalyst and three membranes.

In this paper, we make an effort to solve some of the issues with respect to the number of catalysts required. We have identified some restrictions on the number of catalysts, number of membranes, as well as the kind of rules used, to obtain classes of P systems that are strictly less than RE in power. We have also obtained an optimal universality result for P systems with mobile catalysts. This paper is organized as follows: Section 2 is devoted towards prerequisites, Section 3 deals with two results on pure catalytic P systems, Section 4 is on mobile catalytic P systems.

2 Some Prerequisites

We refer to [13] for the elements of formal language theory we use here. We list a few notions and notations: $\mathbf{N}$ denotes the set of natural numbers; V denotes a finite alphabet; V^* is the is the free monoid generated by V under the operation of concatenation and the empty string denoted by λ, as unit element; by $\mathbf{N}CF, \mathbf{N}RCM(M, CF, \emptyset)$, $\mathbf{N}MAT^{\lambda}, \mathbf{N}RC_{p,f}, \mathbf{N}CS$ and $\mathbf{N}RE$ we denote the family of context-free sets, random-context matrix sets, sets of numbers computed by matrix grammars without appearance checking, random-context sets, context sensitive sets, and recursively enumerable sets of natural numbers, respectively. These can also be looked at as the family of sets of numbers recognized by these languages. It is known that $\mathbf{N}CF \subseteq \mathbf{N}RC_{p,f} \subseteq \mathbf{N}CS \subset \mathbf{N}RE$ and that $\mathbf{N}RCM(M, CF, \emptyset) = \mathbf{N}MAT^{\lambda} \subset \mathbf{N}RE$. We will be using the following well known definitions [1] in later sections of the paper.

- The *left quotient* of a language L by a letter a is given by $\partial_a^l(L) = \{x \mid ax \in L\}$.
- *Matrix Grammars*: A matrix grammar is a quadruple $G = (N, T, M, S)$ where N, T are sets of non-terminals and terminals respectively, S is the start symbol, and M is a finite set of matrices of the form $(r_1, \ldots, r_n), n \geq 1$, with context-free rewriting rules $r_i : (\alpha_i \to \beta_i,\ \alpha_i \in N,\ \beta_i \in (N \cup T)^*$. For two strings x, y we say that $x \Rightarrow y$ iff there are strings $x_0, \ldots, x_m$ and a matrix $(r_1, \ldots, r_n) \in M$ such that $x_0 = x, x_m = y$, and $x_{i-1} = x'_{i-1}\alpha_i x''_{i-1}, x_i = x'_{i-1}\beta_i x''_{i-1}$ for some $x'_{i-1}, x''_{i-1} \in (N \cup T)^*$, for all $0 \leq i \leq n-1$. In other words, a direct derivation in the matrix grammar G corresponds to applying the rules of a matrix, in order. We denote by MAT^{λ} the families of languages generated by matrix grammars (allowing λ-rules).
- *Random Context Grammars*: A random context grammar is a construct $G = (N, T, S, R)$ where N is the set of non-terminals, T is the set of terminals, S is the start symbol and R is a set of rules of the form $p : (A \to w, E_1, E_2)$ where $A \to w$ is a context-free production over $N \cup T$ and E_1, E_2 are subsets of N. Then, p can be applied to a string $x \in (N \cup T)^*$ only if $x = x_1 A x_2, E_1 \subseteq$

$alph(x_1x_2)$, and $E_2 \cap alph(x_1x_2) = \emptyset$. $alph(x_1x_2)$ stands for the set of symbols occurring in x_1x_2. If E_1 or E_2 is the empty set, then no condition is imposed by E_1 or E_2 respectively. E_1 is said to be permitting and E_2 is said to be the set of forbidding context conditions of p. We denote by $RC_{p,f}$ the family of languages generated by random context grammars with permitting and forbidding contexts and λ-free rules. If in all rules, the set E_1 is empty, we denote the family of languages generated by RC_f; similarly, if in all rules, the set E_2 is empty, the resulting family of languages is denoted by RC_p.

- *Random Context Matrix Grammars*: A random context grammar is a construct $G = (N, T, M, S, F)$ where N, T, S are as in a usual matrix grammar and M is a finite set of triples $((A_1 \rightarrow x_1, A_2 \rightarrow x_2, \ldots, A_n \rightarrow x_n), Q, R)$ where $A_i \rightarrow x_i$ are context-free rules, $1 \leq i \leq n$, $Q, R \subseteq N$, $Q \cap R = \emptyset$. A matrix can be applied to a string $x = x_1X_1x_2X_2 \ldots X_lx_{l+1}$ in order to rewrite effectively the symbols $X_1, \ldots, X_l$ only if $x_1, \ldots x_{l+1}$ contains all symbols of Q and no symbols of R. We denote by $RCM(M, \beta, \max(\alpha, \gamma))$ the family of languages generated by random context matrix grammars $G = (N, T, S, M, F)$ with rules of type β, $\beta \in \{CF, CF - \lambda\}$, with arbitrary F if $\gamma = ac$, or with empty F if γ is empty, with arbitrary R in $((r_1, \ldots, r_n), Q, R) \in M$ if $\alpha = ac$ and with empty α if no forbidding contexts are involved. $\max(\alpha, \gamma) = ac$ if atleast one of α, γ is ac. Thus, if no appearance checking is used, and if no forbidding contexts are used, we have the family $RCM(M, \beta, \emptyset)$. It is known [1] that $RCM(M, CF, \emptyset) = MAT^{\lambda}$, and $RCM(M, CF, ac) = MAT^{\lambda}_{ac} = RE$.

For basic elements of membrane computing we refer to [11]; for the state-of-the art of the domain, the reader may consult the bibliography from the web address `http://psystems.disco.unimib.it`. For proving computational universality, we use the concept of Minsky's register machine [9].

2.1 Register Machines

The universality proof in Section 4 is based on the concept of Minsky's register machine [9]. Such a machine runs a program consisting of numbered instructions of several simple types. Several variants of register machines with different number of registers and different instruction sets were shown to be computationally universal.

An *n-register machine* is a construct $M = (n, P, i, h)$, where: (i) n is the number of registers, (ii) P is a set of labeled instructions of the form $j : (op(r), k, l)$, where $op(r)$ is an operation on register r of M, j, k, l are labels from the set $Lab(M)$ (which numbers the instructions in a one-to-one manner), (iii) i is the initial label, and (iv) h is the final label.
The machine is capable of the following instructions:

$(add(r), k, l)$: Add one to the contents of register r and proceed to instruction k or to instruction l; in the deterministic variants usually considered in the literature we demand $k = l$.

$(sub(r), k, l)$: If register r is not empty, then subtract one from its contents and go to instruction k, otherwise proceed to instruction l.
halt: Stop the machine. This additional instruction can only be assigned to the final label h.

In their *deterministic variant*, such n-register machines can be used to compute any partial recursive function $f : \mathbf{N}^{\alpha} \to \mathbf{N}^{\beta}$; starting with $(n_1, \ldots, n_{\alpha}) \in \mathbf{N}^{\alpha}$ in registers 1 to α, M has computed $f(n_1, \ldots, n_{\alpha}) = (r_1, \ldots, r_{\beta})$ if it halts in the final label E with registers 1 to β containing r_1 to r_{β}. If the final label cannot be reached, then $f(n_1, \ldots, n_{\alpha})$ remains undefined.

A deterministic m-register machine can also analyze an input $(n_1, \ldots, n_{\alpha}) \in \mathbf{N}_0^{\alpha}$ in registers 1 to α, which is recognized if the register machine finally stops by the halt instruction with all its registers being empty. If the machine does not halt, the analysis was not successful.

In their *non-deterministic variant*, n-register machines can compute any recursively enumerable set of non-negative integers (or of vectors of non-negative integers). Starting with all registers being empty, we consider a computation of the n-register machine to be successful, if it halts with the result being contained in the first (β) register(s) and with all other registers being empty.

3 Pure Catalytic P Systems

In this section, we introduce briefly the basic system under consideration. A pure catalytic P system of degree m is a construct $\Pi = (V, C, \mu, w_1, \ldots, w_m, R_1, \ldots, R_m, i_0)$ where (i)V is an alphabet, (ii)$C \subseteq V$ is the set of catalysts, (iii) μ is a membrane structure consisting of m membranes, (iv)w_i, $1 \leq i \leq m$ are multisets of objects associated with region i of μ, (v) R_i are finite sets of rules over V of the form $ca \to cv$, $c \in C$, $a \in V/C$, v is a string from $((V/C) \times \{here, out, in\})^*$ associated with the regions i of μ, and (vi) i_0 is a number between 1 and m representing the output membrane of Π.

The rules are applied in a non-deterministic maximally parallel way, as usual in P systems. Thus, in each step, whenever a rule involving a catalyst is applicable, it should be applied. The number of objects in the output membrane at the end of a halting configuration is the output of Π. The class of all sets of numbers computable by pure catalytic P systems of degree $\leq m$, using $\leq k$ catalysts is denoted by $\mathbf{N}PC_m^{\lambda}(cat_k)$. When only λ-free rules are used, (rules are of the form $ca \to cv$, $v \in ((V/C) \times \{here, out, in\})^+$) the family is denoted by $\mathbf{N}PC_m(cat_k)$.

In the case of one-membrane systems, rules of the kind $ca \to c(v, out)$ is equivalent to $ca \to c$, since the v is lost outside the membrane. Thus, when we consider λ-free one membrane systems, we assume that we do not have rules (i) $ca \to cv$ where v consists of symbols (w, out), and (ii) rules $ca \to c$. However, in the case of systems with two or more membranes, when we say we have λ-free rules, we allow in all membranes other than the skin, (i) all kinds of rules $ca \to cw$, where $w \in (V/C \times \{here, out, in\})^+$, and, in the skin membrane, (ii) all rules other than $ca \to cw$ such that $w = \lambda$ or w contains a (v, out).

In the following, we show that 2 catalysts and one membrane cannot characterize RE when λ-free rules are used.

Theorem 1. $\mathbf{N}PC_1(cat_2) \subseteq \mathbf{N}RC_{p,f} \subset \mathbf{N}RE$.

Proof. Consider the pure catalytic P system $\Pi = (V, C, [\,_1]_1, w, R, 1)$, where $C = \{c_1, c_2\}$. As discussed above, all rules of R are of the form $ca \to c(w, here)$, $a \in V, w \in V^+$. Let $U_n, U_1, U_2, U_b \subseteq V$ be subsets of V as defined below:

$$U_n = \{a \in V \mid \nexists \text{ rules } c_1a \to c_1v \wedge \nexists \text{ rules } c_2a \to c_2w\},$$
$$U_1 = \{a \in V \mid \exists \text{ rules } c_1a \to c_1v \wedge \nexists \text{ rules } c_2a \to c_2w\},$$
$$U_2 = \{a \in V \mid \nexists \text{ rules } c_1a \to c_1w \wedge \exists \text{ rules } c_2a \to c_2v\},$$
$$U_b = \{a \in V \mid \exists \text{ rules } c_1a \to c_1v \wedge \exists \text{ rules } c_2a \to c_2w\}$$

Clearly, $V = U_n \cup U_1 \cup U_2 \cup U_b$. Let us construct a random context grammar $G = (N, T, S, P)$ where $N = V \cup V' \cup V'' \cup V''' \cup \widehat{V} \cup \{O, F, E\}$ with $V' = \{a' \mid a \in V\}$, $V'' = \{a'' \mid a \in V\}$, $V''' = \{a''' \mid a \in V\}$ and $\widehat{V} = \{\hat{a} \mid a \in V\}$. The set T of terminals is $T = \{H\} \cup \bar{V}$, where $\bar{V} = \{\bar{a} \mid a \in V\}$. We assume $F, O, E, H \notin V$. The set of productions P is as follows:

1. $(S \to Ow, \emptyset, \emptyset)$,
2. $(O \to F, \emptyset, V' \cup V'' \cup V''' \cup \widehat{V})$,
3. $(a \to v', \{F\}, \{O, E\} \cup V')$, if $c_1a \to c_1v \in R$,
4. $(a \to v'', \{F\}, \{O, E\} \cup V'')$, if $c_2a \to c_2v \in R$,
5. $(a \to a''', \{F\} \cup \{e'\}, \{O, E\})$, $e \in V, a \in U_1$,
6. $(a \to a''', \{F\} \cup \{e''\}, \{O, E\})$, $e \in V, a \in U_2$,
7. $(a \to v'', \{F\} \cup \{e'\}, \{O, E\} \cup V'')$, $e \in V, a \in U_b$, and $c_2a \to c_2v \in R$,
8. $(a \to v', \{F\} \cup \{e''\}, \{O, E\} \cup V')$, $e \in V, a \in U_b$, and $c_1a \to c_1v \in R$,
9. $(a \to a''', \{F\} \cup \{e', f''\}, \{O, E\})$, $e, f \in V, a \in U_b$,
10. $(a \to \hat{a}, \{F\}, \{O, E\})$, $a \in U_n$,
11. $(F \to O, \emptyset, V)$,
12. $(a' \to a, \{O\}, \{F, E\})$,
13. $(a'' \to a, \{O\}, \{F, E\})$,
14. $(a''' \to a, \{O\}, \{F, E\})$,
15. $(\hat{a} \to a, \{O\}, \{F, E\})$,
16. $(F \to E, \emptyset, V \cup V' \cup V'' \cup V''')$,
17. $(\hat{a} \to \bar{a}, \{E\}, V \cup V' \cup V'' \cup V''')$, $a \in V$,
18. $(E \to H, \emptyset, V \cup V' \cup V'' \cup V''' \cup \widehat{V})$.

We start with rule 1, rewriting S with Ow. The only applicable rule now is rule 2, which replaces O by F, provided the only other symbols in the string are over V. This being the case in Ow, we obtain Fw. In the presence of F, we simulate a computation step of Π. Thus, we are supposed to use a rule corresponding to c_1 if applicable, and a rule of c_2 if applicable. Rule 3 tells us how to simulate a c_1 rule. The symbol a is replaced by v' in the presence of F, provided there are

already no other symbols of V'. This ensures that we apply any c_1 rule atmost once. Similarly, a c_2 rule is simulated by replacing a with v''. The actual issue is that we have to know somehow when the simulation of a computation step is over. For this, we replace all other symbols in the string with either a''' or $\hat{a}$, so that we obtain a string with no symbols of V.

We will now look into the finer details of the simulation. Rule 5 says that a symbol $a \in U_1$ can be replaced by a''' (thereby not simulating the rule $c_1 a \to c_1 v$) only if there is already some e' in the string (which means that already some c_1 rule has been applied). This way, all symbols of U_1 can be replaced with a''' if there is already some e'. However, if there are no e''s in the string, then rule 5 cannot be applied, then we will have to use rule 3. Similar is the case of rule 6. Now let us look at rules 7,8,9. These deal with symbols $a \in U_b$. Now, we can replace a with a''' (thereby choosing not to simulate either of the rules $c_1 a \to c_1 v$ and $c_2 a \to c_2 v$) only if we already find two symbols e', f'' in the given string. If we find only an e', and do not find any f'', then it means that no c_2 rule has been simulated, and so, we cannot replace a with a'''. This is so, since the rest of the symbols might be belonging to $U_1 \cup U_n$. Thus, we have to replace a with v''. The case of rule 8 is similar wherein we check if any c_1 rule has already been simulated or not. Thus, symbols $a \in U_1, U_2, U_b$ are replaced with a''' only if there already exist some e' or an f'' or both e', f'' in the string respectively. Symbols of U_n are simply replaced with $\hat{a}$ since anyway they do not have any rules.

Thus, at the end of a simulation step, we obtain a string over $\{F\} \cup (V' \cup V'' \cup \widehat{V} \cup V''')^*$. Now, we need to replace all symbols over $V' \cup V'' \cup \widehat{V} \cup V'''$ with symbols over V to start the next simulation. This is done by first replacing F by O (rule 11). We can stop the simulation when we obtain a string over $F\widehat{V}^*$. This means that no rules are applicable to any of the symbols. In this case, we replace F with E and stop. Clearly, the symbols $\hat{a} \in \widehat{V}$ in the string $E\widehat{V}^*$ are those which will be in the membrane at the end of a halting computation. All that remains to be done now is to replace all $\hat{a} \in \widehat{V}$ by $\bar{a}$ in the presence of E. Once this is done, and all symbols are replaced, we replace E by H, obtaining a string $H\bar{V}^*$.

However, we do not require the symbol H. Since $RC_{p,f}$ is closed under left quotient by letters [2], we can remove H and still obtain the remaining string to be in $RC_{p,f}$. Hence, $N_1 PC_1(cat_2) \subseteq N_1(\partial_H^l(L)), L \in RC_{p,f}$ and H is a letter. Thus, $\mathbf{N}_1 PC_1(cat_2) \subseteq \mathbf{N}_1 RC_{p,f}$. □

Note: The above result can also be proved using classical results in complexity theory [10].

Corollary 1. $\mathbf{N}_1 PC_*(cat_2) \subseteq \mathbf{N}_1 PC_1(cat_2) \subseteq \mathbf{N}_1 RC_{p,f} \subset \mathbf{N}_1 RE$.

Proof. Given a system Π with 2 membranes $[_1[_2]_2]_1$, if we have both catalysts in the same region, then there cannot be any rules in the other region. Thus, these systems would be equivalent to one membrane systems with two catalysts.

Consider now the case when c_1 is in membrane 1 and c_2 is in membrane 2. In this case, we can construct a one membrane system Π' such that $\mathbf{N}(\Pi) \subseteq \mathbf{N}(\Pi')$. This is based on the following idea: Let $\Pi = (V, C, [_1[_2]_2]_1, x, y, R_1, R_2, i_0)$. Let

$U_1, U_2 \subseteq V$ be the subset of symbols which have no applicable rules in R_1, R_2 respectively.

Construct $\Pi' = (V_1 \cup V_2, C, [\,]_1, x_1y_2, R, 1)$ where $x_1 = h_1(x), y_2 = h_2(y)$, $h_1 : V \to V_1, h_2 : V \to V_2$ such that $h_i(a) = a_i, i \in \{1, 2\}$. Thus the objects in Π' are indexed by their original positions in Π. Rules of R_1 are applied to objects indexed with 1, and rules of R_2 are applied to objects indexed with 2. A rule $c_2a \to c_2(w, here)(y, out)$ of R_2 is used as $c_2a_2 \to c_2w_2y_1$ in R. Similarly, a rule $c_1a \to c_1(w, here)(y, out)(z, in)$ of R_1 is used as $c_1a_1 \to c_1w_1z_2$, since y is lost anyway. This way, we have in Π', all rules modified with respect to the indices, and Π' halts when Π halts. To obtain the same output as Π, we add rules $c_ia_i \to c_i$, $i \neq i_0$, and $a \in U_i$.

Finally, if we consider systems with 3 or more membranes, and 2 catalysts, then again, regions where the catalysts are not placed will be useless, thereby reducing these systems to one membrane systems Π'. □

Next, we consider some variants of pure catalytic P systems. Let $\Pi = (V, C, \mu, w_1, \ldots, w_m, R_1, \ldots, R_m, i_0)$ be a system with n catalysts $C = \{c_1, \ldots, c_n\}$. Let $C_i \subseteq V$ be defined as $C_i = \{a \in V \mid \exists c_ia \to c_iw \in R_j, \text{ some } j\}$. We consider two restricted systems here : (i) those in which the rules are only of the form $c_ia \to c_iw$, such that $alph(w) \cap C_j = \emptyset, j \neq i$, and (ii) those in which a computation can proceed until every catalyst c_i has an applicable rule in every step. The system halts when there is some catalyst with no applicable rule.

Systems of type (i) are such that the symbols w produced by a catalytic rule $c_ia \to c_iw$ can be used if at all, only by the catalyst c_i. Further, for systems of type (i), the moment a catalyst c_i has no more applicable rules, we can conclude that c_i will be idle forever, since no other c_j can create symbols which can be processed by c_i. This is not the case in usual pure catalytic systems, since a catalyst c_i may be idle for a few steps, and may again start working since new symbols which can be processed by c_i have been created by other c_j's.

Systems of type (ii) on the other hand, are those in which no catalyst is allowed to remain idle. Thus, starting from the initial configuration, all catalysts start working, and if there is a configuration where some catalyst has no rule to be used, then the system halts. The computation proceeds in both systems the same way as for pure catalytic systems, and the number of objects in the output membrane at the end of a halting configuration is the result. Let us denote the class of all sets of numbers computable by systems of types (i), (ii) using atmost k membranes and n catalysts by $\mathbf{N}PIC_k^{\lambda}(cat_n)$ (or $\mathbf{N}PIC_k^{\lambda}(cat_n)$ in case of λ-free rules) and $\mathbf{N}PPC_k^{\lambda}(cat_n)$ (or $\mathbf{N}PPC_k^{\lambda}(cat_n)$ in case of λ-free rules) respectively. The I in $\mathbf{N}PIC_k^{\lambda}(cat_n)$ stands for the independence of catalysts in performing rules (c_i is not dependent on the symbols produced by some c_j) and the P in $\mathbf{N}PPC_k^{\lambda}(cat_n)$ stands for the simultaneous (parallel) action of all catalysts in all steps.

Theorem 2. *1.* $\mathbf{N}PIC_1^{\lambda}(cat_2) \subseteq \mathbf{N}MAT^{\lambda} \subset \mathbf{N}RE$.
2. $\mathbf{N}PPC_1^{\lambda}(cat_2) \subseteq \mathbf{N}MAT^{\lambda} \subset \mathbf{N}RE$.

Proof. Consider a pure catalytic P system of type (i): $\Pi = (V, C, [\,]_1, w', R, 1)$ with $C = \{c_1, c_2\}$. Let $U_i \subseteq V$ be the set of all symbols of V which do not

have any rules corresponding to $c_i, i \in \{1,2\}$. Let there be n rules of the kind $c_1 a \to c_1 v$ and m rules of the kind $c_2 b \to c_2 w$ for $a, b \in V$.

We construct the matrix grammar $G = (N, T, S, M)$ where $N = V/(U_1 \cap U_2) \cup J$, $J = \{O, O_1, O_2, H_1, H_2\}$, $T = U_1 \cap U_2$, and $J \nsubseteq V$. The matrices are given by:

1. $(S \to Xw'), X \in \{O, H_1, H_2\}$,
2. $(a \to w, O \to O_1)$, if $c_1 a \to c_1 w$,
3. $(b \to v, O_1 \to O)$, if $c_2 b \to c_2 v$,
4. $(a \to w, O \to H_1)$, if $c_1 a \to c_1 w$,
5. $(b \to v, O_1 \to H_2)$, if $c_2 b \to c_2 v$,
6. $(b \to v, H_1 \to H_1)$, if $c_2 b \to c_2 v$,
7. $(a \to w, H_2 \to H_2))$, if $c_1 a \to c_1 w$,
8. $(H_i \to \lambda), i \in \{1,2\}$.

To start with, we had w' in membrane 1. Rule 1 is used, obtaining Xw'. Let us first look at the case when $X = O$. In this case, we assume that there is atleast one symbol in w which has a c_1 rule, since matrices 2,4 require c_1 rules. If rule 2 is used, O is replaced with O_1, and we use rule 3 next, simulating a c_2 rule. We can continue using rules 2,3 in alternate steps as long as both c_1, c_2 rules are applicable. One usage of rule 2 followed by rule 3 completes the simulation of a step of Π. At some step, we guess that there are no more symbols to which we can apply a c_1 rule from the next step onward. This is done by rule 4. O is replaced by H_1 instead of O_1. If we have H_1, then we can apply c_2 rules as long as we like using rule 6. At any point we can use rule 8, which will erase H_1. Thus, if we had guessed correctly while applying rule 4, then we will be left with symbols of U_1 in addition to symbols which can be processed by c_2 rules. After a point, if we apply rule 8 (thereby guessing that c_2 rules are also not applicable), then we will be left with symbols of $U_1 \cap U_2$. The case of changing O_1 to H_2 is similar. Note that this works because c_1 does not create symbols which have c_2 rules and vice versa; thus, if we obtain H_1 correctly, any number of c_2 rules will not introduce symbols which can be processed by c_1. Similar is the case for H_2.

Thus, to summarize, if w' has symbols a, b to which c_1 and c_2 rules are applicable, then $X = O$. If $w' = a_1 a_2$, and if a_1 has both c_1, c_2 rules applicable, but if a_2 has only c_2 rules applicable, then either $X = O$, by which both a_1, a_2 are processed, or $X = H_1$, by which we subject both a_1, a_2 to c_2 rules 6. Similar is the case if a_1 had both rules and a_2 has only c_1 rules. In this case, we can have $X = O$, processing a_2 first followed by a_1, or use $X = H_2$, processing both a_1, a_2 by c_1 rules 7. Similarly, if $w' = a, a \in V$ and if a has both c_1, c_2 rules applicable to it, then we can either use $X = H_1$ or $X = H_2$. If all symbols a of w' are in U_2 (U_1), then $X = H_2$ $(X = H_1)$. Finally, when none of the symbols of w' have any applicable rules, $X = H_1$ or H_2, and use rule 8. Hence, by correct guesses, we will end up with a string over $U_1 \cap U_2$, and hence, $\mathbf{N}PIC_1^{\lambda}(cat_2) \subseteq \mathbf{N}MAT^{\lambda}$.

Now we look at result 2. Consider the pure catalytic P system of type (ii), $\Pi = (V, C, [\,]_1, u, R, 1)$ with $C = \{c_1, c_2\}$. Let $U \subseteq V$ be the set of all symbols of V which do not have any rules in R. Let $U_n, U_1, U_2, U_b \subseteq V$ be subsets of V as defined in Theorem 1. Π halts when either one (or both) catalysts have no applicable rules.

We construct the matrix grammar $G = (N, T, M, S)$ where $N = V \cup \{D\} \cup J \cup \langle V \rangle$, where $J = \{O, O_1, E, E_1, E_2, E_3, E_4, E_5\}$, $\langle V \rangle = \{\langle w \rangle \mid \exists c_i a \to c_i w \in R, w \neq \lambda\}$. In case $w = \lambda$, we represent $\langle \lambda \rangle$ by $\langle D \rangle$. Since the number of rules as well as the lengths of all w in rules $c_i a \to c_i w$ is finite, $\langle V \rangle$ is finite. Further, $J \cup \{D\} \not\subseteq V$, $T = \widehat{V}$, $\widehat{V} = \{\hat{a} \mid a \in V\}$. The matrices are given by

1. $(S \to Xu), X \in \{O, E_1\}$,
2. $(a \to \langle \alpha \rangle, b \to \langle \beta \rangle, O \to X),$ if $c_1 a \to c_1 w, c_2 b \to c_2 v,\ X \in \{O_1, E\}$,
   $(\alpha = w,$ if $w \neq \lambda,\ \alpha = D,$ if $w = \lambda,$ and $\beta = v,$ if $v \neq \lambda,\ \beta = D,$ if $v = \lambda)$
3. $(\langle w \rangle \to w, \langle v \rangle \to v, O_1 \to O),\ v, w \in V^+ \cup \{D\}$,
4. $(\langle w \rangle \to w, \langle v \rangle \to v, E \to E_1),\ v, w \in V^+ \cup \{D\}$,
5. $(a \to \hat{a}, E_1 \to E_5), a \in U_n$,
6. $(a \to \hat{a}, E_5 \to E_5), a \in U_n$,
7. $(a \to \hat{a}, E_1 \to E_2), a \in U_b$,
8. $(a \to \hat{a}, E_2 \to E_2), a \in U_n$,
9. $(a \to \hat{a}, E_1 \to E_3), a \in U_1$,
10. $(a \to \hat{a}, E_3 \to E_3), a \in U_1 \cup U_n$,
11. $(a \to \hat{a}, E_1 \to E_4), a \in U_2$,
12. $(a \to \hat{a}, E_4 \to E_4), a \in U_2 \cup U_n$,
13. $(D \to \lambda, E_i \to E_i), i \in \{2, 3, 4, 5\}$,
14. $(E_i \to \lambda), i \in \{2, 3, 4, 5\}$.

We start here with rule 1, obtaining Ou or $E_1 u$. Let us first examine the case of Ou. This means we assume that atleast one step is executed by Π, with both catalysts acting in parallel. Rule 2 does this, simulating rules corresponding to c_1, c_2 replacing O. If O is replaced with O_1, then we use rule 3, and continue with rule 2. Thus, as long as rules 2,3 are used alternately, it means that in every step, a c_1 rule and a c_2 rule is applied. At some point, we guess that from the next step onward, atleast one of c_1, c_2 will not have applicable rules and hence we need to stop. This is done by replacing O with E at the end of some step, which symbolizes the fact that this is the last step wherein both c_1, c_2 have applicable rules. With E, we can use rule 4, obtaining E_1. With E_1, we will check whether we have guessed correctly or not.

Let $E_1 x$ be the current sentential form. If x consists of only symbols of U_n, then we use rules 5,6 obtaining symbol the E_5. All symbols $a \in U_n$ are converted to $\hat{a}$. Then, if rule 14 is applied after replacing all a's by $\hat{a}$, we obtain a terminal string. If x has exactly one symbol of U_b and all other symbols in U_n, then again the guess is correct, since this symbol can be processed either by c_1 or c_2, but

not by both. In this case, rules 7,8 are used, obtaining E_2. If x had symbols of U_1, U_n alone, then again the guess is correct since there are no symbols with c_2 applicable rules. In this case, rules 9, 10 are used, obtaining E_3. Finally, if x was over $U_2 \cup U_n$, we use rules 11, 12. Thus, E_2 symbolizes the case with exactly one symbol of U_b, and the rest over U_n, E_5 symbolizes the case when all symbols are over U_n, E_3 symbolizes the case when symbols are over $U_1 \cup U_n$ and finally E_4, when symbols are over $U_2 \cup U_n$.

Thus, if the guess was not correct, then we will not get the entire string over $\widehat{V}^*$, which means we will not get a terminal string. Thus, we obtain a terminal string iff a configuration where atleast one of c_1, c_2 are forced to be idle is obtained. □

4 Pure Catalytic Systems with Mobile Catalysts

In this section, we consider some variants of pure catalytic P systems with mobile catalysts. A pure catalytic P system with mobile catalysts is a construct $\Pi = (V, T, C, \mu, w_1, \ldots, w_m, R_1, \ldots, R_m, i_0)$. The only difference of this system with respect to a pure catalytic system as defined in Section 3 is that the rules are of the form $ca \rightarrow (c, tar)\beta$ where $tar \in \{here, out, in\}$ and $\beta \in (V \times \{here, out, in\})^*$. Thus, the catalysts themselves can move. We shall denote the set of numbers computed by pure catalytic P systems of degree $\leq m$, having atmost k mobile catalysts by $NPMC_m^{\lambda}(mcat_k)$ (or $NPMC_m(mcat_k)$).

We show now that with 2 membranes and 2 mobile catalysts, using only catalytic rules, all recursively enumerable sets of numbers can be obtained.

Theorem 3. $\mathbf{N}PMC_2^{\lambda}(mcat_2) = \mathbf{N}RE$.

Proof. We only prove the assertion $\mathbf{N}RE \subseteq \mathbf{N}PMC_2^{\lambda}(mcat_2)$, and infer the other inclusion from the Church-Turing thesis. The proof is based on the observation that each set from $\mathbf{N}RE$ is the range of a recursive function. Thus, we will prove the following assertion. For each recursively enumerable function $f : \mathbf{N} \rightarrow \mathbf{N}$, there is a mobile P System Π with 2 membranes having 2 mobile catalysts satisfying the following condition: For any arbitrary $x \in \mathbf{N}$, the system Π first "generates" a multiset of the form o_1^x and halts if and only if $f(x)$ is defined, and, if so, the result of the computation is $f(x)$.

In order to prove this assertion, we consider a register machine with 3 registers, the last one being a special output register which is never decremented. Let there be a program P consisting of h instructions $P_1, \ldots, P_h$ which computes f. Let P_h correspond to the instruction HALT and P_1 be the first instruction. The input value x is expected to be in register 1 and the output value in register 3. Without loss of generality, we can assume that all registers other than the first one are empty at the beginning of a computation. We can also assume that in the halting configuration all registers except the third, where the result of the computation is stored, are empty.

Construct the pure catalytic P system with mobile catalysts $\Pi = (V, C, [_1[_2]_2]_1, \emptyset, \{c_1c_2o\}, R_1, R_2, 2)$ where

$$
\begin{aligned}
V &= \{c_1, c_2, o_1, o_2, o_3\} \cup \{P_j, P'_j, Q'_j, P''_j, Q''_j, P'''_j, Q'''_j \mid 1 \leq j \leq h\},\\
C &= \{c_1, c_2\},\\
R_1 &= \{c_2o_2 \to (c_2, in), c_2Q'''_i \to (c_2, in)(P_j, in)\}\\
&\qquad \text{(simulation of a SUB instruction } i = (SUB(2), k, j))\\
&\quad \cup \{c_1o_1 \to (c_1, in), c_1P'''_i \to (c_1, in)(P_j, in)\},\\
&\qquad \text{(simulation of a SUB instruction } i = (SUB(1), k, j)),\\
R_2 &= \{c_1o \to c_1o(o_1, out),\ c_1o \to c_1P_1\}\\
&\qquad \text{(generation of initial contents of register 1)}\\
&\quad \cup \{c_1P_i \to c_1(o_r, out)P_j, r = 1, 2\} \cup \{c_1P_i \to c_1o_3P_j, r = 3\}\\
&\qquad \text{(simulation of an ADD instruction } i = (ADD(r), j), 1 \leq r \leq 3)\\
&\quad \cup \{c_1P_i \to (c_1, out)P'_i\} \cup \{c_2P'_i \to c_2P''_id\}\\
&\quad \cup \{c_1P''_i \to c_1P_k, c_2d \to c_2\} \cup \{c_2P''_i \to c_2(P'''_i, out), c_1d \to c_1\dagger\}\\
&\qquad \text{(simulation of a SUB instruction } i = (SUB(1), k, j))\\
&\quad \cup \{c_2P_i \to (c_2, out)Q'_i, c_1Q'_i \to c_1Q''_ie\} \cup \{c_2Q''_i \to c_2P_k, c_1e \to c_1\}\\
&\quad \cup \{c_1Q''_i \to c_1(Q'''_i, out), c_2e \to c_2\dagger\}\\
&\qquad \text{(simulation of a SUB instruction } i = (SUB(2), k, j))\\
&\quad \cup \{c_1P_h \to (c_1, out)P'_h, c_2P'_h \to (c_2, out)\}.
\end{aligned}
$$

In the initial configuration, membrane 2 contains the two mobile catalysts as well the object o, and membrane 1 is empty. We shall now examine the working of the system.

1. Generation of o_1^x, the initial contents of register 1: We apply the rules $c_1o \to c_1o(o_1, out)$ in R_2 arbitrarily many times, there by introducing as many copies of o_1 in membrane 1. This is stopped at some point of time by the rule $c_1o \to c_1P_1$ in R_2, introducing the object P_1 representing the initial instruction in membrane 2.
2. Simulation of an ADD instruction $P_i = (ADD(r), j), 1 \leq r \leq 3, 1 \leq i, j \leq h$ This is done in membrane 2. The catalyst c_1 interacts with P_i, converting it into P_j, and also producing either an o_1, o_2 or an o_3 (depending on whether P_i is an instruction for incrementing register 1,2 or 3). o_1, o_2 are sent to membrane 1, while o_3 is retained in membrane 2.
3. Simulation of a SUB instruction $P_i = (SUB(r), k, j), 1 \leq i, j, k \leq h, r = 1, 2$ Let us look at the case when $r = 1$. c_1 interacts with P_i, converting it into P'_i, while c_1 moves out to membrane 1. In membrane 1, c_1 decrements an o_1 if present, and comes back to membrane 2, while in parallel, c_2 interacts with P'_i in membrane 2, converting it into P''_id. Note that c_1 can go inside membrane 1 only in the presence of a decrementing instruction P_i (decrementing register 1). In the next step, if c_1 is back in membrane 2, it means

that there was indeed an o_1 in membrane 1. Then, the rule $c_1 P_i'' \to c_1 P_k$ and $c_2 d \to c_2$ are applied in parallel, completing the simulation.
However, if there was no o_1 in membrane 1, then c_1 would be waiting in membrane 1. In this case, c_2 interacts with P_i'', and send out a P_i'''. c_1 then interacts with P_i''', converting it to P_j, and enters membrane 2. Note that if c_2 acts on P_i'' while c_1 is present in membrane 2, an infinite computation will be induced by the rule $c_1 d \to c_1 \dagger$.
The simulation of SUB instruction for register 2 is similar, except that c_2 plays the role of c_1 and the objects Q_i', Q_i'', Q_i''' play the role of P_i', P_i'', P_i'''.

4. Halting : When the instruction P_h is introduced in membrane 2, there are no more instructions to be simulated. So, c_1 interacts with it, converting it into P_h', and leaves membrane 2. c_2 then acts with P_h', erases it and also leaves membrane 2. Since we assume that all registers other than register 3 are empty at the end of a halting configuration, catalysts c_1, c_2 remain in membrane 1. Thus, at the end, membrane 2 contains the contents of register 3 at the end of the computation.

□

We next show that if only 2 membranes are used, one mobile catalyst cannot give universality.

Theorem 4. $\mathbf{N}PMC_2^{\lambda}(mcat_1) \subseteq \mathbf{N}RCM(M, CF, \emptyset) = \mathbf{N}MAT^{\lambda} \subset \mathbf{N}RE$.

Proof. Consider a pure catalytic system with one mobile catalyst c and 2 membranes $\Pi = (V, \{c\}, [_1[_2]_2]_1, v, w, R_1, R_2, i_0)$, where i_0 is either 1 or 2. Without loss of generality, let us assume that c is in membrane 1 in the initial configuration. We also assume that we remove components (x, out) from rules in R_1 which contain a (x, out) on the RHS, since anyway x is lost.

Construct the random context matrix grammar $G = (N, T, M, S)$ without appearance checking with $N = V_1 \cup V_2 \cup \{1, 2\}$, where $V_1 = \{a_1 \mid a \in V\}$, $V_2 = \{a_2 \mid a \in V\}$. Let $U_1 = \{a_1 \in V_1 \mid \nexists ca \to cv \in R_1\}$ and let $U_2 = \{a_2 \in V_2 \mid \nexists ca \to cv \in R_2\}$. Then, let $T = V_{i_0}'$, where $V_{i_0}' = \{a_{i_0}' \mid a \in V\}$. The matrices are the following:

1. $((S \to 1 v_1 w_2), \emptyset, \emptyset),\ v_1 \in V_1^*, w_2 \in V_2^*,$
2. $((a_i \to x_i y_{tar}, i \to tar), \emptyset, \emptyset),$ if $ca \to (c, tar)(x, here)(y, tar) \in R_i,$ and $i, tar\ \in \{1, 2\},$
2′. $((a_i \to \lambda, i \to tar), \emptyset, \emptyset),$ if $ca \to (c, tar) \in R_i,$ and $i, tar\ \in \{1, 2\},$
3. $((i \to S_i), \emptyset, \emptyset), i \in \{1, 2\},$
4. $((a_{i_0} \to a_{i_0}'), \{S_{i_0}\}, \emptyset),\ a_{i_0} \in U_{i_0},$
5. $((a_j \to \lambda), \{S_{i_0}\}, \emptyset),$ if $j \neq i_0,$
6. $((a_i \to \lambda), \{S_i\}, \emptyset),$ if $i_0 \neq i$ and $a_i \in U_i,$
7. $((a_{i_0} \to a_{i_0}'), \{S_i\}, \emptyset),\ a_{i_0} \in V_{i_0},\ i \neq i_0,$
8. $((S_i \to \lambda), \emptyset, \emptyset), i \in \{1, 2\}.$

If we have v and w in membranes 1 and 2 in the initial configuration, then we replace S with $1 v_1 w_2$, where the 1 signifies the initial position of the mobile

catalyst. At every step, we need to remember the position of the mobile catalyst since it is this position which determines which rule is applicable.

In the presence of an i, $i \in \{1, 2\}$, we replace a symbol a_i, if there is a rule involving a in R_i. This is done by rule 2. Note that rule 2 is applicable only when we have i. a_i is replaced by a string $x_i y_{tar}$ if the rule was $ca \rightarrow (c, here)(x, here)(y, tar)$. In this case, the second rule in the matrix would read $i \rightarrow i$, since the catalyst is still in the same region. However, if the rule involved (c, tar), then we would have had the second rule of the matrix as $i \rightarrow tar$. Thus, after applying each rule, we know the current position of the catalyst.

At some point of time, we need to stop this process. We make a guess at some point of time, by replacing i with S_i. Now suppose we were in i_0 and replaced it with S_{i_0}. Then, the guess is correct if no symbols in i_0 have any applicable rules. However, if there are symbols over $V_{i_0} - U_{i_0}$, then our guess is wrong since there are symbols with applicable rules. We replace symbols $a_{i_0} \in U_{i_0}$ by a'_{i_0}. We do not require symbols of membranes other than i_0 since they do not form part of the output. Thus, we erase all $a_j, j \neq i_0$, in the presence of S_{i_0}. Thus, if our guess was right, then at the end, we would obtain a string over $S_{i_0}(V'_{i_0})^*$. We can erase S_{i_0} and thus obtain a terminal string. Note however that even if our guess was right, if we erase S_{i_0} prematurely, then we will not get a terminal string. Similarly, if our guess was wrong, then we will definitely not get a terminal string since symbols over $V_{i_0} - U_{i_0}$ will remain. Rules 4, 5 are used in this case.

Now suppose we stop with $S_i, i \neq i_0$. Then, if the guess is right, we will have no symbols in membrane i with applicable rules. Since these symbols do not contribute to the output, we can erase them. Thus, if all symbols were over U_i, then all of them can be erased. The remaining symbols of the string are in i_0 and belong to the output. We replace all symbols a_{i_0} with a'_{i_0} in the presence of $S_i, i \neq i_0$. Then we obtain a terminal string after erasing S_i. Note here also that if the guess was wrong, then we will have symbols of $V_i - U_i$ which cannot be erased, and so will never get a terminal string. Similarly, if we erase S_i prematurely also, we may not get a terminal string. Rules 6, 7 are used here.

Thus, irrespective of whether we have $S_i, i \neq i_0$ or S_{i_0}, we will always get a terminal string, which represents the number of symbols in membrane i_0 at the end of a halting configuration if (i) we make the correct guess and (ii) we erase S_i after all other symbols in the string are processed. Clearly, for any $w \in L(G)$, $|w|$ represents the number of symbols in the output membrane at the end of a halting computation, and hence, $\mathbf{N}PMC_2(mcat_1) \subseteq \mathbf{N}RCM(M, CF, \emptyset)$. □

5 Conclusion

In this paper, we have attempted to find the exact computing power of P systems using catalysts. We have partially answered the interesting question regarding universality of P systems with 2 catalysts. We have also shown that universality can be obtained using 2 or more mobile catalysts, but cannot be obtained if less than 2 mobile catalysts are used. The interesting question about the power of 2 catalysts while using λ-rules is still open.

References

1. J. Dassow, Gh. Păun: *Regulated Rewriting in Formal Language Theory*. Springer-Verlag, Berlin, 1989.
2. J. Dassow: *Grammars with Regulated Rewriting*, Handout given during the Ph.D school at Terragona, 2003 (`theo.cs.uni-magdeburg.de/dassow/tarraphd.pdf`).
3. R. Freund, L. Kari, M. Oswald, P. Sosik: Computationally universal P systems without priorities: two catalysts are sufficient, *Theoretical Computer Science*, 330 (2005), 251–266.
4. R. Freund, M. Oswald, P. Sosik: Reducing the number of catalysts required in computationally universal systems without priorities, *proceedings of DCFS 2003*, 102–113.
5. Oscar H. Ibarra, H.-C. Yen: On Deterministic Catalytic Systems, *Proceedings of CIAA 2005*, 163-175.
6. Oscar H. Ibarra, Zhe Dang, O. Egecioglu, G. Saxena: Characterizations of Catalytic Membrane Computing Systems,*Proceedings of MFCS 2003*, 480-489.
7. S.N. Krishna, A. Păun: Results on catalytic and evolution-communication P systems. *New Generation Computing*, 22, 4 (2004), 377–394.
8. S. N. Krishna: On Pure Catalytic P Systems, Technical Report, IIT Bombay, 2006 (`www.cse.iitb.ac.in/~krishnas/uc06.ps`).
9. M.L. Minsky: *Computation: Finite and Infinite Machines*. Prentice Hall, Englewood Cliffs, New Jersey, 1967.
10. C. Papadimitriou: *Computational Complexity, 1st edition*, Addison Wesley, 1993.
11. Gh. Păun: *Membrane Computing – An Introduction*. Springer-Verlag, Berlin, 2002.
12. Gh. Păun: 2006 Research Topics in Membrane Computing, manuscript, 2006.
13. G. Rozenberg, A. Salomaa, eds.: *Handbook of Formal Languages*, Springer, 1997.
14. P. Sosik: The power of catalysts and priorities in membrane computing, *Grammars*, 6(1) (2003), 13–24.
15. P. Sosik, R. Freund: P Systems without priorities are computationally universal, *Proceedings of WMC 2002*, 400–409.

Mapping Non-conventional Extensions of Genetic Programming

W.B. Langdon

Department of Computer Science
University of Essex, UK

Abstract. Conventional genetic programming research excludes memory and iteration. We have begun an extensive analysis of the space through which GP or other unconventional AI approaches search and extend it to consider explicit program stop instructions (T8) and any time models (T7). We report halting probability, run time and functionality (including entropy of binary functions) of both halting and anytime programs. Turing complete program fitness landscapes, even with halt, scale poorly.

1 Introduction

Recent work on strengthening the theoretical underpinnings of genetic programming (GP) has considered how GP searches its fitness landscape [1]. Results gained on the space of all possible programs are applicable to both GP and other search based automatic programming techniques. We have proved convergence results for the two most important forms of GP, i.e. trees (without side effects) and linear GP. Few researchers allow their GP's to include iteration or recursion. Indeed there are only about 60 papers (out of 4000) where loops or recursion have been included in GP [2, Appendix B]. Without some form of looping and memory there are algorithms which cannot be represented and so GP stands no chance of evolving them.

We have recently shown [3] in the limit of large T7 programs (cf. Figure 1) that:

– The T7 halting probability falls sub-linearly with program length. Our models suggest the chance of not looping falls as $O(\text{length}^{-1/2})$. Whilst including both non-looping and programs which escape loops we observe $O(\text{length}^{-1/4})$.

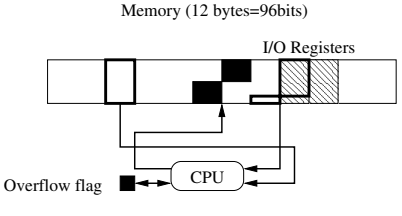


Fig. 1. T7 and T8 have the same bit addressable memory and input–output

C.S. Calude et al. (Eds.): UC 2006, LNCS 4135, pp. 166–180, 2006.

- Run time of terminating programs grows sub-linearly with program length. Again both mathematical and Markov models are confirmed by experiments and show run time of non-looping programs grows as $O(\text{length}^{1/2})$. Similarly, including both non-looping and programs which exit loops, for the T7, we observe run time $\leq O(\text{length}^{3/4})$.
- Despite the fraction of programs falling to zero, the shear number of programs, means the number of halting T7 programs grows exponentially with their size.
- Experimentally the types of loop and their length varies with the size of T7 program. Long programs are dominated by programs which fall into, and cannot escape from, one of two types of loop. In both cases the loops are very tight. So (in our experiments) even for the longest programs (we considered programs of up to 16 million instruction) on average no more than a few hundred different instructions are executed.

It is important to stress that these, and our previous results, apply not only to genetic programming, but to any other unconventional computation embedded in the same representation.

While the T7 computer is Turing complete, [2, Appendix A], these results are not universal. The T7 was chosen since it is a minimal Turing complete von Neumann architecture computer with strong similarities with both real computers and linear genetic programming [4]. At the 2006 Dagstuhl "Theory of Evolutionary Algorithms" [Seminar 06061] the question of the generality of the T7 was raised. In Section 4 we shall show that the impact of the addition of an explicit halt instruction is, as predicted, to dramatically change the scaling laws. With the T8 computer (T7+halt) almost all programs stop before executing more than a few instructions.

Sections 5 and 6 consider a third alterative halting technique: the any time algorithms [5]. In this regime, each program is given a fixed quantum of time and then aborted. The program's answer is read from the output register regardless of where its execution had reached. These last two experimental sections (5 and 6) consider program functionality, rather than just if they stop or not.

2 T7 and T8 – Example Turing Complete Computers

To test our theoretical results we need a simple Turing complete system. In [3] we introduced the T7 seven instruction CPU, itself based on the Kowalczy F-4 minimal instruction set computer `http://www.dakeng.com/misc.html`, cf. appendix of [2]. The T8 adds a single halt instruction to the T7 instruction set.

The T8 (see Figure 1 and Table 1) consists of: directly accessed bit addressable memory (there are no special registers), a single arithmetic operator (ADD), an unconditional JUMP, a conditional Branch if oVerflow flag is Set (BVS) jump, four copy instructions and the program halt. COPY_PC allows a programmer to save the current program address for use as the return address in subroutine calls, whilst the direct and indirect addressing modes allow access to stacks and arrays.

Table 1. T8 Turing Complete Instruction Set

Instruction	*args*	*operation*
ADD	3	A + B→C v set
BVS	1	#addr→pc if v=1
COPY	2	A→B
LDi	2	@A→B
STi	2	A→@B
COPY_PC	1	pc→A
JUMP	1	addr→pc
HALT	0	pc→end

Every ADD either sets or clears the overflow bit v. COPY_PC and JUMP use just enough bits to address each program instruction. LDi and STi, treat one of their arguments as the address of the data. (LDi and STi data addresses are 4 or 8 bits.) JUMP addresses are moded with program length. Programs terminate either by executing their last instruction (which must not be a jump) or by executing a HALT.

In Section 4 eight bit byte data words are used, whilst Sections 5 and 6 both use four bit nibbles. The number of bits in address words is just big enough to be able to address every instruction in the program. E.g., if the program is 300 instructions, then BVS, JUMP and COPY_PC instructions use 9 bits. These experiments use 12 bytes (96 bits) of memory (plus the overflow flag).

3 Experimental Method

There are too many programs to test all of them, instead we gather representative statistics about those of a particular length by randomly sampling. By sampling a range of lengths we create a picture of the whole search space. Note we do not bias the sampling in favour of short programs.

One hundred thousand programs of each of various lengths (1...16 777 215 instructions) are each run from a random starting point (NB not necessarily from the start) with random inputs, until either they execute a HALT, reach their last instruction and stop, an infinite loop is detected or an individual instruction has been executed more than 100 times. (In practise we can detect almost all infinite loops by keeping track of the machine's contents, i.e. memory and overflow bit. We can be sure the loop is infinite, if the contents is identical to what it was when the instruction was last executed.) The program's execution paths are then analysed. Statistics are gathered on the number of instructions executed, normal program terminations, type of loops, length of loops, start of first loop, etc.

4 Terminating T8 Programs

Figure 2 shows, as expected, inclusion of the HALT instruction dramatically changes the nature of the search space. Almost all T8 programs stop, with only a small fraction looping. This is the opposite of the T7 (most programs loop).

Figures 3 and 4 show the run time of terminating T8 programs. In both programs stopped by reaching their end (Figure 3) and by a HALT instruction (Figure 4), the fraction of programs falls exponentially fast with run time. In both cases, it falls most rapidly with short programs and appears to reach a limit of $(7/8)^{-\text{length}}$ for longer programs. A decay rate of 7/8 would be expected if

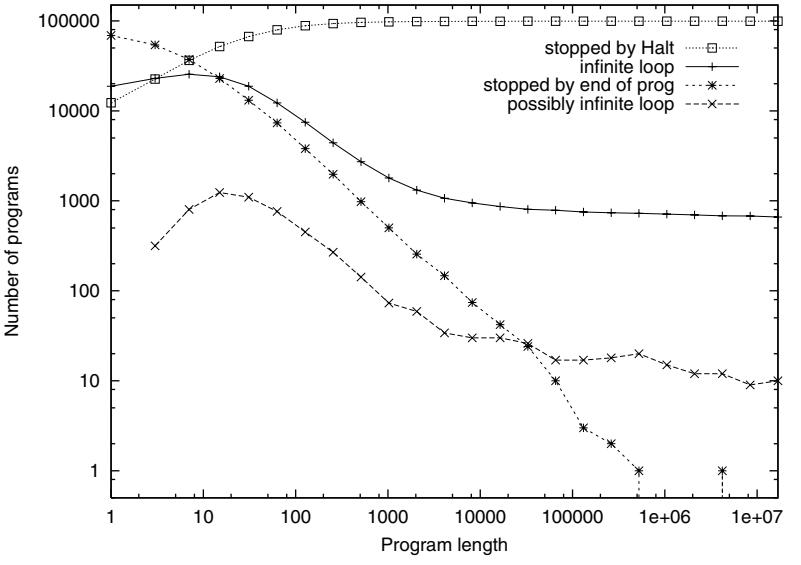


Fig. 2. Almost all short T8 programs are stopped by reaching their end (*). This proportion falls rapidly (about $\propto 1/\text{length}$) towards zero. Longer programs are mostly stopped by a halt instruction □. The fraction of programs trapped in loops (+ and ×) appears to settle near a limit of 1 in 150.

programs ran until they reach a HALT instruction. I.e. to a first approximation, run time of long terminating T8 programs can be estimated by ignoring the possibility of loops. This gives a geometric distribution and so an expected run time of 8 instructions regardless of program size. For all but very short programs, Figure 5 confirms the mean is indeed about 8. For a geometric distribution the standard deviation is 7.48 (also consistent with measurements) so almost all T8 programs terminate after executing no more than 31 instructions (mean+3σ). Again this is in sharp contrast with the T7, where long terminating T7 programs run many instruction, and so perhaps may do something more useful.

5 T8 Functions and Any Time Programs

The introduction of Turing completeness into genetic programming raises the halting problem, in particular how to assign fitness to a program which may loop indefinitely [6]. Here we look at any time algorithms [5] implemented by T8 computers. I.e. we insist all program halt after a certain number of instructions. Then we extract an answer from the output register regardless of whether it terminated or was aborted. (The input and output registers are mapped to overlapping memory locations, which the CPU treats identically to the rest of the memory, cf. Figure 1.) We allow the T8 53 instructions. (53 was chosen since by then we expect all but 0.1% of non-looping T8 programs to have stopped, cf. Figure 4.)

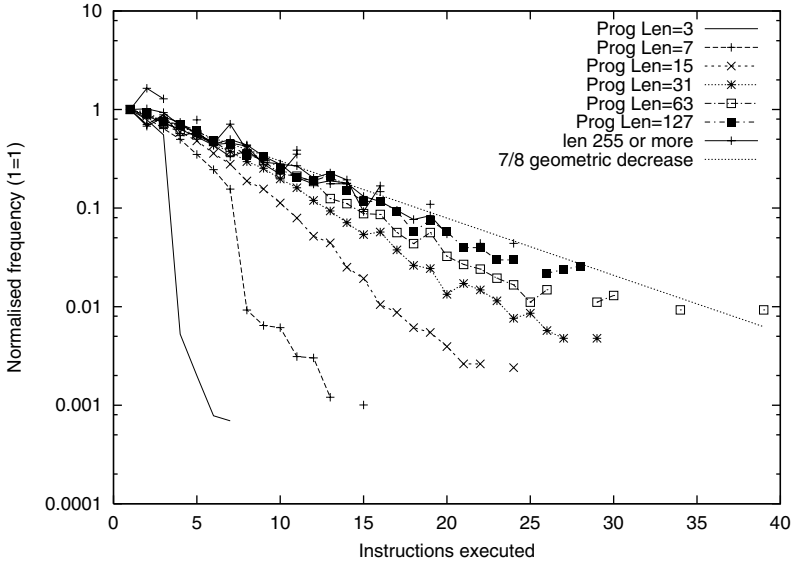


Fig. 3. Distribution of frequency of normally stopping T8 programs by their run time. Noisy data suppressed. To ease comparison, all data has been vertically rescaled so that data at the left lie on top of each other. Distributions fit geometric decay. As program gets longer the decay constant tends to 7/8.

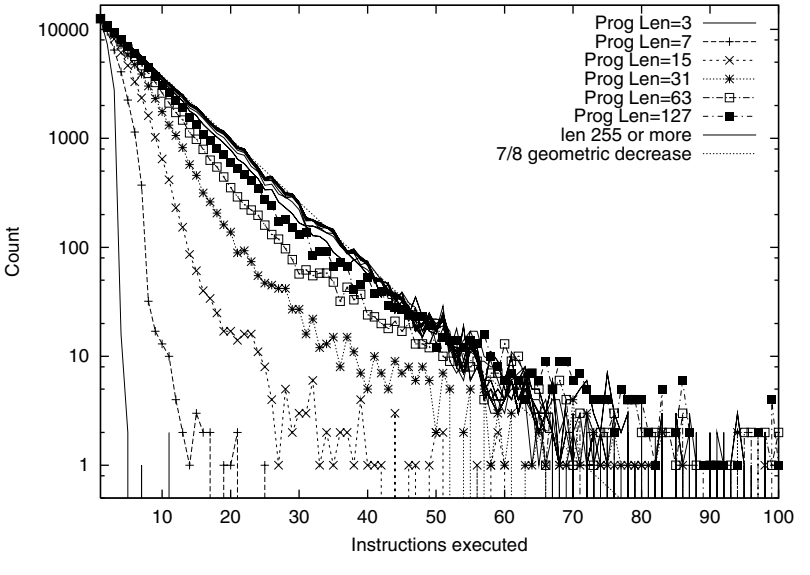


Fig. 4. Distribution of frequency of HALTed T8 programs by their run time. There is an geometric fall in frequency at all lengths, however for lengths < 127 the decay is faster than (1-1/8). For longer random T8 programs, the decay constant tends to 7/8.

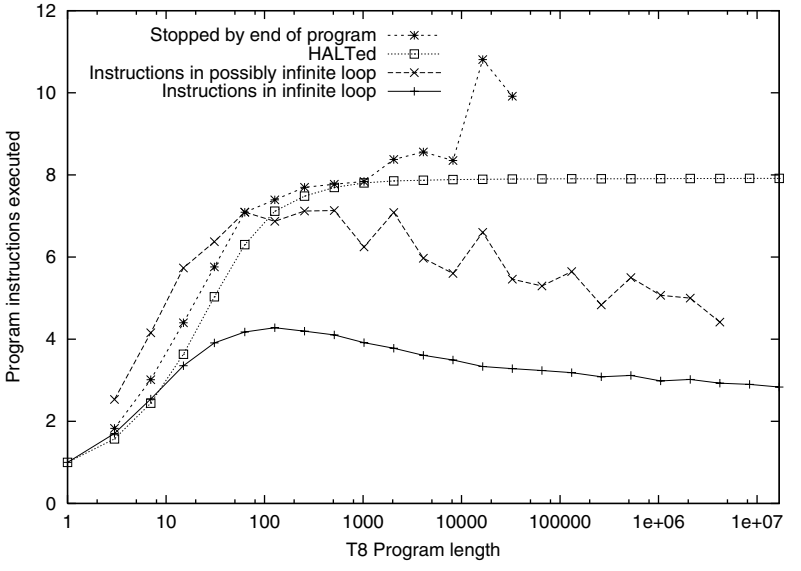


Fig. 5. Mean number of T8 instructions obeyed by terminating programs ($*$ and $\square$) and length of loops ($\times$ and $+$) v. program size. Noisy data suppressed. Data are consistent with long random T8 programs having a geometric distribution, with mean of 8 and thus standard deviation is $\sqrt{8^2 7/8} = 7.48$. Final loops in non-terminating programs are even tighter than in T7 [3, Fig. 9].

In this section and Section 6 we look at functions of two inputs, by defining two input registers (occupying adjacent 4 bit nibbles) and looking at the data left in memory after the program stops (or is stopped). In these sections, the data word size is 4 bits. Each random program is started from a chosen random starting point 256 times just as before, except the two input registers are given in turn each of their possible values. To avoid excessive run time and since we are now running each program 256 times (rather than once) the number of programs tested per length is reduced from 100 000 to 1000.

In addition to studying the random functions generated by the T8 we also study the variation between individual programs runs with each of the 256 different inputs and how this varies as the program runs. We use Shannon's [7] information theoretic entropy measure $S = -\sum_k p_k log_2(p_k)$, to quantify the difference between the state of the T8 (programme counter, overflow bit and memory) on different runs, with different inputs, at the same time.

5.1 Any Time T8 Entropy

We have two confounding effects. We measure the change in the variation between T8 processors as they run the same program (from different inputs), so a major influence is whether a particular processor is still running or has obeyed a HALT instruction or reached the end of its program. For simplicity when

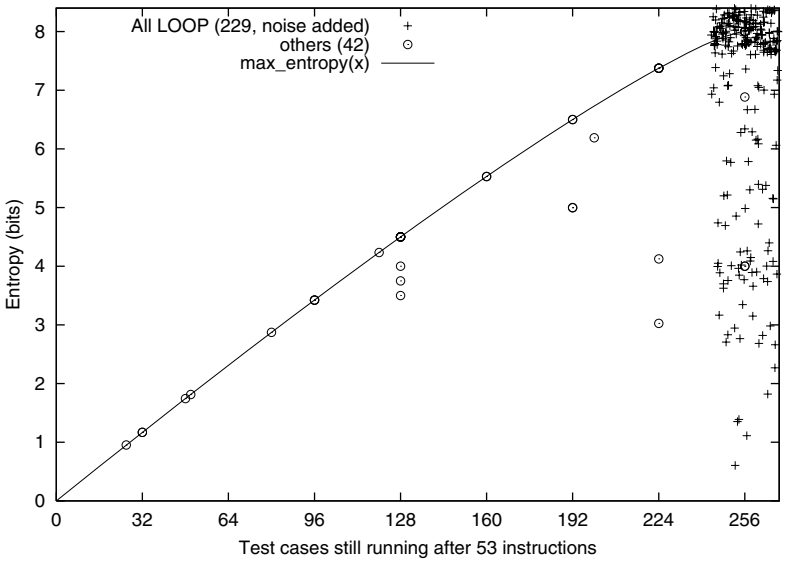


Fig. 6. Entropy of T8 programs of 7 instructions still running at least one test case after 53 time steps. Noise is added to + data to spread them.

calculating the entropy of a mixture of stopped and still executing programs, the stopped programs are treated as if they were all in the same stopped state. This means in the graphs that follow, the number of runs of a program that reach the t^{th} instruction has an impact of the entropy as well as (for example) the difference between the contents of memory. (Remember apart from the input register, each of the 256 runs, start in the same, random, state.) To illustrate this, Figure 6 shows the 229 (+) of 1000 T8 programs of 7 instructions which always get stuck in a loop tend to have a high entropy. In contrast the entropy of the remaining 42 programs *circ* strongly depends upon how many runs (test cases) are still executing. (Figure 6 takes a snap shot after 53 instructions have been obeyed). It is clear the number of active test cases has a strong influence on the variation in memory contents etc. between test cases. I.e. entropy, in most short T8 programs, is as large as possible, given the number of test cases still running. This is consistent with the fact that most short programs implement the identity function, see Figure 9, which has maximum entropy.

Figure 7 shows the average number of test cases still running up to instruction 53. The shortest programs tend to stop or loop immediately. Only those still looping show on Figure 7. Short programs which loop on one test case tend to loop on all of them, giving the almost constant plots for short programs seen in Figure 7. Longer random programs, tend to run for longer and have more variation between the number of instructions they execute on the different test cases. Figure 8 shows there is a corresponding behaviour in terms of variation between the same program given different inputs. I.e., loops are needed to keep small programs running and compact loops mean small programs tend to keep

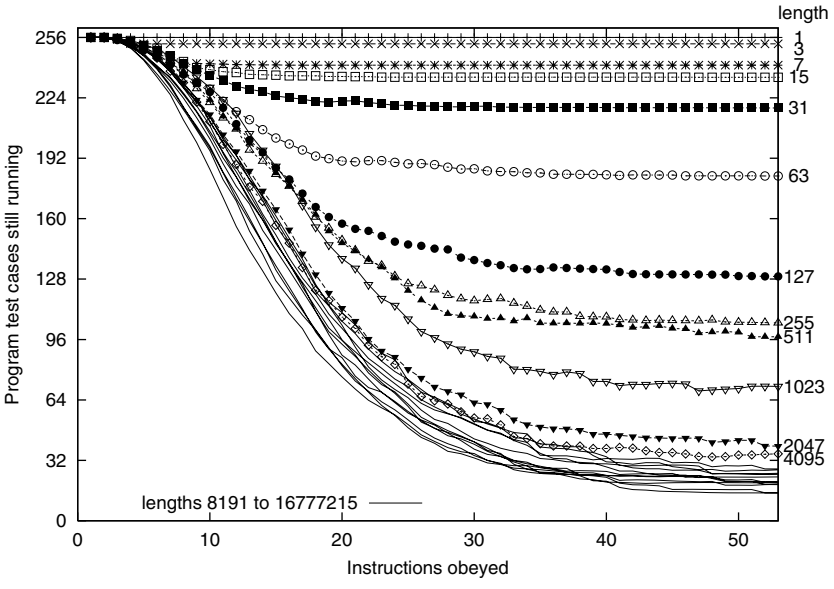


Fig. 7. Mean number of test cases where T8 program has not HALTed or stopped. 1000 random T8 programs of different lengths.

their variability. This gives the almost constant high entropy plots for short programs seen in Figure 8. However longer random programs tend to run for longer and use more instructions. More random instructions actually means that the memory etc. tends to behave the same on every input and this convergence increases as the programs run for more time. Indeed there is also less variation in average behaviour with longer random T8 programs. Leading to the general decrease in entropy with run time seen in Figure 8. In the next section we will restrict ourselves to just looking at the I/O registers rather than the whole of memory, that is the notion of programs as implementing functions which map from inputs to output. However we shall see the two views: entropy and functionality, are consistent.

5.2 Any Time T8 Convergence of Functions

There are $256^{2^4 \times 2^4} = 3.23\,10^{616}$ possible functions of two 4 bit inputs and an 8 bit output. However, as shown by Figure 9, uniformly chosen random programs of a given length sample these functions very unequally. (This is also true of the T7, cf. Figures 12 and 13). In particular the identity function and the 256 functions which return constants are much more likely than others. Figure 9 also plots two variations on the identity (where the least significant or most significant nibble implement a 4 bit identity function) and two cases of 4 bit constants. In these four cases the other 4 bits are free to vary. Note that while they represent a huge number of functions they are less frequent than either of their 8 bit namesakes.

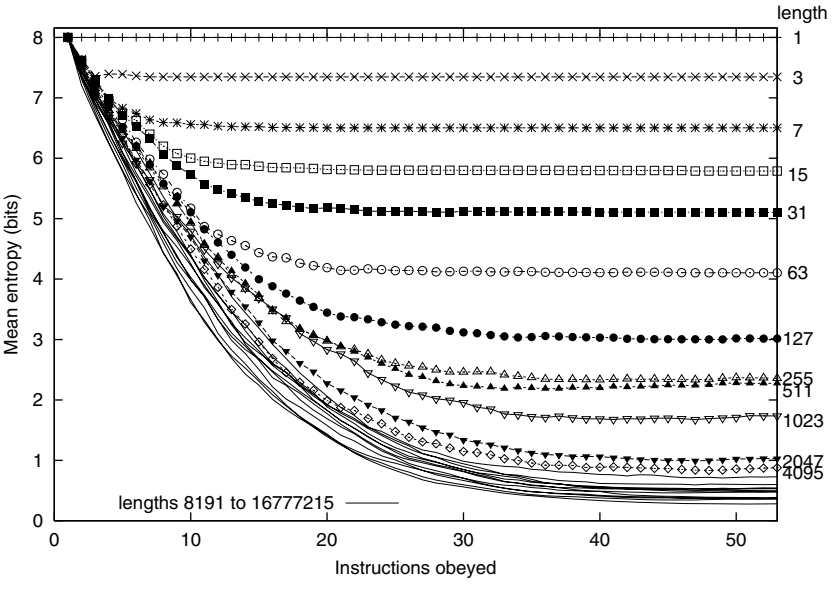


Fig. 8. Evolution of variation between test cases in 1000 random T8 programs of different lengths. Same programs as in Figure 7.

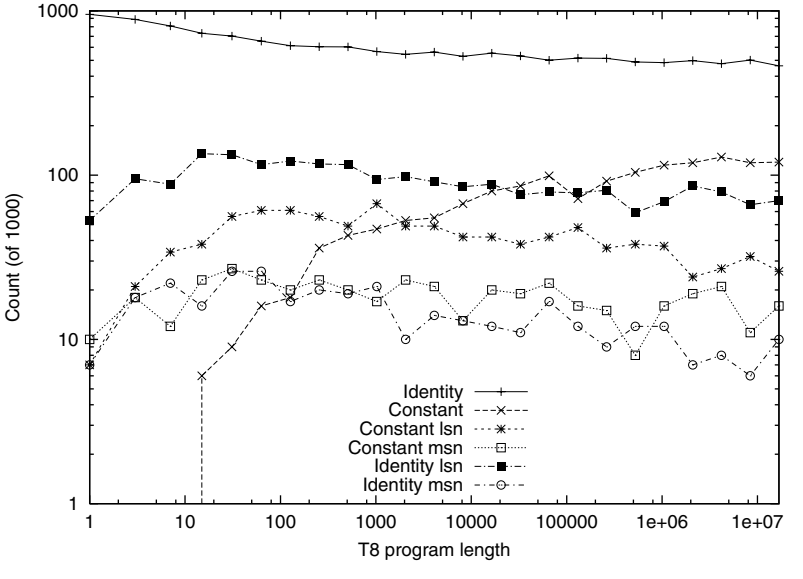


Fig. 9. Frequency of common functions implemented by random T8 programs of different lengths by 53 instruction cycles. Data are noisy due to sample size, but the trend to falling proportion of functions, except the constants, with program size can be seen.

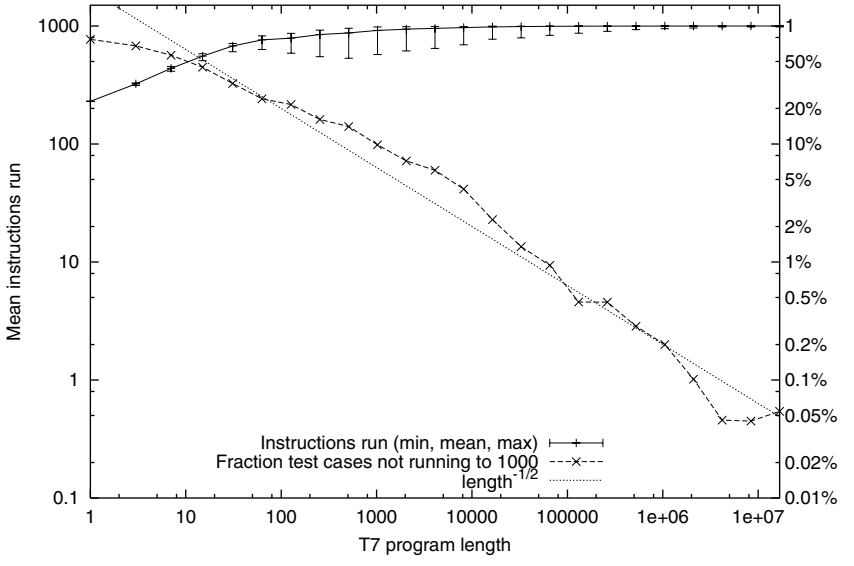


Fig. 10. Run time on 256 test cases of 1000 random T7 programs of each of a variety of lengths. Some short programs run to their end and stop but almost all longer programs run up to the limit of 1000 instructions on all test cases.

The rise in the frequency of the constant functions and fall in the identity function with increasing program size (Figure 9) are consistent with the corresponding fall in entropy seen in Figure 8. (The same will be seen in the next section for the T7, cf. Figures 12 and 14.)

6 T7 Functions and Any Time Programs

Having shown the success of the any time approach, we return to the T7. The measurements in this section are based on running the T7 on 256 test cases as in Section 5. However since without a HALT instruction, the T7 programs tend to run for much longer, we increase the any time limit from 53 to 1000 instructions.

Figure 10 confirms removing HALT does indeed mean most programs run up to the any time limit. Figure 10 relates to all 256 runs of each random program. Whilst the error bars (top solid line) show on average there is some variation between identical programs starting with different inputs for middle sized programs both very large and very small programs have the same (any time) run time. This suggests most big T7 programs loop regardless of their input. Whilst short programs halt whatever their input. Only at intermediate lengths can the input switch random programs looping/halting behaviour.

The diagonal line shows that, for program shorter than 4 000 000, the fraction of runs which stop falls approximately as $1/\sqrt{\text{length}}$, as expected. (For

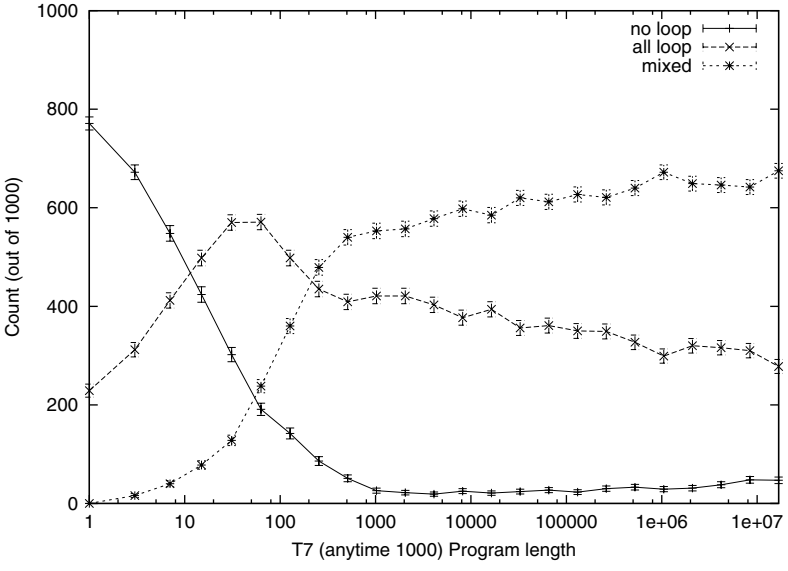


Fig. 11. Looping characteristics of Random T7 programs on 256 test cases. All programs still running are aborted after 1000 instructions. Error bars show standard error.

even longer T7 programs, the 1000 instruction limit aborts a few programs even though they are not stuck in loops.)

Figure 11 shows that the fraction of programs which never loop, falls as $O(1/\sqrt{\text{length}})$, as we found previously [3]. Figure 11 plots combined behaviour over 256 test cases rather than a single run and the data word size is half that reported in [3]. However, initially we have similar results: the fraction of T7 programs which do not loop falls with program length. Notice that for longer programs the fraction does not continue towards zero. This is because we now use the any time approach to abort non-terminated programs and so a few programs (19–47 out of 1000) are stopped early, when they might have continued to find themselves in loops.

As expected, when we allow the T7 to run for longer the variation between test cases reduces and there is an increased tendency for programs to become independent of their inputs. If a program's output does not depend on its input, i.e. all 256 test cases yield the same answer, then it effectively returns a constant. In Figure 12 the constant functions ($\times$) are those where the program's output (after up to 1000 instructions) does not depend upon its inputs. Notice the rise in the proportion of constants with program length, even though, in most cases, each program runs exactly 256×1000 instructions.

In [3] we found that longer T7 programs tend to obey more random instructions (about $\sqrt{\text{length}}$) before they get stuck in tight loops. We suggest that the rise in constants with program length in Figure 12, is due to the greater loss of information in longer random sequence of non-looping instructions before a

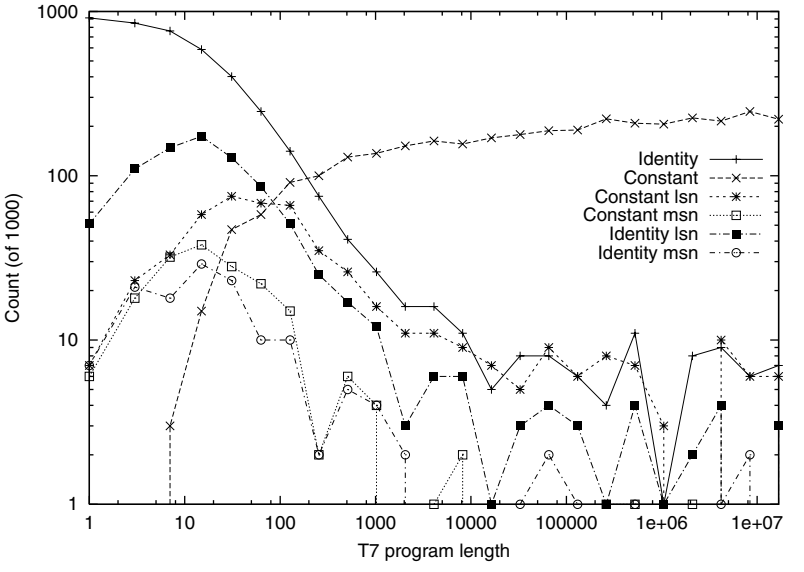


Fig. 12. Variation of some common functions implemented by T7 with program length. Note, except for the 256 constant functions, after 1000 instructions most of the functions considered in Figure 9 become rare

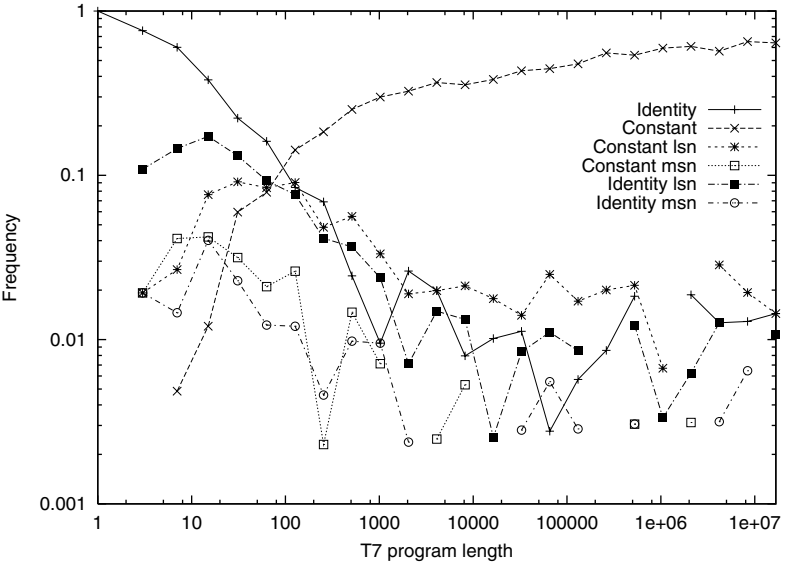


Fig. 13. Variation of some common functions implemented by T7 with program length. As Figure 12 except we include only programs which do not loop at all.

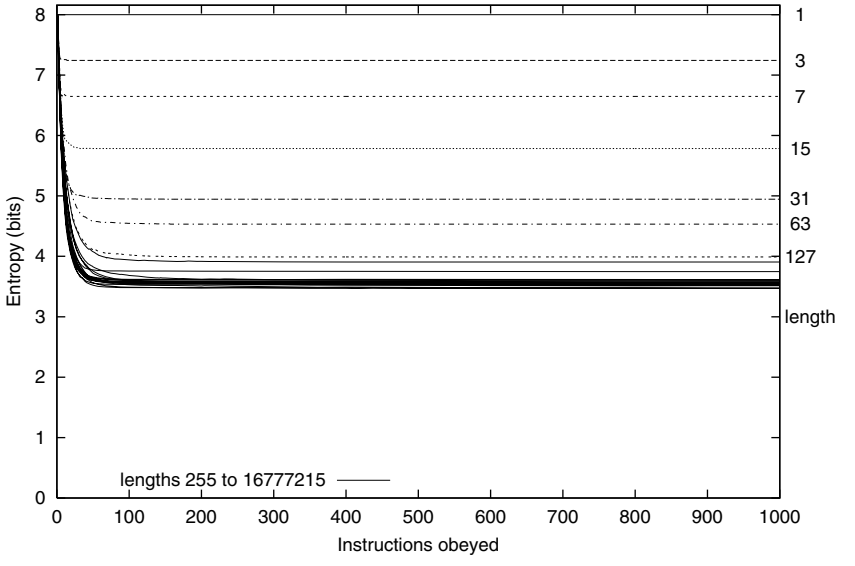


Fig. 14. Average reduction in variation between test cases for 1000 random T7 programs of each of a variety of lengths run on 256 test cases, up to 1000 steps. The smooth averages conceal strong peaks in the data at 0 (constants), 1, 2, 3, 4, 5 and 8 bits (e.g. identity).

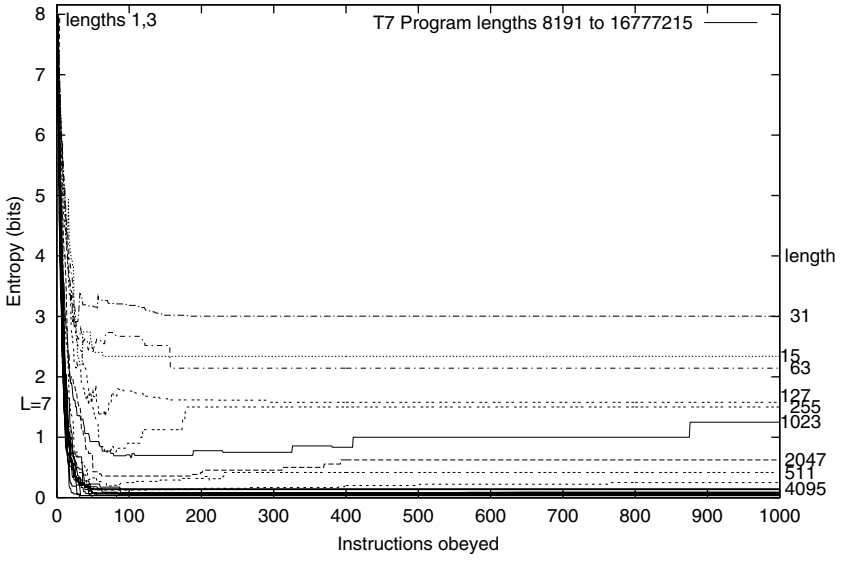


Fig. 15. As Figure 14 except we include only runs which did not loop on any test case.

tight loop is entered. Also, the loss of knowledge about the input registers in the final loop is usually either small or repeating the same instructions many times does not loose any more information.

This explanation is reinforced if we look at only the programs which did not loop. Figure 13 shows the rise in the proportion of constants with program length is even more pronounced. Whilst Figure 15 shows, if loops are excluded, in most cases variation between different input values falls rapidly to zero. Figures 14 and 15 also show test cases become more similar as the programs run. However programs which become locked into loops have less of a tendency to converge than those which are not looping. I.e. loops actually lock in variation and without them random programs are dissipative and so implement only the 2^n constant functions.

7 Discussion

Of course the undecidability of the Halting problem has long been known, however it appears to have become an excuse for not looking at unconventional approaches to evolve more powerful than O(1) functions. More recently work by Chaitin [8] started to consider a probabilistic information theoretic approach. However this is based on self-delimiting Turing machines (particularly the "Chaitin machines") and has lead to a non-zero value for Ω [9] and post-modern metamathematics. The special self-delimiting approach means halting programs cannot be extended and so each blocks out an exponentially large part of the search space. This can give very different statistics for the whole space. Our approach is firmly based on the von Neumann architecture, which for practical purposes is Turing complete. Indeed the T7 is similar to the linear genetic programming area of existing Turing complete genetic programming research.

Real computer systems lose information. We had expected this to lead to further convergence properties in programming languages with iteration and memory. However these results hint at strong differences between looping and non-looping programs. It appears that many tight loops are non-dissipative, in the sense that they cycle the computer through the same sequence of states indefinitely. In contrast, non-looping programs continue to explore the computer's state space but in doing so they become disconnected from where they started, in that they arrive at the same state regardless of where they started. This means they are useless, since they implement a constant.

Requiring input and output to be via fixed width registers is limiting. Variable sized I/O (cf. Turing tapes) is needed in general. Real CPUs achieve this by multiplexing their use of I/O registers. May be this too can be modelled.

It may be possible to obtain further results for the space of von Neumann architecture computer programs by separating the initial execution from looping. These initial experiments suggest the program path (i.e. conditional branches, jumps, etc.) of the program is initially not so important and that our earlier models on linear programs might be relevant. If, in other machines, most loops are also drawn from only a small number of types (in the case of T7 only two) it may be possible to build small predictive models of loop formation and execution.

Only a tiny fraction of the whole program is used. The rest has absolutely no effect. In future it may be possible to derive bounds on the effectiveness of testing (w.r.t. ISO 9001 requirements) based on code coverage.

8 Conclusions

The introduction of an explicit HALT instruction leads to almost all programs stopping. The geometric distribution gives an expected run time of the inverse of the frequency with which the HALT is used. This gives, in these experiments, very short run times and few interesting programs.

We also explored the any time approach, looking particularly at common functions and information theoretic measures of running programs. Entropy clearly illustrates a difference between non-dissipative looping programs and dissipative non-looping programs. There is some evidence that large random non-looping programs converge on the constant functions, however, possibly due to the size of the available memory, this is not as clear as we expected. This needs further investigation. We anticipate that detailed mathematical and Markov models could be applied to both the T7 and T8 any time approaches.

While genetic programming is perhaps the most advanced automatic programming technique, we have been analysing the fundamentals questions concerning the nature of programming search spaces. Therefore these results apply to any form of unconventional computing technique using this or similar representations which seeks to use search to create programs.

Acknowledgements. I would like to thank Dagstuhl Seminar 06061. Funded by EPSRC GR/T11234/01.

References

1. W.B. Langdon and R. Poli. *Foundations of Genetic Programming*. Springer, 2002.
2. W.B. Langdon and R. Poli. On Turing complete T7 and MISC F-4 program fitness landscapes. Technical Report CSM-445, Computer Science, Uni. Essex, UK, 2005.
3. W.B. Langdon and R. Poli. The halting probability in von Neumann architectures. In Collet *et al.* editors, *EuroGP-2006*, LNCS 3905, Springer, 2006, 225–237.
4. W. Banzhaf, P. Nordin, R.E. Keller, and F.D. Francone. *Genetic Programming – An Introduction*. Morgan Kaufmann, 1998.
5. A. Teller. Genetic programming, indexed memory, the halting problem, and other curiosities. In *FLAIRS*, Pensacola, Florida, 1994, IEEE Press, 270–274.
6. S.R. Maxwell III. Experiments with a coroutine model for genetic programming. *1994 IEEE World Congress on Computational Intelligence*, 413–417a.
7. C.E. Shannon and W. Weaver. *The Mathematical Theory of Communication*. The University of Illinois Press, Urbana, 1964.
8. G.J. Chaitin. An algebraic equation for the halting probability. In R. Herken, ed., *The Universal Turing Machine A Half-Century Survey*, Oxford Univ. Press., 1988, 279–283.
9. C.S. Calude, M.J. Dinneen, and C.-K. Shu. Computing a glimpse of randomness. *Experimental Mathematics*, 11(3):361–370, 2002.

The Number of Orbits of Periodic Box-Ball Systems

Akihiro Mikoda[1], Shuichi Inokuchi[2], Yoshihiro Mizoguchi[2], and Mitsuhiko Fujio[3]

[1] Graduate School of Mathematics,
Kyushu University, Japan,
[2] Faculty of Mathematics, Kyushu University, Japan,
ym@math.kyushu-u.ac.jp,
[3] Department of Systems Innovation and Informatics,
Kyushu Institute of Technology, Japan

Abstract. A box-ball system is a kind of cellular automata obtained by the ultradiscrete Lotka-Volterra equation. Similarities and differences between behavious of discrete systems (cellular automata) and continuous systems (differential equations) are investigated using techniques of ultradiscretizations. Our motivations is to take advantage of behavious of box-ball systems for new kinds of computations. Especially, we tried to find out useful periodic box-ball systems(pBBS) for random number generations. Applicable pBBS systems should have long fundamental cycles. We focus on pBBS with at most two kinds of solitons and investigate their behaviours, especially, the length of cycles and the number of orbits. We showed some relational equations of soliton sizes, a box size and the number of orbits. Varying a box size, we also found out some simulation results of the periodicity of orbits of pBBS with same kinds of solitons.

1 Introduction

In 1990, Takahashi and Satsuma introduced a soliton cellular automaton (SCA)[7]. The SCA is now called a box and ball system(BBS) because they explained transitions of the system using an infinite array of boxes and a finite number of balls. BBS has a property of solitons because of its transition being obtained by the ultradiscrete Lotka-Volterra equation[6,8].

In 1997, a new soliton cellular automaton is proposed by Takahashi et al[6]. That system is called box and ball system with a carrier(BBSC). BBSC can be considered as a kind of abstract model of Hyper-Threading(HT) Technology. HT Technology is a recent attractive CPU hardware technology. The main aim of HT Technology brings out the parallel efficiency of CPUs and improves the performance of a system. We hope that we could make a connection between a study of BBSC and the HT Technology in the future.

C.S. Calude et al. (Eds.): UC 2006, LNCS 4135, pp. 181–194, 2006.

Recently, the researche areas using ultradiscritizations is extending and it contains crystal formulations, combinatorics, stochastic cellular automata and algorithms[1,2,4,5].

In 2003, the notion of periodic box-ball system(pBBS) is introduced by Yoshihara et al[9]. They have shown a formula to determine the fundamental cycle of a pBBS for a given initial state. In the same year, Habu et al.[3] investigated properties about randomness and autocorrelations of configurations of pBBS and compared with Gold sequences. They showed some experimental results about their properties for a fixed system size varying the number of balls and the size of solitons.

In this paper, we focus on pBBS with at most two kinds of solitons. We re-formulate the pBBS and define sets of configurations precisely. A set of configurations with a same type is divided into some disjoint same size of orbits. We investigate the size of the configuration set and the number of orbits for designing a pBBS with a longer fundamental cycle. According to the result of Yoshihara et al.[9], we reformulate the equation of the fundamental cycles. Further, we induce the equation of the number of orbits and prove that its upperbound is not depened on the size of boxes. Finally, we show some experimental results between a size of boxes and the number of orbits.

2 Periodic Box-Ball Systems (pBBS)

Let $Q=\{0,1\}$, N a natural number, $\bar{N}=\{1,2,\ldots,N\}$ and $\overline{2N}=\{1,2,\ldots,2N\}$. We define three functions $dbl : Q^{\bar{N}} \to Q^{\overline{2N}}$, $snd : Q^{\overline{2N}} \to Q^{\bar{N}}$ and $trs : Q^{\overline{2N}} \to Q^{\overline{2N}}$ by $dbl(c)_j = c_{((j-1) \mod N)+1}$, $snd(c)_j = c_{N+j}$ and $trs(c)_j = \min\left(1-c_j, \sum_{i=1}^{j-1}(c_i - trs(c)_i)\right)$. The shift function $sft_\alpha : Q^{\bar{N}} \to Q^{\bar{N}}$ is defined by $sft_\alpha(c)_j = c_{((j-1+\alpha) \mod N)+1}$ $(\alpha = 0, \cdots, N-1)$.

Definition 1 (N-pBBS). *The periodic box-ball system with the size N (N-pBBS) is the dynamical system (C, f), where $C = \{c \in Q^{\bar{N}} | \sum_{j=1}^{N} c_j < \frac{N}{2}\}$ and the transition function $f : C \to C$ is defined by $f = snd \circ trs \circ dbl$.*

The definition of the *N-pBBS* is well-defined. It is guaranteed by the next proposition.

Proposition 1. *Assume $\#\{i \in \bar{N} | c_i = 1\} \leq \frac{N}{2}$ for $c \in Q^{\bar{N}}$.*

(1) *$\sharp\{i \in \bar{N} | c_i = 1\} = \sharp\{i \in \bar{N} | (snd \circ trs \circ dbl(c))_i = 1\}$, where $\sharp S$ is the size of the set S.*

(2) *$(snd \circ trs \circ dbl) \circ sft_\alpha(c) = sft_\alpha \circ (snd \circ trs \circ dbl)(c)$ $(\alpha = 0, 1, \cdots, N-1)$.*

The proposition is proved using the following lemma.

Lemma 1. *For $c \in Q^{\overline{N}}$, we put $\delta_j = \sum_{i=1}^{j} (dbl(c)_i - trs(dbl(c))_i)$,*

$$\Delta_j = \sum_{i=1}^{j} \left(dbl(c)_i - \overline{dbl(c)}_i \right),$$ *where $\overline{x}$ denote the complement $1-x$ for $x \in Q$. Then we have*

(1) $\delta_j = \Delta_j + \max_{1 \le i \le j} \{dbl(c)_i - \Delta_i\}$ $(j = 1, 2, \ldots, 2N)$.
(2) $\Delta_{N+j} = \Delta_N + \Delta_j$ $(j = 1, 2, \ldots, N)$.
(3) $\delta_{N+j} = \max\{\delta_N + \Delta_j, \delta_j\}$ $(j = 1, 2, \ldots, 2N)$.

The proof of Lemma 1 and Proposition 1 is listed in an appendix.

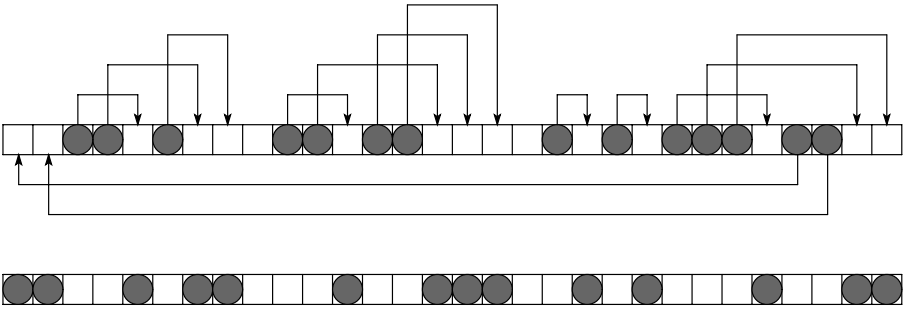


Fig. 1. Transition of pBBS

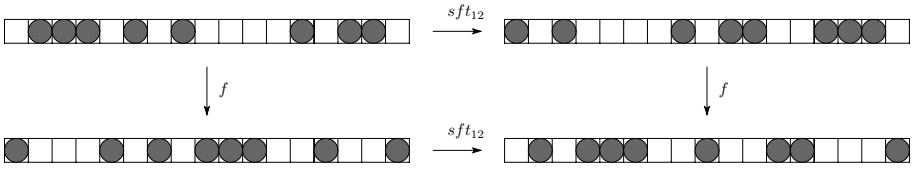


Fig. 2. Commutative diagram

Example 1. Fig. 1 is an example of a transition of pBBS with size 30. Fig. 2 is an example transition ($f \circ sft_{12} = sft_{12} \circ f$) to confirm Proposition 1(2).

Definition 2 (Fundamental cycle of a pBBS). *Let (C, f) be a pBBS with size N. The fundamental cycle of a configuration $c \in C$ is defined by $l(c) = \min\{t | f^t(c) = c, t > 0\}$.*

Yoshihara et al. classified configurations of pBBS using size of solitons $L_1, \cdots, L_s$ and introduced an equation to compute the fundamental cycle of it.

Theorem 1 (Yoshihara 2003[9]). *Let (C, f) be a pBBS with size N. If a configuration $c \in C$ has a type $(L_1, L_2, \cdots, L_s)$, then the fundamental cycle T of the configuration c is*

$$T = L.C.M\left(\frac{N_s N_{s-1}}{l_s l_0}, \frac{N_{s-1} N_{s-2}}{l_{s-1} l_0}, \cdots, \frac{N_1 N_0}{l_1 l_0}, 1\right),$$

where $l_j = L_j - L_{j+1}$ $(j = 1, 2, \cdots, s-1)$ and $N_j = l_0 + 2\sum_{i=1}^{j} n_i (L_i - L_{j+1})$. □

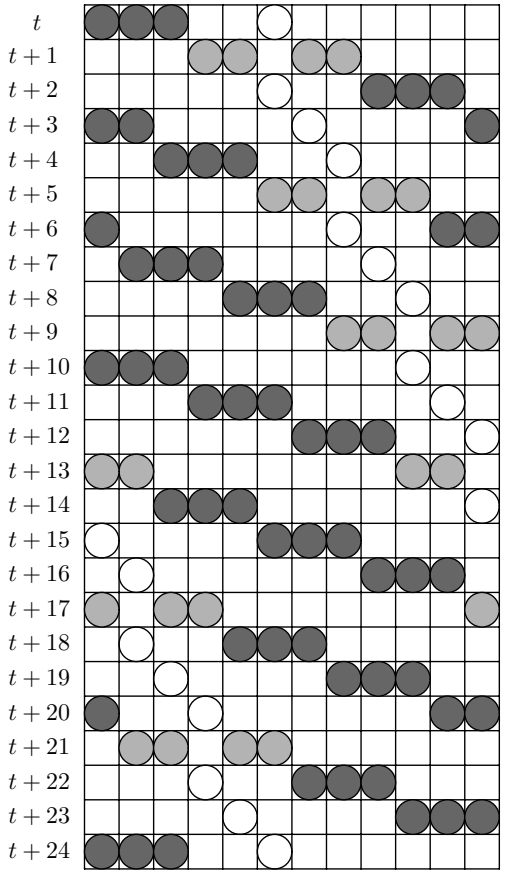


Fig. 3. Time evolution rule of pBBS

3 The Number of Orbits of a pBBS

In this section, we restrict the number of solitons up to 2. We re-formulate the class of configurations and imply a simple equation of the fundamental cycle.

We also introduce an equation of the total number of all configurations and the number of orbits.

Definition 3. *Let (C, f) be pBBS with size N. All configurations with two solitons is defined by*

$$C_2 = \{c \in C | c = 0^{x_1}1^{l_1}0^{x_2}1^{l_2}0^{x_3},\ 0 \le x_1, x_3,\ 1 \le l_1, l_2, x_2,\ x_1+l_1+x_2+l_2+x_3 = N\}.$$

For numbers L_1 and L_2 ($L_1 + L_2 < \frac{N}{2}$, $L_1 \ge L_2$), we define a set $C_{(L_1,L_2,N)}$ of configurations with a type (L_1, L_2, N) as follows:

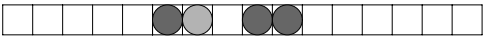
(a) *If $c = 0^{x_1}1^{l_1}0^{x_2}1^{l_2}0^{x_3}$ and $(l_1 \ge l_2,\ l_2 \le x_2)$ then $c \in C_{(l_1,l_2,N)}$ and $sft_\alpha(c) \in C_{(l_1,l_2,N)}$ for $\alpha = 0, 1, \cdots, N-1$.*
(b) *If $c = 0^{x_1}1^{l_1}0^{x_2}1^{l_2}0^{x_3}$ and $(l_1 \ge l_2,\ x_2 < l_2)$ then $c \in C_{(l_1+l_2-x_2,x_2,N)}$, and $sft_\alpha(c) \in C_{(l_1+l_2-x_2,x_2,N)}$ for $\alpha = 0, 1, \cdots, N-1$.*

We note that we can find some number L_1 and L_2 for a configuration $c = 0^{x_1}1^{l_1}0^{x_2}1^{l_2}0^{x_3}$ $(l_1 < l_2)$ to belong in $C_{(L_1,L_2,N)}$ using above Definition and sft_α.

Example 2. Let $N = 16$.

(a) $c = 0^3 1^3 0^2 1^2 0^6 \in C_{(3,2,N)}$

(b) $c = 0^5 1^2 0^1 1^2 0^6 \in C_{(3,1,N)}$



Definition 4 ((L_1, L_2, N)-pBBS). *We define a subsystem (L_1, L_2, N)-pBBS of pBBS (C, f) with size N by a dynamical system $(C_{(L_1,L_2,N)}, f)$. The fundamental cycles for all $c \in C_{(L_1,L_2,N)}$ are the same number T. We call T as the fundamental cycle of $C_{(L_1,L_2,N)}$.*

The definition of the (L_1, L_2, N)-*pBBS* is well-defined. It is guaranteed by the next proposition.

Proposition 2. (1) *$f(c) \in C_{(L_1,L_2,N)}$ for any $c \in C_{(L_1,L_2,N)}$.*
(2) *If $c_0, c_1 \in C_{(L_1,L_2,N)}$ then $l(c_0) = l(c_1)$.*
(3) *Let $\alpha = L_1 + L_2, \beta = L_1 - L_2, N = 2(L_1 + L_2) + n$. The number of configurations of (L_1, L_2, N)-pBBS is $(2\alpha + n)(2\beta + n)$.*

We denote the number $S = (2\alpha + n)(2\beta + n)$ in Proposition 2(3) by S.

Definition 5 (Orbits of pBBS). *Configuration c and d are on the same orbit if and only if $d = f^i(c)$ for some i.(cf. Fig. 4)*

$C_{(L_1,L_2,N)}$ is covered by several disjoint orbits like $\{f^i(c) | i \ge 0\}$. By Proposition 2(2), each orbits contains T elements, where T is the fundamental cycle of $C_{(L_1,L_2,N)}$.

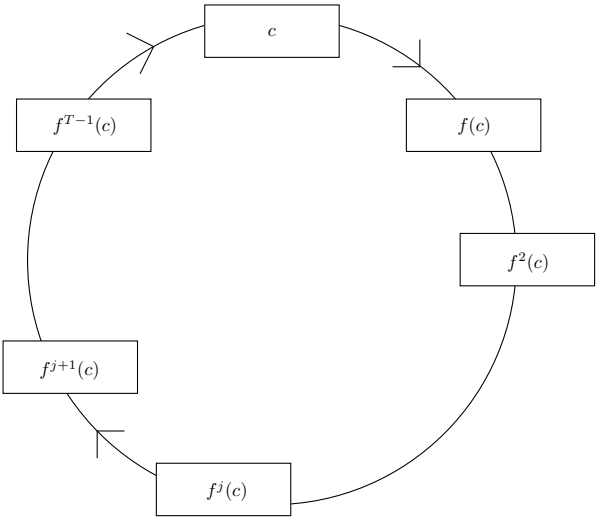


Fig. 4. The orbits of (L_1, L_2, N)-pBBS

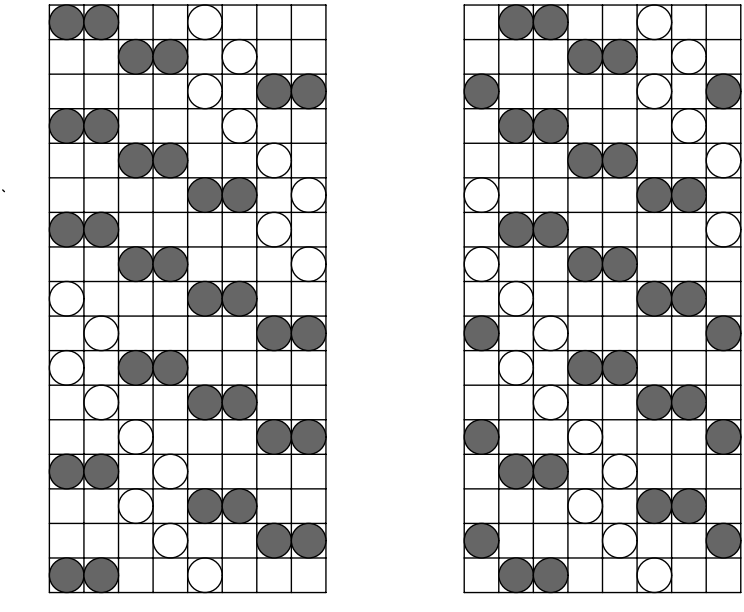


Fig. 5. The orbits of $(2, 1, 8)$-pBBS

Example 3. Fig. 5 is an example of orbit sets. $C_{(2,1,8)}$ is covered by two orbit sets where each set contains 17 elements. The fundamental cycle of c is 17 for any $c \in C_{(2,1,8)}$. $S = 34$, $T = 17$ and $K = 2$.

Theorem 2 (The number of orbits). *Let* $\alpha = L_1 + L_2, \beta = L_1 - L_2, N = 2(L_1 + L_2) + n$.

(1) *The fundamental cycle* T *of* $(L_1, L_2, 2(L_1 + L_2) + n)$*-pBBS is*
$T = L.C.M\left(\frac{(2\alpha+n)(2\beta+n)}{G.C.D\left((2\alpha+n)(2\beta+n), \frac{\alpha-\beta}{2}n\right)}, \frac{2\beta+n}{G.C.D(2\beta+n,\beta)}\right)$,

(2) *The number of orbits* K *of* $(L_1, L_2, 2(L_1 + L_2) + n)$*-pBBS is*
$K = G.C.D\left((2\alpha + n)(2\beta + n), (2\alpha + n)\beta, \frac{\alpha-\beta}{2}n\right)$.

Proof. (1) is induced by Theorem 1. Since $T = L.C.M\left(\frac{N_2N_1}{l_2l_0}, \frac{N_1N_0}{l_1l_0}, 1\right)$, $N_2 = N$, $N_1 = N - 4L_2$, $N_0 = l_0$, $l_1 = L_1 - L_2$ and $l_2 = L_2$, we have $T = L.C.M\left(\frac{N(N-4L_2)}{L_2(N-2L_1-2L_2)}, \frac{N-4L_2}{L_1-L_2}, 1\right)$. Since $\alpha = L_1 + L_2, \beta = L_1 - L_2$, we have $T = L.C.M\left(\frac{(2\alpha+n)(2\beta+n)}{G.C.D\left((2\alpha+n)(2\beta+n), \frac{\alpha-\beta}{2}n\right)}, \frac{2\beta+n}{G.C.D(2\beta+n,\beta)}\right)$.

(2) By Proposition 2(3) and above results, we have

$$
\begin{aligned}
K &= \frac{(2\alpha + n)(2\beta + n)}{L.C.M\left(\frac{(2\alpha+n)(2\beta+n)}{G.C.D\left((2\alpha+n)(2\beta+n), \frac{\alpha-\beta}{2}n\right)}, \frac{2\beta+n}{G.C.D(2\beta+n,\beta)}\right)} \\
&= \frac{2\alpha + n}{L.C.M\left(\frac{2\alpha+n}{G.C.D\left((2\alpha+n)(2\beta+n), \frac{\alpha-\beta}{2}n\right)}, \frac{1}{G.C.D(2\beta+n,\beta)}\right)} \\
&= \frac{(2\alpha + n)G.C.D\,(2\beta + n, \beta)}{L.C.M\left(\frac{(2\alpha+n)G.C.D(2\beta+n,\beta)}{G.C.D\left((2\alpha+n)(2\beta+n), \frac{\alpha-\beta}{2}n\right)}, 1\right)} \\
&= \frac{(2\alpha + n)G.C.D\,(2\beta + n, \beta)}{\frac{(2\alpha+n)G.C.D(2\beta+n,\beta)}{G.C.D\left((2\alpha+n)(2\beta+n), (2\alpha+n)\beta, \frac{\alpha-\beta}{2}n\right)}} \\
&= G.C.D\left((2\alpha + n)(2\beta + n), (2\alpha + n)\beta, \frac{\alpha - \beta}{2}n\right)
\end{aligned}
$$

□

The next theorem shows some relations between the box-size n and the number of orbits K, especially the upper bound of the number of orbits K.

Theorem 3. *Let* $\alpha = L_1 + L_2, \beta = L_1 - L_2$.

(1) $gcd(L_1 - L_2, n)|K$,
(2) $gcd(2, n)|K$,
(3) $gcd(L_1 + L_2, n)|K$, *and*
(4) $K|\frac{\alpha\beta(\alpha - \beta)}{G.C.D\,(L_1, L_2)}$.

Proof. **(1)** Let $L_1 - L_2 = ka, n = ma$. We have

$$\begin{aligned} K &= G.C.D\left((2\alpha+n)(2\beta+n), (2\alpha+n)\beta, \frac{\alpha-\beta}{2}n\right) \\ &= G.C.D\left((2\alpha+ma)(2ka+ma), (2\alpha+ma)ka, \frac{\alpha-ka}{2}ma\right) \\ &= a \times G.C.D\left((2\alpha+ma)(2k+m), (2\alpha+ma)k, \frac{\alpha-ka}{2}m\right). \end{aligned}$$

(2) Let $n = 2k$. We have

$$\begin{aligned} K &= G.C.D\left((2\alpha+n)(2\beta+n), (2\alpha+n)\beta, \frac{\alpha-\beta}{2}n\right) \\ &= G.C.D\left((2\alpha+2k)(2\beta+2k), (2\alpha+2k)\beta, \frac{\alpha-\beta}{2}2k\right) \\ &= 2 \times G.C.D\left(2(\alpha+k)(\beta+k), (\alpha+k)\beta, \frac{\alpha-\beta}{2}k\right). \end{aligned}$$

(3) Let $L_1 + L_2 = ka$, $n = ma$. We have

$$\begin{aligned} K &= G.C.D\left((2\alpha+n)(2\beta+n), (2\alpha+n)\beta, \frac{\alpha-\beta}{2}n\right) \\ &= G.C.D\left((2ka+ma)(2\beta+ma), (2ka+ma)\beta, \frac{ka-\beta}{2}ma\right) \\ &= a \times G.C.D\left((2k+m)(2\beta+ma), (2k+m)\beta, \frac{ka-\beta}{2}m\right). \end{aligned}$$

(4) Let $g = G.C.D\,(an+b, cn)$. Since $g|cn$ and $cn = G.C.D\,(a,c) \times \frac{c}{G.C.D(a,c)} \times n$, we can set $g = g_a g_c g_n$ where $g_a | G.C.D\,(a,c)$, $g_c | \frac{c}{G.C.D(a,c)}$ and $g_n | n$. Since $g_a g_n | an$ and $g_a g_n | (an+b)$, we have $g_a g_n | b$. So we can induce $g_a g_c g_n \mid \frac{bc}{G.C.D(a,c)}$.

Let $a = \beta$, $b = 2\alpha\beta$ and $c = \frac{\alpha-\beta}{2}$.

Then we have $g = G.C.D\left(2\alpha\beta + \beta n, \frac{\alpha-\beta}{2}n\right) \mid \frac{2\alpha\beta \cdot \frac{\alpha-\beta}{2}}{G.C.D(\beta, \frac{\alpha-\beta}{2})}$.

$$\begin{aligned} K &= G.C.D\left((2\alpha+n)(2\beta+n), (2\alpha+n)\beta, \frac{\alpha-\beta}{2}n\right) \\ &= G.C.D\left((2\alpha+n)(2\beta+n), G.C.D\left(2\alpha\beta+\beta n, \frac{\alpha-\beta}{2}n\right)\right) \\ &\mid G.C.D\left((2\alpha+n)(2\beta+n), \frac{\alpha\beta(\alpha-\beta)}{G.C.D\left(\beta, \frac{\alpha-\beta}{2}\right)}\right) \\ &\mid \frac{\alpha\beta(\alpha-\beta)}{G.C.D\,(L_1-L_2, L_2)} \\ &= \frac{\alpha\beta(\alpha-\beta)}{G.C.D\,(L_1, L_2)}. \end{aligned}$$

□

4 Simulations

The lefthand side of Fig. 6 is a graph of n and K for $C_{(13,2,2(13+2)+n)}$. A peak of K is 660 and $\dfrac{\alpha\beta(\alpha-\beta)}{G.C.D(L_1,L_2)} = \dfrac{15\cdot 13\cdot 2\cdot 2}{G.C.D(13,2)} = 660$. The righthand side of Fig. 6 is a graph of n and K for $C_{(12,3,2(12+3)+n)}$. A peak of K is 270 and $\dfrac{\alpha\beta(\alpha-\beta)}{G.C.D(L_1,L_2)} = \dfrac{15\cdot 9\cdot 2\cdot 3}{G.C.D(12,3)} = 270$.

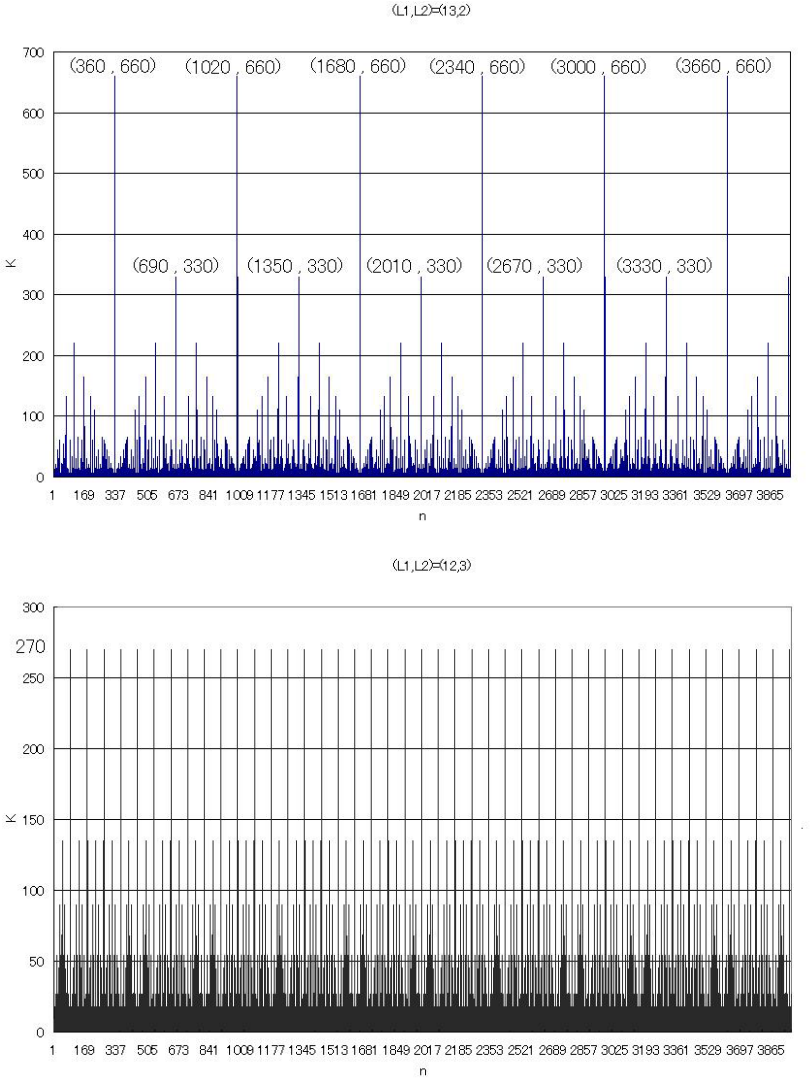


Fig. 6. Simulation results

In Theorem 3 we showed an upperbound of K. By the simulation results $\frac{\alpha\beta(\alpha-\beta)}{G.C.D(L_1,L_2)}$ may not only be an upper boud but also the maximum value of K.

Finally we have another conjecture from experimental resuts. pBBS with the number of orbits $K = 1$ must have a longer fundamental cycle, so the next conjecture may be useful to design a pBBS with a longer fundamental cycle.

Conjecture 1. Let K be the number of orbits for $C_{(L_1,L_2,2(L_1+L_2)+n)}$. If $gcd(L_1 - L_2, n) = 1$, $gcd(2, n) = 1$ and $gcd(L_1 + L_2, n) = 1$ then $K = 1$.

5 Concluding Remarks

We re-formulate the pBBS with up to 2 kinds of solitons using precise equations. We showed the formula for the fundamental cycle and the number of orbits for pBBS. Further we proved the number of orbits is bounded some constant defined by the type of solitons. This means that we can design pBBS with longer fundamental cycle if we can choose larger box size pBBS. Future works contain to investigate a expression of orbits and behaviour of orbits when we increase sorts and number of solitons.

Acknowledgments

The author thanks the anonymous reviewers for their valuable comments. This work is partially supported by Grand-in-Aid for Scientific Research, Ministry of Education, Culture, Sports, Science and Technology, Japan.

References

1. K.Fukuda. *Box-ball systems and Robinson-Schensted-Knuth correspondence.* J.Algebraic combinatorics19(2004).
2. K.Fukuda, M.Okado and Y.Yamada. *Energy Functions in Box Ball Systems.* Int.J.Mod.Phys.A15(2000)1379-1392.
3. N.Habu, K.Imazu, M.Fujio, private communications, 2003.
4. N.Konno, T.Kunimatsu and X.Ma. *From stochastic partial difference equations to stochastic cellular automata through the ultra-discretization.* Applied Mathematics and Computation.155(2004)727-735.
5. A.Nagai, D.Takahashi and T.Tokihiro. *Soliton cellular automaton,Toda molecule equation and sorting algorithm.* Phys.Lett.A255(1999)265-271.
6. D.Takahashi and J.Matsukidaira. *Box and ball system with a carrier and ultradiscrete modified KdV equation.* J.Phys.A:Math.Gen.30(1997)L733-L739.
7. D.Takahashi and J.Satsuma. *A Soliton Cellular Automaton.* J.Phys.Soc.Jpn.59 (1990)3514-3519.

8. T.Tokihiro, D.Takahashi, J.Matsukidaira and J.Satsuma. *From Soliton Equations to Integrable Cellular Automata through a Limiting Procedure.* Phys.Rev. Lett.76(1996)3247-3250.
9. D.Yoshihara, F.Yura and T.Tokihiro. *Fundamental cycle of a periodic box-ball system.* J.Phys.A:Math.Gen.36(2003)99-121.

Appendix

Proof. (of Lemma 1)

(1) Since for any c, $trs(c)_1 = 0$, we have for $j = 1$, (left hand side) $= c_1 - trs(c)_1 = c_1 =$ (right hand side). Now suppose that the equality holds for some j. Then it follows that

$$\begin{aligned}
\delta_{j+1} &= \delta_j + dbl(c)_{j+1} - trs(dbl(c))_{j+1} \\
&= \delta_j + dbl(c)_{j+1} - \min\left\{\overline{dbl(c)_{j+1}},\, \delta_j\right\} \\
&= \delta_j + \max\left\{dbl(c)_{j+1} - \overline{dbl(c)_{j+1}},\, dbl(c)_{j+1} - \delta_j\right\} \\
&= \delta_j + \max\left\{\Delta_{j+1} - \Delta_j,\, dbl(c)_{j+1} - \delta_j\right\} \\
&= \max\left\{\delta_j - \Delta_j + \Delta_{j+1},\, dbl(c)_{j+1}\right\} \\
&= \max\left\{\max_{1\le i\le j}\{dbl(c)_i - \Delta_i\} + \Delta_{j+1},\, dbl(c)_{j+1}\right\} \\
&= \Delta_{j+1} + \max\left\{\max_{1\le i\le j}\{dbl(c)_i - \Delta_i\},\, dbl(c)_{j+1} - \Delta_{j+1}\right\} \\
&= \Delta_{j+1} + \max_{1\le i\le j+1}\{dbl(c)_i - \Delta_i\},
\end{aligned}$$

which establishes the equality for $j + 1$.

(2) It follows from the fact that $dbl(c)_{N+i} = dbl(c)_i$ $(i = 1, 2, \ldots, N)$.

(3) By virtue of (1) and (2),

$$\begin{aligned}
\delta_{N+j} &= \Delta_{N+j} + \max_{1\le i\le N+j}\{dbl(c)_i - \Delta_i\} \\
&= \Delta_{N+j} + \max\left\{\max_{1\le i\le N}\{dbl(c)_i - \Delta_i\},\, \max_{1\le i\le j}\{dbl(c)_{N+i} - \Delta_{N+i}\}\right\} \\
&= \Delta_N + \Delta_j \\
&\qquad + \max\left\{\max_{1\le i\le N}\{dbl(c)_i - \Delta_i\},\, \max_{1\le i\le j}\{dbl(c)_i - \Delta_N - \Delta_i\}\right\} \\
&= \max\left\{\Delta_j + \Delta_N + \max_{1\le i\le N}\{dbl(c)_i - \Delta_i\},\right. \\
&\qquad\qquad \left.\Delta_j + \max_{1\le i\le j}\{dbl(c)_i - \Delta_i\}\right\} \\
&= \max\left\{\Delta_j + \delta_N,\, \delta_j\right\}.
\end{aligned}$$

□

Proof. (of Proposition 1).

(1) By (3) of Lemma 1, $\delta_{2N} = \max\{\delta_N + \Delta_N,\ \delta_N\}$. On the other hand, by the assumption, $\Delta_N = \sum_{i=1}^{N} \left(dbl(c)_i - \overline{dbl(c)}_i \right) = \left(\sum_{i=1}^{N} 2dbl(c)_i \right) - N = 2\left(\sum_{i=1}^{N} c_i \right) - N \leq 0$. Hence we have $\delta_{2N} = \delta_N$. This implies that

$$\begin{aligned}
&\#\{i \in \overline{N} | c_i = 1\} - \#\{i \in \overline{N} | (snd \circ trs \circ dbl(c))_i = 1\} \\
&= \left(\sum_{i=1}^{N} c_i \right) - \left(\sum_{i=1}^{N} (snd \circ trs \circ dbl(c))_i \right) \\
&= \left(\sum_{i=N+1}^{2N} d_i \right) - \left(\sum_{i=N+1}^{2N} trs(d)_i \right) \\
&= \sum_{i=N+1}^{2N} (d_i - trs(d)_i) \\
&= s_{2N} - s_N \\
&= 0.
\end{aligned}$$

(2) Since $sft_\alpha = \underbrace{sft_1 \circ \cdots \circ sft_1}_{\alpha}$, it suffices to show this for $\alpha = 1$. For the sake of simplicity, we put $d = dbl(c)$ and $e = dbl(sft_1(c))$. Then the equations are rewritten as

$$sft_1(snd(trs(d)))_j = snd(trs(e))_j \quad (j = 1, 2, \ldots, N) \tag{1}$$

Furthermore, to describe the effect of shift, we put $\delta_j = \sum_{i=1}^{j} (d_i - trs(d_i))$, $\varepsilon_j = \sum_{i=1}^{j} (e_i - trs(e_i))$, $\Delta_j = \sum_{i=1}^{j} \left(d_i - \overline{d}_i \right)$, $E_j = \sum_{i=1}^{j} (e_i - \overline{e}_i)$.

These variables are related as $\Delta_{j+1} = E_j + (c_1 - \overline{c}_1)$, $\delta_{j+1} = \max\{\varepsilon_j,\ E_j + c_1\}$ for $j = 1, 2, \ldots, 2N-1$. In fact, $\Delta_{j+1} = \sum_{i=1}^{j+1} \left(d_i - \overline{d}_i \right) = (d_1 - \overline{d}_1) + \sum_{i=1}^{j} (e_i - \overline{e}_i) = E_j + (c_1 - \overline{c}_1)$.

For the second one, by Lemma 1 (1),

$$\begin{aligned}
\delta_{j+1} &= \Delta_{j+1} + \max_{1 \leq i \leq j+1} \{d_i - \Delta_i\} \\
&= \Delta_{j+1} + \max\left\{ d_1 - \Delta_1,\ \max_{2 \leq i \leq j+1} \{d_i - \Delta_i\} \right\} \\
&= E_j + (c_1 - \overline{c}_1) + \max\left\{ \overline{c}_1,,\ \max_{1 \leq i \leq j} \{d_{i+1} - \Delta_{i+1}\} \right\}
\end{aligned}$$

$$
\begin{aligned}
&= E_j + \max\left\{c_1,\ \max_{1\le i\le j}\{d_{i+1} - \Delta_{i+1} + (c_1 - \overline{c}_1)\}\right\}\\
&= E_j + \max\left\{c_1,\ \max_{1\le i\le j}\{e_i - E_i\}\right\}\\
&= \max\left\{E_j + c_1,\ E_j + \max_{1\le i\le j}\{e_i - E_i\}\right\}\\
&= \max\{E_j + c_1,\ \varepsilon_j\}\,.
\end{aligned}
$$

Next, we claim that $sft_1(snd(trs(d)))_j$ and $snd(trs(e))_j$ are related by

$$sft_1(snd(trs(d)))_j = \max\{snd(trs(e))_j,\ \min\{\overline{e}_j,\ E_{N+j-1} + c_1\}\} \quad (2)$$

for $j = 1, 2, \ldots, N$. In fact, if $j < N$,

$$
\begin{aligned}
sft_1(snd(trs(d)))_j &= snd(trs(d))_{j+1}\\
&= trs(d)_{N+j+1}\\
&= \min\left\{\overline{d}_{N+j+1},\ \delta_{N+j}\right\}\\
&= \min\{\overline{e}_{N+j},\ \max\{\varepsilon_{N+j-1},\ E_{N+j-1} + c_1\}\}\\
&= \max\{\min\{\overline{e}_{N+j},\ \varepsilon_{N+j-1}\},\ \min\{\overline{e}_{N+j},\ E_{N+j-1} + c_1\}\}\\
&= \max\{trs(e)_{N+j},\ \min\{\overline{e}_{N+j},\ E_{N+j-1} + c_1\}\}\\
&= \max\{snd(trs(e))_j,\ \min\{\overline{e}_j,\ E_{N+j-1} + c_1\}\}\,.
\end{aligned}
$$

For $j = N$,

$$
\begin{aligned}
sft_1(snd(trs(d)))_N &= snd(trs(d))_1\\
&= trs(d)_{N+1}\\
&= \min\left\{\overline{d}_{N+1},\ \delta_N\right\}\\
&= \min\{\overline{e}_N,\ \max\{\varepsilon_{N-1},\ E_{N-1} + c_1\}\}\\
&= \max\{\min\{\overline{e}_N,\ \varepsilon_{N-1}\},\ \min\{\overline{e}_N,\ E_{N-1} + c_1\}\}\\
&= \max\{trs(e)_N,\ \min\{\overline{e}_N,\ E_{N-1} + c_1\}\}\\
&= \max\{snd(trs(e))_N,\ \min\{\overline{e}_N,\ E_{N-1} + c_1\}\}\,.
\end{aligned}
$$

Now all we have to show is that

$$snd(trs(e))_j \ge \min\{\overline{e}_j,\ E_{N+j-1} + c_1\} \quad (j = 1, 2, \ldots, N). \quad (3)$$

In fact, by combining this with the relation (2), we obtain (1).
To show (3), we apply similar argment about δ_j's and Δ_j's to ε_j's and E_j's. Recall that, from the assumption of c, it follows that $E_N \le 0$. By Lemma 1 (3), we have $\varepsilon_N = \max\{\varepsilon_N + E_N, \varepsilon_N\} = \varepsilon_{2N}$. On the other hand, by Lemma 1 (1),

$$\varepsilon_{2N} = E_{2N} + \max_{1\le i\le 2N}\{e_i - E_i\} \ \ge\ E_{2N} + e_{2N} - E_{2N} \ =\ e_{2N} \ =\ c_1.$$

Thus we have $\varepsilon_N \geq c_1$. From this it follows that

$$\begin{aligned}\varepsilon_{N+j-1} &= \max\{\varepsilon_{j-1},\, \varepsilon_N + E_{j-1}\} \\ &\geq \varepsilon_N + E_{j-1} \\ &\geq c_1 + E_{j-1} \\ &\geq c_1 + E_{j-1} + E_N \\ &= E_{N+j-1} + c_1.\end{aligned}$$

Consequently, we have

$$\begin{aligned}snd(trs(e))_j &= trs(e)_{N+j} \\ &= \min\{\overline{e}_{N+j},\, \varepsilon_{N+j-1}\} \\ &\geq \min\{\overline{e}_{N+j},\, E_{N+j-1} + c_1\} \\ &= \min\{\overline{e}_j,\, E_{N+j-1} + c_1\},\end{aligned}$$

that is, the inequality (3). □

The Euclid Abstract Machine: Trisection of the Angle and the Halting Problem

Jerzy Mycka[1,⋆], Francisco Coelho[2], and José Félix Costa[3]

[1] Institute of Mathematics,
University of Maria Curie-Sklodowska
Lublin, Poland
Jerzy.Mycka@umcs.lublin.pl
[2] Department of Mathematics,
Universidade de Évora, Portugal
fcoelho@uevora.pt
[3] Department of Mathematics,
I.S.T., Universidade Técnica de Lisboa
and CMAF – Centro de Matemática e Aplicações Fundamentais,
Lisboa, Portugal
fgc@math.ist.utl.pt

He took the golden Compasses, prepar'd
In Gods Eternal store, to circumscribe
This Universe, and all created things:
One foot he center'd, and the other turn'd
Round through the vast profunditie obscure,
And said, thus farr extend, thus farr thy bounds,
This be thy just Circumference, O World.

Milton, Paradise Lost

Abstract. What is the meaning of hypercomputation, the meaning of computing more than the Turing machine? Concrete non-computable functions always hide the halting problem as far as we know. Even the construction of a function that grows faster than any recursive function — the Busy Beaver — a more natural function, hides the halting function, that can easily be put in relation with the Busy Beaver. Is this super-Turing computation concept related only with the halting problem and its derivatives? We built an abstract machine based on the historic concept of compass and ruler construction which reveals the existence of non-computable functions not related with the halting problem. These natural, and the same time, non-computable functions can help to understand the nature of the uncomputable and the purpose, the goal, and the meaning of computing beyond Turing.

⋆ Corresponding author.

C.S. Calude et al. (Eds.): UC 2006, LNCS 4135, pp. 195–206, 2006.

1 Operations

Let us imagine a construction of algorithms acting in the framework of Euclid's geometry. We can use an infinite (in reality sufficiently large) sheet of paper, an unmarked ruler and a compass. Now we need to specify the list of possible operations.

- $P(P_1, \ldots, P_n)$ — Draw a finite number of distinct points $P_1, \ldots, P_n$.[1]
- $C(P, Q)$ — Draw the circle with the center P and going through the point Q.
- $LC(P, Q; A)$ — Give the label A to the circle with the center P and going through the point Q.
- $L(P, Q)$ — Draw the line passing through P and Q.
- $LL(P, Q; A)$ — Give the label A to the line passing through P and Q.
- $LP(O_1, O_2; A, B)$ — Give the label A to the point of the intersection of the objects (lines or circles) O_1 and O_2, in the case of two intersections choose freely the order of labeling by A and B.
- $D(A)$ — Delete the label A.
- $X \in C : n$ — If the point X is in the circle C, then execute the n-th instruction; otherwise go to the next instruction.

Of course, from the first operation we see that points are always labeled, unless labels are ultimately removed through a D instruction. Let us add that each label can be used only in the unique way, i.e., one label can identify exactly one object. This does not mean that some objects cannot have two or more labels.

A program is a numbered list of operations of the above types. After the n-th operation the next one (with the number $n + 1$) is executed, unless it is the last operation or it is the test operation $X \in C : n$.

Example 1. Let us consider the construction of two perpendicular lines. We need to start with two points P, Q, then draw the line through these points. Next we need two circles to construct a perpendicular line. Here is the code.

$01 :: P(P, Q)$
$02 :: L(P, Q)$
$03 :: LL(P, Q; A)$
$04 :: C(P, Q)$
$05 :: LC(P, Q; C)$
$06 :: D(Q)$
$07 :: LP(A, C; Q_1, Q_2)$
$08 :: C(Q_1, Q_2)$
$09 :: LC(Q_1, Q_2, C_1)$
$10 :: C(Q_2, Q_1)$
$11 :: LC(Q_2, Q_1, C_2)$

[1] We can think about this operation as a weak version of the choice axiom – we can always choose finite set of different points from Euclidean plane.

$12 :: LP(C_1, C_2; S_1, S_2)$
$13 :: L(S_1, S_2)$

By the similarly constructed programs we can give Euclid machines, which draw equilateral triangles or a bisector of some angle.

Let us consider an analogy which exists between programs of Euclid machines and theorems of Euclidean geometry.

Example 2. We can start by recalling Thales' Theorem: *An inscribed angle in a semicircle is a right angle.* How can this fact be checked by means of Euclid machines. Let us imagine the following construction.

- Draw three different, non-colinear points O, A, X.
- Draw the circle C with the center O and going through A; draw the line L through O and A, label the point of intersection of L and C by B.
- Draw the line through O and X, label the other point of the intersection of this line and C as P.
- Draw two lines: the first one going through A and P; the second one going through B and P.
- Draw the perpendicular L' for the line BP going by P.
- Label the intersection of L' and L as A'.

Now let us analyze the above part of the program (which can be translated into instructions of the Euclid machine in the obvious manner). We have constructed the angle $\angle APB$ and, after that, we have added the perpendicular L' to PB in the point P. Thus the fact that $\angle APB$ is a right angle is equivalent to the fact that the points A (the intersection of AP with L) and A' (the intersection of L' and L are identical. We can use the test operation to check the last statement, let us assume that every point is a circle with a radius of the length 0.

- $A' \in A : n$

We can use this situation to build some kind of output. For example, the program would end its activity if this condition is true; otherwise it would go into infinite loop. Or we can draw some previously chosen labels for some point (e.g. O): $+$ for the positive test; $-$ for the negative one.

In the light of the above example we can translate proposed proofs of Euclidean geometry in equivalent programs; the proof is correct if for all initial configurations we obtain the previously chosen special sign (e.g., $+$) of an acceptance.

2 *URM* Machines

In this section we present the Turing completeness of the above described geometrical machine. We use for this purpose the unlimited register machine [3] (URM). Every unlimited register machine program is a finite sequence of instructions acting on (potentially) infinite number of registers containing natural numbers. The instructions of URM machines programs can be chosen in the following way.

- $Z(n)$ — Put 0 into the n-th register.
- $S(n)$ — Increment the current value of the n-th register.
- $J(n, m, k)$ — If the values in the n-th and m-th registers are equal, jump to the k-th instruction.

2.1 The Emulation of Registers

Let us consider a program P of some URM machine, and let r be a number of registers used in this program. Then we will use a pencil of r lines to emulate these registers. The construction will be done in the following way.

- Draw two distinct points P, Q.
- Draw the line through P and Q.
- Label this line as R_1.
- Draw the perpendicular to R_1 in P.
- Label this perpendicular as R_n.
- Construct the bisector of R_1 and R_n.
- Label it as R_{n-1}.
- Construct the consecutive bisectors of R_1 and $R_{n-1}, R_{n-2}, \ldots, R_2$ and label them as $R_{n-2}, \ldots, R_1$.
- Draw the circle with the center P and going through the point Q.
- Label this circle as C.
- Label the intersections of C and $R_1, \ldots, R_n$ as $X_1, Y_1, \ldots, X_n, Y_n$.

The line R_i is used to remember values of the i-th register. The distance of point X_i to P, where X_i, *lying in the circle*, informs us about the current value which is equal to $\log_2 \frac{|PQ|}{|PX_i|}$. In the case we need to put zero into some register we should move the point X_i to the intersection of R_i and C again.

Let us add an important remark. During the whole computation (or rather drawing) the labels of the main elements of our system, i.e., the starting points P, Q, register lines R_1, ..., R_n, and the circle C will be not removed or changed.

2.2 The Translation of URM Instructions

Let us describe the translation of URM instructions into operations of Euclid machines

$Z(n)$: move the point X_n to the intersection of R_n and C
$k :: D(X_n)$
$k + 1 :: LP(R_n, C; X_n)$

$S(n)$: divide the segment PX_n into two subsegments with the same length and label the center point as X_n
$k :: C(P, X_n)$
$k + 1 :: LC(P, X_n; C_1)$
$k + 2 :: C(X_n, P)$
$k + 3 :: LC(X_n, P; C_2)$
$k + 4 :: LP(C_1, C_2; P_1, P_2)$

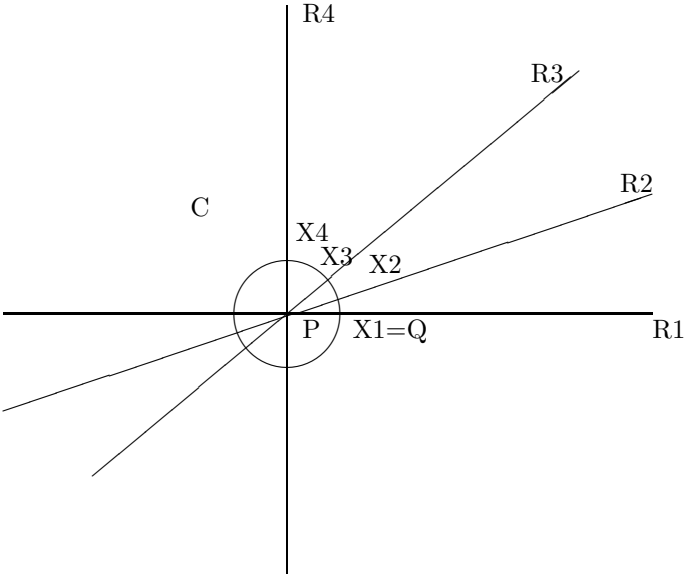


Fig. 1. Simulation of 4 registers

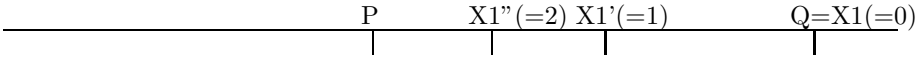


Fig. 2. Values in one register

$k+5 :: L(P_1, P_2)$
$k+6 :: LL(P_1, P_2; L)$
$k+7 :: D(X_n)$
$k+8 :: LP(L, R_n; X_n)$
$k+9 :: D(C_1)$
$k+10 :: D(C_2)$
$k+11 :: D(P_1)$
$k+12 :: D(P_2)$
$k+13 :: D(L)$

$J(n, m, s)$: test whether the point X_n is in the circle with the center P and the radius PX_m and whether the point X_m is in the circle with the center P and the radius PX_n
$k :: C(P, X_n)$
$k+1 :: LC(P, X_n; C_n)$
$k+2 :: C(P, X_m)$
$k+3 :: LC(P, X_m; C_m)$
$k+4 :: X_n \in C_m : k+6$
$k+5 :: P \in C : k+7$

$k+6 :: X_m \in C_n : k+10$
$k+7 :: D(C_n)$
$k+8 :: D(C_m)$
$k+9 :: P \in C : k+13$
$k+10 :: D(C_n)$
$k+11 :: D(C_m)$
$k+12 :: P \in C : s'$
where s' is the starting number of the Euclid corresponding instruction, equivalent to the s-th instruction of the URM machine.

Note that, in the machine above, we used the *unconditional jumping instruction* $P \in C$. This unconditional jumping could have been translated directly from the URM language into an appropriate geometrical instruction.

2.3 Euclid Machines are Turing Complete

Let us add the we have to proceed with the re-enumeration of the instructions due to the fact that every Z instruction needs 2 operations, every S instruction needs 14 operations and J needs 13 operations. With this re-enumeration we have a complete description how to translate any URM machine into some Euclid machine. So, we obtain the following proposition.

Proposition 1. *Every URM machine can be simulated by some Euclid machine.*

Example 3. Let us start with the simple example of the sum of two natural numbers. We start with a preparation of 3 lines R_1, R_2, R_3 of the same pencil with the center P, and the circle C going through these lines with the points of intersections called X_1, X_2, X_3.

$\backslash\backslash$ draw the line R_1
$01 :: P(P,Q)$
$02 :: L(P,Q)$
$03 :: LL(P,Q;R_1)$
$04 :: C(P,Q)$
$05 :: LC(P,Q;C)$
$06 :: D(Q)$
$07 :: LP(R_1,C;X_1,Y_1)$
$\backslash\backslash$ draw the perpendicular line R_3
$08 :: C(X_1,Y_1)$
$09 :: LC(X_1,Y_1,C_1)$
$10 :: C(Y_1,X_1)$
$11 :: LC(Y_1,X_1,C_2)$
$12 :: LP(C_1,C_2;S_1,S_2)$
$13 :: L(S_1,S_2)$
$14 :: LL(S_1,S_2,R_3)$
$15 :: D(C_1)$
$16 :: D(C_2)$
$17 :: D(S_1)$
$18 :: D(S_2)$
$19 :: D(Y_1)$
$20 :: LP(R_3,C;X_3,Y_3)$
$21 :: D(Y_3)$,
$\backslash\backslash$ draw the bisector of the angle R_3,P,R_1 and call it R_2
$22 :: C(X_1,P)$
$23 :: LC(X_1,P;C_1)$
$24 :: C(X_3,P)$
$25 :: LC(X_3,P;C_3)$
$26 :: LP(C_1,C_3;S_1,S_2)$
$27 :: L(S_1,S_2)$
$28 :: LL(S_1,S_2,R_2)$
$29 :: D(S_1)$
$30 :: D(S_2)$
$31 :: D(C_1)$
$32 :: D(C_3)$
$33 :: LP(R_2,C;X_2,Y_2)$
$34 :: D(Y_2)$

To start a computation for some $n, m \in \mathbb{N}$ we need to place the points X_1, X_2 on the lines R_1, R_2 in such a way that the following conditions hold: $|PX_1| =$

$\frac{|PX_1'|}{2^n}$, $|PX_2| = \frac{|PX_2'|}{2^m}$, where X_0', X_1' represent the initial position of X_1, X_2. For this purpose we need to use n times the operation $S(1)$ and m times the operation $S(2)$.

We can use the following URM machine program to implement the problem of an addition. We assume the arguments are in the registers 1 and 2; the rest of registers is initially equal to zero.

$1 : J(2,3,6)$
$2 : S(1)$
$3 : S(3)$
$4 : J(2,3,6)$
$5 : J(1,1,2)$

Now this sequence of the URM instructions can be translated into operations of the Euclid machine in the following manner.

$\backslash\backslash J(2,3,6)$
$01 :: C(P, X_2)$
$02 :: LC(P, X_2; C_2)$
$03 :: C(P, X_3)$
$04 :: LC(P, X_3; C_3)$
$05 :: X_2 \in C_3 : 7$
$06 :: P \in C : 8$
$07 :: X_3 \in C_2 : 11$
$08 :: D(C_2)$
$09 :: D(C_3)$
$10 :: P \in C : 14$
$11 :: D(C_2)$
$12 :: D(C_3)$
$13 :: P \in C : 68$
$\backslash\backslash S(1)$
$14 :: C(P, X_1)$
$15 :: LC(P, X_1; C_1)$
$16 :: C(X_1, P)$
$17 :: LC(X_1, P; C_2)$
$18 :: LP(C_1, C_2; P_1, P_2)$
$19 :: L(P_1, P_2)$
$20 :: LL(P_1, P_2; L)$
$21 :: D(X_1)$
$22 :: LP(L, R_1; X_1)$
$23 :: D(C_1)$
$24 :: D(C_2)$
$25 :: D(P_1)$
$26 :: D(P_2)$
$27 :: D(L)$
$\backslash\backslash S(3)$
$28 :: C(P, X_3)$
$29 :: LC(P, X_3; C_1)$
$30 :: C(X_3, P)$
$31 :: LC(X_3, P; C_2)$
$32 :: LP(C_1, C_2; P_1, P_2)$
$33 :: L(P_1, P_2)$
$34 :: LL(P_1, P_2; L)$
$35 :: D(X_3)$
$36 :: LP(L, R_3; X_3$
$37 :: D(C_1)$
$38 :: D(C_2)$
$39 :: D(P_1)$
$40 :: D(P_2)$
$41 :: D(L)$
$\backslash\backslash J(2,3,6)$
$42 :: C(P, X_2)$
$43 :: LC(P, X_2; C_2)$
$44 :: C(P, X_3)$
$45 :: LC(P, X_3; C_3)$
$46 :: X_2 \in C_3 : 48$
$47 :: P \in C : 49$
$48 :: X_3 \in C_2 : 52$
$49 :: D(C_2)$
$50 :: D(C_3)$
$51 :: P \in C : 55$
$52 :: D(C_2)$
$53 :: D(C_3)$
$54 :: P \in C : 68$
$\backslash\backslash J(1,1,2)$
$55 :: C(P, X_1)$
$56 :: LC(P, X_1; C_1)$
$57 :: C(P, X_1)$
$58 :: LC(P, X_1; C_1)$
$59 :: X_1 \in C_1 : 61$
$60 :: P \in C : 62$
$61 :: X_1 \in C_1 : 65$
$62 :: D(C_1)$
$63 :: D(C_1)$
$64 :: P \in C : 68$
$65 :: D(C_1)$
$66 :: D(C_1)$
$67 :: P \in C : 14$

3 Coordinates of Points

What we have shown in the preceding sections is that a suitable encoding of URM machines exist in the Cartesian plane, by performing geometric constructions using an unmarked ruler and a compass. Many other such encodings exist,

possibly more efficient. We did not really define computable functions in the sense of an Euclid-computable analogous to, e.g., the Turing-computable concept. In fact, we didn't need of that concept.

However, we can have it directly over the plan, as we are going to show in this section.

Let us recall some useful notions. A field $\mathbb{F}'$ is said to be a field extension of a field $\mathbb{F}$, if $\mathbb{F}$ is a subfield of $\mathbb{F}'$. Given some field we can extend it by several methods, for us the most natural one is to pick some elements p_j not in $\mathbb{F}$, and then to define $\mathbb{F}' = \mathbb{F}(p_j)$ as the smallest field containing $\mathbb{F}$ and all p_j. For instance, the real numbers can be extended by $i = \sqrt{-1}$ to the field of complex numbers.

In our case we are interested in points on the Euclidean plane with good (from the computational point of view) coordinates. The most convenient choice is the field $\mathbb{A}$ of algebraic numbers, which are computable and enumerable. Because we want to start with completely freely chosen points we need to extend this field by the set of all initial points (strictly speaking by the set of real, non-algebraic coordinates). Hence, for the starting points $P_1 = (x_1, y_1)$, ..., $P_k = (x_k, y_k)$ we obtain the extended field $\mathbb{A}(x_1, y_1, \ldots, x_k, y_k)$.

We can enumerate elements of such field $\mathbb{A}(x_1, y_1, \ldots, x_k, y_k)$ by natural numbers, hence the problem of any construction of points on Euclidean plane can be seen as some computation on natural numbers.

Let us precise the above remark. Every construction available with Euclid machines is done by drawing circles, lines, and finding intersections. Hence, we can obtain coordinates of these newly constructed points from the previously constructed by solving systems of equations of at most second degree. This means that new points will be also in $\mathbb{A}(x_1, y_1, \ldots, x_k, y_k)$. In this way we have the following theorem.

Proposition 2. *For any Euclid machine, with the initial points* $P_1 = (x_1, y_1)$, ..., $P_k = (x_k, y_k)$, *all points reachable have their coordinates in the field* $\mathbb{A}(x_1, y_1, \ldots, x_k, y_k)$.

If we start with points with algebraic coordinates (in $\mathbb{A}$), then all constructed points will be also (with respect to their coordinates) in $\mathbb{A}$.

Now, let us observe this fact closer for its connection with computability. Of course, there are enumerations of all points with algebraic coordinates by natural numbers, let us denote by $\nu(P)$ the index of the point P in some fixed enumeration.

Let us assume that we use a uniform method of labeling points created during the activity of an Euclid machine, for example Q_0, ..., Q_k. Then the final configuration of points can be described by the natural number obtained by any fixed coding $\langle \ldots \rangle$ of the indexes of the points $\langle \nu(Q_0), \ldots, \nu(Q_k) \rangle$. Now, we can connect with every Euclid machine some natural function, where as arguments we have $\nu(P_0)$, ..., $\nu(P_n)$ for the initial points P_0, ..., P_n and the result is given by the index of the final configuration reached during the computation (e.g., a single point). Such functions can be called Euclid computable.

4 Undecidable Problems

Let us clarify the important point. We can think about two different types of activity for Euclid's machines. The first one is connected with the described method of computation on encodings (given by points) of natural numbers. The second type of activity is simply drawing of points with a ruler and a compass. Now we need to distinguish carefully these two levels: a simulation of computations and drawings.

Let us exemplify this problem by means of the trisection problem. Angle trisection is the division of an arbitrary angle into three equal angles. It was one of the three famous geometric problems of antiquity for which solutions using only compass and ruler were sought (the other two were: circle squaring and cube duplication). The construction was proved to be impossible by Wantzel [1] only in 19th century. From this result we can infer an obvious corollary.

Proposition 3. *The problem of an angle trisection can not be solved by any Euclid machine.*

But now, we can reformulate the question about trisection. We can represent any angle $\angle AOB$ by three points A, O, B. If we restrict ourselves to points from $\mathbb{A}$, then with the use of the above mentioned coding we obtain the following new problem: does there exist such Euclid machine that given three numbers $\nu(A), \nu(O), \nu(B)$, it finds the number representation $\nu(P)$ of the point of the trisection of $\angle AOB$, i.e., $\angle AOB = 3\angle AOP$.

The first claim needing justification in this problem is the existence of such point P with algebraic coordinates. But this fact can be obtained by simple arithmetic taken from analytic geometry.

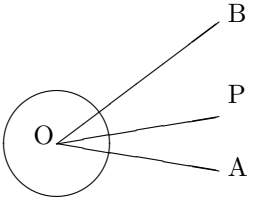


Fig. 3. Angle trisection

Proposition 4. *For any points A, O, B, with algebraic coordinates, there exists the point P with algebraic coordinates too, such that $\angle AOB = 3\angle AOP$.*

Proof. We use simple methods of analytic geometry to prove that for an angle placed in the center of a given circle (where the center of this circle and the points of intersections of the angle with this circle are given by algebraic coordinates), then the point which gives a solution of the trisection problem on this circle has also algebraic coordinates.

Without any loss of generality we can identify the point O with the origin $(0,0)$, because we can always use a translation with algebraic parameters to obtain such a situation. Now, we have two lines: OA and OB, for $A=(x_A,y_A)$, $B=(x_B,y_B)$ they have the equations: $x_Ay-y_Ax=0, x_By-y_Bx=0$. We can find now $\tan(\angle AOB)=\frac{y_Bx_A-y_Ax_B}{x_Ax_B+y_Ay_B}$. Of course, $\tan(\frac{1}{3}\angle AOB)$ can be found from the equation

$$\tan(\angle AOB)=\frac{3\tan(\frac{1}{3}\angle AOB)-\tan^3(\frac{1}{3}\angle AOB)}{1-3\tan^2(\frac{1}{3}\angle AOB)},$$

which means that $\tan(\frac{1}{3}\angle AOB)$ is an algebraic number.

The next step is devoted to compute the coefficient of the line OP given in the Cartesian plane by $y=ax$, with a given by

$$\frac{\frac{y_B}{x_B}+\tan(\frac{1}{3}\angle AOB)}{1-\frac{y_B}{x_B}\tan(\frac{1}{3}\angle AOB)}$$

(x_B can always be made different from 0 by some rotation). And now to find coordinates of P all we need is a solution of the following system of equations with algebraic coefficients: $x_P^2+y_P^2=x_A^2+y_A^2$ and $y_P=ax_P$, such systems have always algebraic solutions. □

So, now we are concerned with the crucial question. Can the number $\nu(P)$ be computed? Our first observation is that if it would be possible for some URM machine, then this process of computation could be presented in the well known manner by Euclid machines. By observation of the proof of Proposition 4 we have such the method which can be performed on (possibly infinite) decimal expansion of the coordinates (for example, by machines of Type Two Theory [4]). But our problem needs a computation on natural numbers, not on infinite sequences of digits. And, let us recall, that even if we can generate from the natural label of some algebraic number x its decimal expansion, it is impossible to obtain from finite subsequences of this expansion that natural number, which represents x (from density of the set of algebraic numbers we can always find infinite number of natural descriptions of algebraic numbers which agree with given finite sequence of digits).

But the above paragraph does not solve our problem. We can not compute the $\nu(P)$ from its decimal expansion, but maybe there is some direct method to solve this problem.

Let us assume at the moment that we have two special families of machines working on the Euclidean plane possibly with more instructions then Euclid machines. If we fix some enumeration ν of algebraic points on the Euclidean plane then for a given pencil of registers described in Section 2.1 we have the machines E_n^1 which for the given X_n register with the value k draw the point P, such that $\nu(P)=k$. Contrary, the machines E_n^2 for given point P draw the register X_n with the value equal to $\nu(P)$.

Theorem 1. *Let us define the trisection function* $T : \mathbb{N}^3 \to \mathbb{N}$ *in the following way*

$$T(\nu(A), \nu(O), \nu(B)) = \nu(P) \iff \angle AOB = 3\angle AOP.$$

Moreover, let T^* *denote the machine working on the Euclidean plane and equivalent to* T.[2] *Then the composition of* $E_n^1 \circ T^* \circ E_n^2$ *is not computable by any Euclid machine.*

Proof. With the above given machines we can draw the trisection in the following manner. First we translate points A, O, B by E_1^1, E_2^1, E_3^1 machines into X_1, X_2, X_3. Then we use the Euclid version of T to compute $\nu(P)$ in some register, e.g. X_4. Then the machine E_4^2 draws the solution of the trisection problem. If this activity could be done with Euclid machines then we would have a contradiction with Proposition 3. □

We can ask about a possibility of the trisection construction by a ruler and a compass restricted to points with algebraic coordinates. But, let us recall, the classical example of impossibility of this construction is the angle of $\frac{\pi}{3}$, which can be completely described by points with algebraic coordinates.

The above theorem creates a question about a source of Euclid non-computability of the trisection problem. We have three choices:

1. E_n^1, E_n^2 are not Euclid computable, but T is Turing computable;
2. E_n^1, E_n^2 are Euclid computable, but T is not Turing computable;
3. E_n^1, E_n^2 are not Euclid computable and T is not Turing computable.

Of course, we know that some points with algebraic coordinates can not be drawn (with fixed initial points with algebraic coordinates) by a ruler and a compass. But it is not clear whether this observation implies that E_n^1, E_n^2 - which are some transformations of points on the Euclidean plane - can not be done by Euclid machines. This consideration leads us to the following conjecture.

Conjecture. If E_n^1, E_n^2 are Euclid computable, then T is not Turing computable.

Let us observe that the above statement is always true. But we formulate it as a conjecture to stress that its non-vacuous character depends on the truth of the antecedent of the implication, which is still unknown for us.

5 Remarks

It is very interesting to observe that the trisection function does not have a character of a self-referential problem (like, e.g., the halting problem). It would be worth of explanation whether such function has any connection to classical uncomputable functions like the halting function or the busy beaver function.

We can also ask the natural question: is every Euclid computable function also Turing computable? The obvious suggestion to this question is the answer

[2] T^* uses registers in the same way as Euclid machines, but with a possibility of different instructions.

YES, by Church's thesis. Of course, we can interpret this model as a model with infinite precision, which leads us to comparison with such constructions as BSS machines. Whatever, the fully mathematical answer will need a precise construction of a proof.

Let us also add that Fourier series can be interpreted as sums of circles with decreasing *radii*. This could be used to obtain another (functional) interpretation of Euclid machines.

Acknowledgements

Part of this work was done ten years ago by Francisco Coelho in collaboration with José Félix Costa, in the context of his MSc dissertation on Diophantine equations, advised by Professor Franco de Oliveira from the Universidade de Évora. Thus we acknowledge Franco de Oliveira as friend and adviser. Let us also thank to J. F. Costa's student Bruno Loff and to Udi Boker and Nachum Dershowitz from Tel Aviv (School of Computer Science) for discussions about Theorem 8.

References

1. Martin, G.E. *Geometric Constructions*, Springer-Verlag, 1998.
2. Plouffe, S. The computation of certain numbers using a ruler and compass. *Journal of Integer Sequences*, 1, 1998.
3. Shepherdson, J.C. and Sturgis, H.E. Computability of recursive functions. *Journal of the ACM*, 10(2), 217-255, 1963.
4. Weihrauch, K. *Computable analysis, An Introduction*, Springer-Verlag, 2000.

$1/f$ Noise in Elementary Cellular Automaton Rule 110

Shigeru Ninagawa

Division of Information and Computer Science,
Kanazawa Institute of Technology,
Ohgigaoka, Nonoichi, Ishikawa 921-8501, Japan
ninagawa@infor.kanazawa-it.ac.jp

Abstract. Cellular Automata are considered to be discrete dynamical systems as well as computing systems. Spectral analysis has been employed to investigate the behavior of dynamical systems. We calculated the power spectra from the evolutions starting from a random initial configuration to analyze the temporal behavior in elementary cellular automata. As a result, rule 110 has $1/f$ spectrum for the longest time steps. Rule 110 alone has proved to be capable of supporting universal computation in elementary cellular automata. These results suggest that there is a relationship between computational universality and $1/f$ noise in cellular automata.

1 Introduction

Cellular automata (CAs) are spatially and temporally discrete dynamical systems with large degrees of freedom. In this paper we deal with one-dimensional and two-state, three-neighbor CAs which are called elementary CAs (ECAs). ECAs have been investigated in detail for their simplicity ([1], appendix in [2]). Although ECA rule space is not large, there is an interesting rule such as rule 110 which exhibits complex behavior including several types of gliders [3]. We apply power spectral method to analyze the behavior of ECAs. While it was used for the analysis of the spatial structure produced by ECAs in connection with regular languages [4], we apply it to the analysis of the temporal behavior of ECAs especially for the purpose of investigating the relationship between the computational universality and the dynamical behavior of ECAs.

2 Spectral Analysis of Elementary Cellular Automata

Let $x_i(t)$ be the value of site i at time step t in an ECA. The value of each site is specified as 0 or 1. The site value evolves by iteration of the mapping,

$$x_i(t+1) = F(x_{i-1}(t), x_i(t), x_{i+1}(t)). \tag{1}$$

Here F is an arbitrary function specifying the ECA rule. The ECA rule is determined by a binary sequence with length $2^3 = 8$,

$$F(1,1,1), F(1,1,0), \cdots, F(0,0,0). \tag{2}$$

C.S. Calude et al. (Eds.): UC 2006, LNCS 4135, pp. 207–216, 2006.

Therefore the total number of possible distinct ECA rules is $2^8 = 256$ and each rule is abbreviated by the decimal representation of the binary sequence (2) as used in [1]. Out of the 256 ECA rules 88 of them remain independent (appendix in [5]).

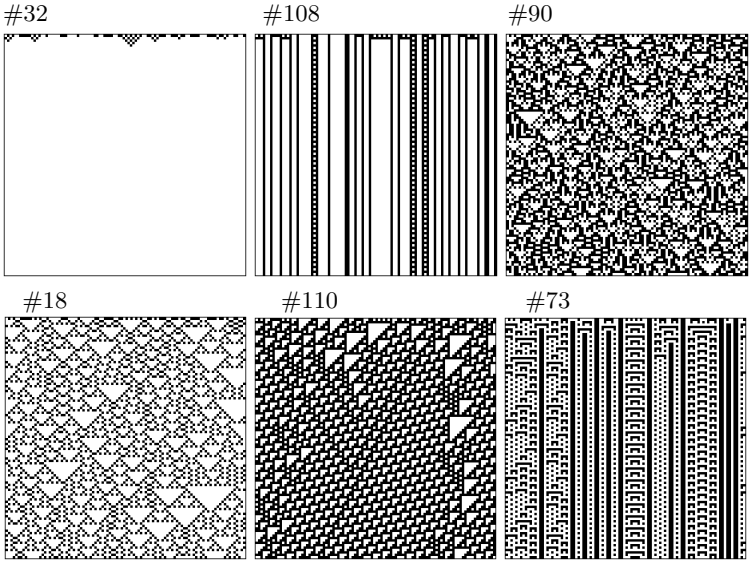


Fig. 1. Typical examples of space-time patterns of elementary cellular automata of 100 cells for 100 time steps from a random initial configuration

Configurations obtained at successive time steps in one-dimensional CA evolution are shown on successive horizontal lines in which black squares represent sites with value 1, white squares sites with value 0. It is called a space-time pattern. Figure 1 shows several examples of space-time patterns of ECA with 100 cells for 100 time steps from a random initial configuration. Throughout this paper we use random initial configurations in which each site takes state 0 or state 1 randomly with independent equal probabilities and periodic boundary conditions where each end of the array is connected like a ring.

The evolution in rule 32 leads to a homogeneous configuration in several time steps. It is called Class I in the qualitative classification scheme by Wolfram [6]. The evolution in rule 108 leads to periodic structures. It is called Class II. Rule 90 and rule 18 which are classified into Class III exhibit chaotic patterns. The evolution in rule 110 which is classified into Class IV has complex localized structures. Class IV CAs are expected to be capable of supporting universal computation [6]. Rule 73 is somewhat exceptional and is called "locally chaotic" [5] because the array is divided into some independent domains by stable "walls". Periodic or chaotic patterns are generated in each domain.

Spectral analysis is one of the useful methods to investigate the behavior of dynamical systems [7]. Therefore it is reasonable to use the spectral analysis for investigating the behavior of CAs because CAs are considered to be discrete dynamical systems.

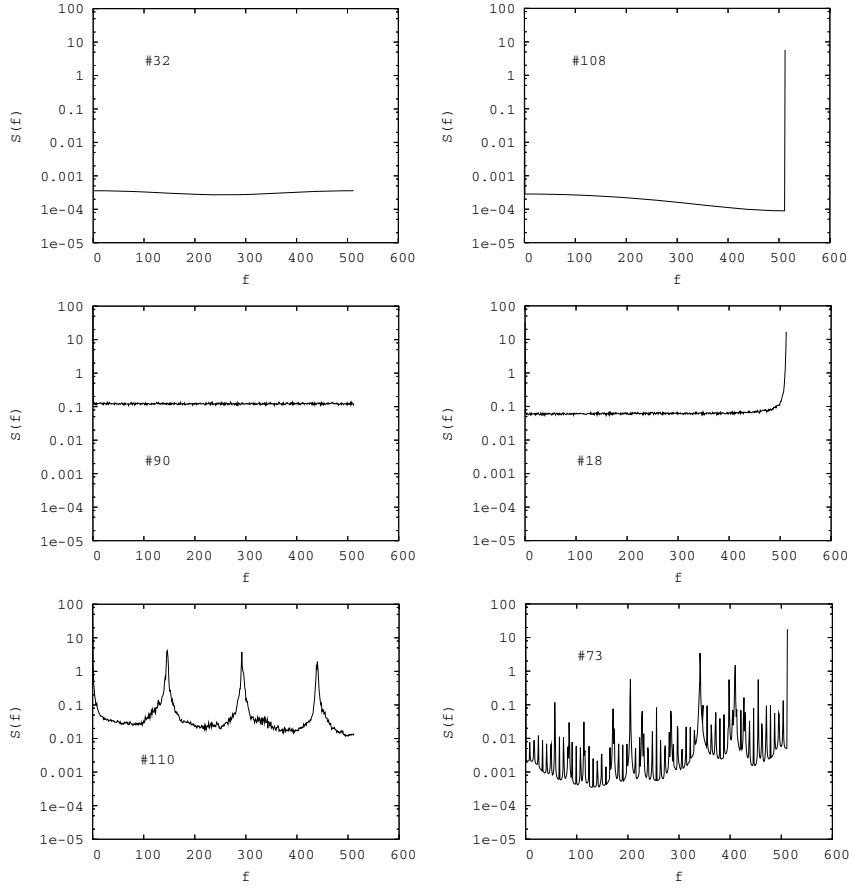


Fig. 2. Typical examples of power spectra of elementary cellular automata calculated from the evolution starting from a random initial configuration. The y-axis is plotted on a logarithmic scale. Array size is 500 and the number of observed time steps is 1024.

The discrete Fourier transform of time series of state $x_i(t)$ of the site i for $t = 0, 1, ..., T-1$ is given by

$$\hat{x}_i(f) = \frac{1}{T} \sum_{t=0}^{T-1} x_i(t) \exp(-i \frac{2\pi t f}{T}), \tag{3}$$

where T means the number of observations [8].

It is natural to define the power spectrum of CAs as

$$S(f) = \sum_i |\hat{x}_i(f)|^2, \tag{4}$$

where the summation is taken over all cells in the array. The power $S(f)$ at frequency f intuitively means the "strength" of the periodic vibration with period T/f in the evolution of T time steps.

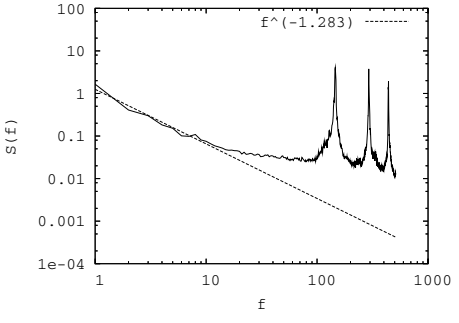


Fig. 3. The same power spectrum of rule 110 as is taken in Fig. 2. Both x and y-axis are plotted on a logarithmic scale. The dashed line represents the least square fitting of the spectrum from $f = 1$ to $f = 10$ by $\ln S(f) = \alpha + \beta \ln f$, with $\beta = -1.283$.

Figure 2 shows the typical examples of power spectra of ECAs calculated from the evolution starting from a random initial configuration of 500 cells for $T = 1024$ time steps. Only half of the components are shown since the other half are redundant.The y-axis is plotted on a logarithmic scale. The power of rule 32 is extremely low at all frequencies except $f = 0$. The extremely low power is ascribed to the initial transient behavior which vanishes in the first few time steps. The power spectrum of rule 108 is characterized by a spike in $f = 512$ which is caused by periodic structures with period $T/f = 1024/512 = 2$. The power spectrum of rule 90 is white noise which has almost equal power at all frequencies. This result implies that the evolution of rule 90 is virtually in disorder, although it is causally determined. The power spectrum of rule 18 is white noise with a peak with period $T/f = 1024/512 = 2$. The peak with period 2 in power spectrum is accumulated not by particular cells selectively but by all cells equivalently. This result shows that evolution of rule 18 exhibits chaotic behavior including periodic vibration with period 2. The power spectrum of rule 110 has not only the components in the wide range of frequencies but also peaks. The peak in $f = 146$ implies that there is a lot of periodic structures with period $T/f = 1024/146 = 7$. They are periodic background characteristic in rule 110 and called "ether". Other two peaks in $f = 292$ and $f = 440$ are the second and third harmonics respectively. The shape of power spectra of rule 73 varies abruptly depending on the behavior in the formed domains, although the peaks

of period 2($f = 512$) and 3($f = 341$) are always observed. Unlike rule 18 and rule 110, the peaks in power spectra of rule 73 are accumulated by particular cells selectively.

3 1/f Noise in Elementary Cellular Automata

Figure 3 shows the same power spectrum of rule 110 as is taken in Fig. 2, although both x and y-axis are plotted on a logarithmic scale. The dashed line represents the least square fitting of the spectrum from $f = 1$ to $f = 10$ by $\ln S(f) = \alpha + \beta \ln f$, with $\beta = -1.283$. This spectrum behaves like $S(f) \propto f^{\beta}$ with $\beta \approx -1$ at low frequencies and it is called $1/f$ noise. $1/f$ noise is a random process which has been observed in the voltage of vacuum tubes, the rate of traffic flow, the loudness of music and so on. But its origin is not well understood [9]. The lower the frequency becomes in $1/f$ noise, the more power the component has. It implies that there is a long-term correlation in the fluctuation.

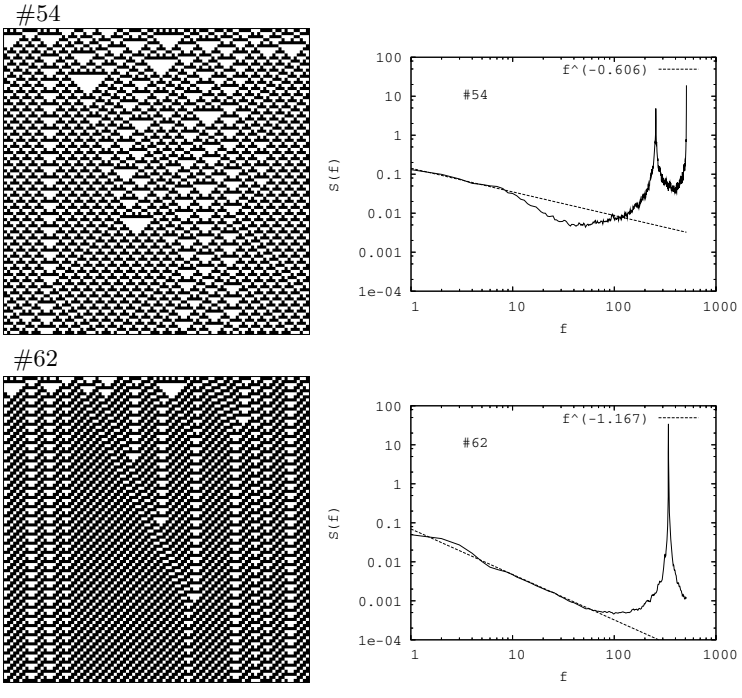


Fig. 4. Space-time patterns (left) and the power spectra (right) of rule 54 (top) and rule 62 (bottom). The space-time pattern consists of 100 cells for 100 time steps from a random initial configuration. The power spectrum is calculated from the evolution starting from a random initial configuration of 500 cells for 1024 time steps. The dashed line represents the least square fitting of the spectrum from $f = 1$ to $f = 10$ by $\ln S(f) = \alpha + \beta \ln f$, $\beta = -0.606$ for rule 54 and $\beta = -1.167$ for rule 62.

There are two other ECA rules with $1/f$ spectrum than rule 110. Those are rule 54 and rule 62 whose space-time patterns and power spectra are shown in Fig. 4. Rule 54 has been considered to be class IV but is not proved to be computationally universal. Wolfram guesses that rule 62 is not capable of supporting universal computation because the patterns produced by the rule are "in essence purely nested" (p.694 in [10]).

We investigate the value of β calculated from the least square fitting of the power spectra $\ln S(f) = \alpha + \beta \ln(f)$ from $f = 1$ to $f = 10$ for longer observed times steps $T = 2000$ to compare the long-term behavior of rule 110, 54 and 62. One-dimensional CAs in general have larger variations in the value of β with initial configurations than two-dimensional CAs. Therefore we calculate the average value of β for 400 distinct random initial configurations in the array of 200 cells. As a result, the average value of β is -1.276 in rule 110, whereas -0.467 in rule 54 and -0.638 in rule 62. This result means that rule 110 retains $1/f$ noise even in the evolution for $T = 2000$ time steps but rule 54 and rule 62 do not. In other words rule 110 has the longest duration during which power spectra exhibit $1/f$-shape in ECAs.

4 $1/f$ Noise in Rule 110

The most controversial problem in $1/f$ noise is whether $1/f$ noise lasts forever or not. If there is an upper limit of the duration time in 1/f noise, the power spectra obtained from the time series with the duration time beyond the upper limit do not exhibit power law at low frequencies. We can guess that $1/f$ spectrum in CAs is caused by transient behavior because periodic behavior does not generate $1/f$ spectrum but spikes. Generally speaking, the evolution on a finite array in CAs leads to periodic behavior sooner or later, as long as deterministic boundary conditions such as periodic boundary conditions are employed. Therefore, the spectrum turns close to flat line at low frequencies as the observed time steps T becomes long on finite array.

Figure 5 shows the average value of β calculated from the least square fitting of the power spectrum $\ln S(f) = \alpha + \beta \ln(f)$ from $f = 1$ to $f = 10$ for 400 random initial configurations as a function of array size N and the observed time steps T in rule 110. As the observed time steps T becomes larger in a fixed array size N, the average value of β becomes larger. This result can be explained as due to the reason mentioned in the previous paragraph. In case of $N = 200$ the average values of β are smaller than those in other array sizes. This result implies that the average transient time becomes long in case of array size $N = 200$. This result is contradictory to [11] which reports that the average transient time T_{ave} in rule 110 increases algebraically with array size N, $T_{ave} \propto N^{\alpha}$, with $\alpha \approx 1.08$. In [11] the comparison of periodic configurations is made up to spatial shifts. Therefore rigorous comparison of periodic configurations might result in different dependence of transient time on array size from the result in [11].

Next we compare the two evolutions from distinct random initial configurations to confirm the aforesaid guess that $1/f$ noise in CAs is caused by transient

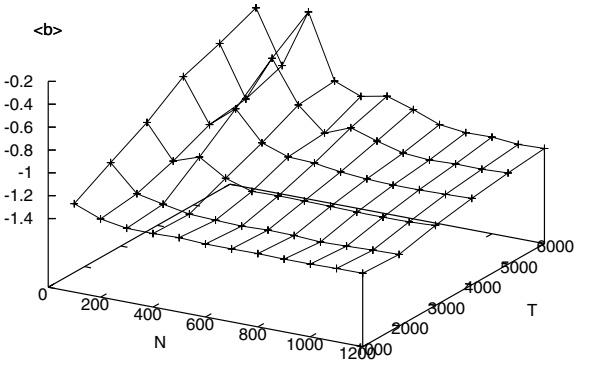


Fig. 5. Average of β calculated from the least square fitting of the power spectrum $\ln S(f) = \alpha + \beta \ln(f)$ from $f = 1$ to $f = 10$ for 400 random initial configurations as a function of array size N and the observed time steps T in rule 110

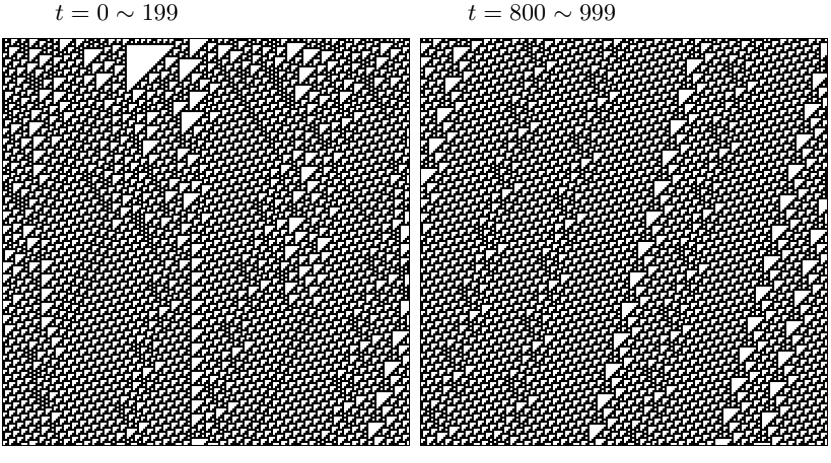


Fig. 6. Space-time patterns of rule 110 from a random initial configuration. The space-time pattern from 0 to 199 time steps is shown on the left and from 800 to 999 time steps on the right. Array size is 200. The evolution becomes periodic at 306 time steps with period 750. Several gliders are moving monotonously in the evolution from 800 to 999 time steps.

behavior. Figure 6 shows a set of space-time patterns from a random initial configuration of 200 cells in rule 110. The evolution from 0 to 199 time steps is shown on the left and from 800 to 999 time steps on the right. The evolution at first exhibits transient behavior but becomes periodic at 306 time steps with period 750. The space-time pattern from 800 to 999 time step shows that several gliders are moving monotonously in periodic background. Figure 7 shows another set of space-time patterns from another random initial configuration. The evolution keeps transient behavior at time step $t = 999$ although it eventually becomes

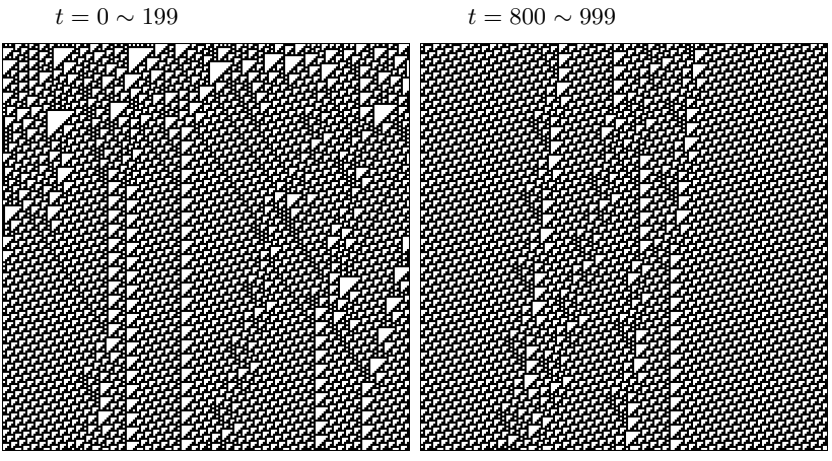


Fig. 7. Space-time patterns of rule 110 from other random initial configuration than the one used in Fig. 6. The space-time pattern from 0 to 199 time steps is shown on the left and from 800 to 999 time steps on the right. Array size is 200. Several gliders interact complexly in the evolution from 800 to 999 time steps. The evolution eventually becomes periodic at time step $t = 2416$ with period 400.

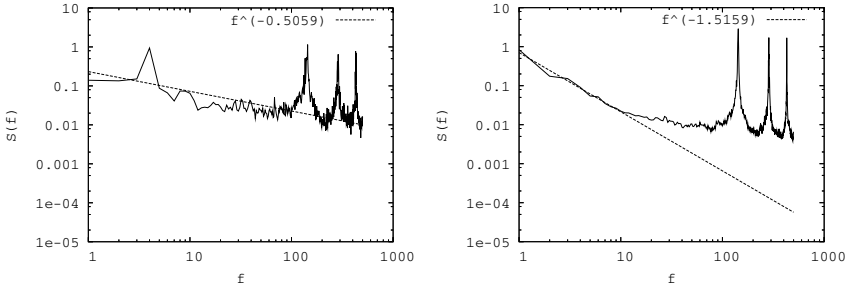


Fig. 8. Power spectra of rule 110 from random initial configurations. The left power spectrum is calculated from the evolution in Fig. 6 and the right one from Fig. 7. The dashed line represents the least square fitting of the spectrum from $f = 1$ to $f = 10$ by $\ln S(f) = \alpha + \beta \ln f$, with $\beta = -0.5059$ (left) and $\beta = -1.5159$ (right).

periodic at time step $t = 2416$ with period 400. The space-time pattern from 800 to 999 time step shows that several gliders interact complexly in periodic background. Figure 8 shows the power spectra calculated from the evolutions for $T = 1000$ time steps shown in Fig. 6 and Fig. 7. The left one is calculated from the evolution shown in Fig. 6 and the right one from Fig. 7. The dashed line represents the least square fitting of the spectrum from $f = 1$ to $f = 10$ by $\ln S(f) = \alpha + \beta \ln f$, with $\beta = -0.5059$ (left) and $\beta = -1.5159$ (right). The left power spectrum is not considered to be $1/f$-type because β is too larger than

-1, but the right one is. The comparison between these two evolutions suggests that the difference in the shape of power spectra at low frequencies depends on the duration of transient behavior.

5 Conclusion

In this research we performed the spectral analysis on the evolution of ECAs and concluded that rule 110 has $1/f$ spectra during the longest time steps in all ECAs. Rule 110 has been expected to be capable of supporting universal computation [10]. It was proved that rule 110 is computationally universal [12].

The Game of Life (LIFE) is one of the two-dimensional and two-state, nine-neighbor outer totalistic CAs [13]. Although the rule of LIFE is very simple, it generates complicated patterns such as a glider. It is supposed that a universal computer can be constructed on the array of LIFE by considering a glider as a pulse in a digital circuit. Moreover, the evolution from random initial configurations in LIFE is characterized by $1/f$ noise [14]. These results suggest that there is a relationship between computational universality and $1/f$ noise in CAs.

The hypothesis of "the edge of chaos" has evoked considerable controversy [15]. This hypothesis says the ability to perform universal computation in a system arises near a transition from regular behavior to chaotic behavior such as Class IV CAs. So far various statistical quantities, such as entropy and difference pattern spreading rate, have been proposed to detect Class IV CAs [16]. $1/f$ power spectrum might be able to be measurement of Class IV CAs.

We need to find more CAs which exhibit $1/f$ spectra to confirm the guess about the relationship between computational universality and $1/f$ noise in CAs. But most CA rule spaces except for ECAs are too large to calculate the power spectra of those rules. So we have searched for two-dimensional two-state nine-neighbor outer totalistic CA with $1/f$ spectrum using genetic algorithms (GAs) [17]. The rule obtained by the search exhibited $1/f$ spectrum. While the rule is different in two bits from that of LIFE, its behavior is extremely similar to that of LIFE, and moreover, there is the same glider as in LIFE. We are planning to search for other one-dimensional CAs with $1/f$ spectrum by GAs in future work.

References

1. Wolfram, S.: Statistical Mechanics of Cellular Automata. Reviews of Modern Physics **55** (1983) 601–644
2. Wolfram, S. (ed.): Theory and Applications of Cellular Automata, World Scientific, Singapore (1986)
3. Martínez, G.J., McIntosh, H.V., and Mora J.C.S.T.: Gliders in Rule 110. Int. Journ. of Unconventional Computing **2** (2005) 1–49
4. Li, W.: Power Spectra of Regular Languages and Cellular Automata. Complex Systems **1** (1987) 107–130
5. Li, W. and N. Packard.: The Structure of the Elementary Cellular Automata Rule Space, Complex Systems **4** (1990) 281–297

6. Wolfram, S.: Universality and Complexity in Cellular Automata. Physica D **10** (1984) 1–35
7. Crutchfield, J., Farmer, D., Packard, N., Shaw, R, Jones, G., and Donnelly, R.J.: Power Spectral Analysis of a Dynamical System. Phys. Lett. **76A** (1980) 1–4
8. Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: Numerical Recipes in C++ 2nd ed., chapter 13, Cambridge University Press, Cambridge (2002)
9. Keshner, M.S.: $1/f$ Noise. Proc. IEEE **70** (1982) 211–218
10. Wolfram, S.: A New Kind of Science, Wolfram Media (2002)
11. Li, W., Nordahl, M.G.: Transient Behavior of Cellular Automaton Rule 110. Physics Letters A **166** (1992) 335–339
12. Cook, M.: Universality in Elementary Cellular Automata. Complex Systems **15** (2004) 1–40
13. Berlekamp, E.R., Conway, J.H., Guy, R.K.: Winning Ways for Your Mathematical Plays, Vol.2, Academic Press, New York (1982)
14. Ninagawa, S., Yoneda, M., Hirose, S.: $1/f$ Fluctuation in the "Game of Life". Physica D **118** (1988) 49–52
15. Langton, C.: Computation at the Edge of Chaos: Phase Transitions and Emergent Computation. Physica D **42** (1990) 12–37
16. Li, W., Packard, N.H., and Langton, C.G.: Transition Phenomena in Cellular Automata Rule Space. Physica D **45** (1990) 77–94
17. Ninagawa, S.: Evolving Cellular Automata by $1/f$ Noise. In Capcarrere, M.S. et al.(ed.), Advances in Artificial Life, 453–460, Springer, Heidelberg (2005)

A Light-Based Device for Solving the Hamiltonian Path Problem

Mihai Oltean

Department of Computer Science,
Faculty of Mathematics and Computer Science,
Babeş-Bolyai University, Kogălniceanu 1
Cluj-Napoca, 400084, Romania
moltean@cs.ubbcluj.ro

Abstract. In this paper we suggest the use of light for performing useful computations. Namely, we propose a special device which uses light rays for solving the Hamiltonian path problem on a directed graph. The device has a graph-like representation and the light is traversing it following the routes given by the connections between nodes. In each node the rays are uniquely marked so that they can be easily identified. At the destination node we will search only for particular rays that have passed only once through each node. We show that the proposed device can solve small and medium instances of the problem in reasonable time.

1 Introduction

Using the light to perform computations is an exciting idea whose applications can be already seen on the market.

An important step was made by Intel researchers who have developed the first continuous wave all-silicon laser using a physical property called the Raman Effect [7,16,17,18]. The device could lead to such practical applications as optical amplifiers, lasers, wavelength converters, and new kinds of lossless optical devices.

Another solution comes from Lenslet [13] which has created a very fast processor for vector-matrix multiplications. This processor can perform up to 8000 Giga Multiple-Accumulate instructions per second. Lenslet technology has already been applied to data analysis using k-mean algorithm [15] and video compression.

In this paper we suggest a new way of performing computations by using some properties of light waves. The idea may be used within a special device for solving the Hamiltonian path problem.

We are not taking into account the quantum properties of light which have been used for solving the Traveling Salesman Problem [4,10].

The paper is organized as follows: The Hamiltonian path problem is briefly described in section 2. The proposed device is presented in section 3. Mathematical background of the labeling system is described in section 3.2. The way in which the proposed device works is given in section 3.3. A list of components required by the proposed device is given in section 3.4. Complexity is discussed

C.S. Calude et al. (Eds.): UC 2006, LNCS 4135, pp. 217–227, 2006.

in section 3.5. Suggestions for improving the device are given in sections 3.8 and 3.9. Further work directions are suggested in section 4.

2 The Hamiltonian Path Problem

The description of the Hamiltonian Path (HP) problem for a direct graph is the following:

Given a directed graph $G = (V, E)$ with $|V| = n$ nodes and a start node (v_{start}) and a stop node (v_{stop}), the problem asks to compute is there is a simple path, beginning with node v_{start} and ending with node v_{stop}, containing all nodes exactly once. The output for this decision problem is either YES or NO depending on whether the Hamiltonian path does exist or not.

The Hamiltonian path problem arises in many real-word applications [3,6].

The problem belongs to the class of NP-complete problems [9]. No polynomial time algorithm is known for it.

A small instance of this problem was also the first problem solved using a DNA computer [1].

3 The Proposed Device

Our idea is based on two properties of light:

- The speed of light has a limit. The value of the limit is not very important at this stage of explanation. What is important is the fact that we can delay the ray by forcing it to pass through an optical fiber cable of a certain length.
- The ray can be easily divided into multiple rays of smaller intensity/power.

Initially a light ray is sent to the start node. Generally speaking two operations must be performed when a ray passes through a node :

- The light ray is marked uniquely so that we know that it has passed through that node.
- The ray is divided into a number of rays equal to the external degree of that node. Each obtained ray is directed toward one of the nodes connected to the current node.

At the destination node we will search only for particular rays that have passed only once through each node.

This section deeply describes the proposed system. First step is to find a way to mark the signals which passes through nodes such that the interesting signals can be easily identified at the destination node. The mathematical background required for this operation is described in section 3.2 and the hardware implementation of the labeling system is described in section 3.4.

3.1 Labeling System

At the destination node we will wait for a particular ray which has passed through all nodes of the graph exactly once. This is why we need to find a way to label that particular ray so that it could be easily identified.

Actually we are interested in marking all rays which pass through a particular node with a unique label, such that Hamiltonian path is uniquely identified at the destination node (v_{stop}).

In the solution proposed in this paper, the rays passing through a node are marked by delaying them with a certain amount of time. This delay can be easily obtained by forcing the rays to pass through an optical fiber of a certain length. Roughly speaking, we will know if a certain ray has traversed a Hamiltonian path only if its delay (at the destination node) is equal to the sum of delays of all nodes in that graph. We will also know the particular moment when the expected ray (the one which has completed a Hamiltonian path) will arrive. In this case the only thing that we have to do is to "listen" if there is a fluctuation in the intensity of the signal at that particular moment. Due to the special properties of the proposed system we know that no other ray will arrive, at the destination node, at the moment when the Hamiltonian path ray has arrived.

The delays, which are introduced by each node, cannot take any values. If we would put random values for delays we might have different rays (which are not Hamiltonian paths) arriving, at the destination node, in the same time with a ray representing a Hamiltonian path.

We need only the ray, which has traversed a Hamiltonian path, to arrive in the destination node at the moment equal to the sum of delays of each node (the moment when the ray has entered in the start node is considered moment 0). Thus, the delaying system must have the following property:

Property of the delaying system

Let us denote by d_1, d_2, ..., d_n the delays introduced by each node of the graph. A correct set of values for this system must satisfy the condition:

$d_1 + d_2 + ... + d_n \neq a_1 \cdot d_1 + a_2 \cdot d_2 + ... + a_n \cdot d_n$,

where a_i $(1 \leq i \leq n)$ are natural numbers and cannot be all 1 in the same time.

If a given value a_i is strictly greater than 1 it means that the ray has passed at least twice through node 1.

3.2 Mathematical Background for the Labeling System

Finding the appropriate labeling system was two steps process. First of all we have written a computer program which generates this numbers by using a backtracking procedure [5]. We also wanted to generate numbers such that the highest number in a system is the smallest possible. This will ensure that the network is constructed in an efficient way. The labeling systems generated by our computer programs are given in Table 1.

From Table 1 it can easily seen these numbers follow a general rule:

Table 1. The labeling system generated by our backtracking procedure. First column contains the number of nodes of the graph. The second column represents the labels applied to nodes.

n	Labels (delays)
1	1
2	2, 3
3	4, 6, 7
4	8, 12, 14, 15
5	16, 24, 28, 30, 31
6	32, 48, 56, 60, 62, 63

For a graph with n nodes one of the possible labeling systems is:

$2^n - 2^{n-1}$,
$2^n - 2^{n-2}$,
$2^n - 2^{n-3}$,
... ,
$2^n - 2^0$.

As the second step we have to prove that the property of delaying system (see section 3.1) holds for this sequence of numbers.

Actually we have to prove that the equality:

$$\begin{aligned} 2^n - 2^{n-1} + 2^n - 2^{n-2} + 2^n - 2^{n-3} + ... + 2^n - 2^0 = \\ a_1 \cdot (2^n - 2^{n-1}) + a_2 \cdot (2^n - 2^{n-2}) + a_3 \cdot (2^n - 2^{n-3}) + ... + a_n \cdot (2^n - 2^0) \end{aligned} \quad (1)$$

is not possible unless all a_i are equal to 1.

The left part of the equality is:
$2^n - 2^{n-1} + 2^n - 2^{n-2} + 2^n - 2^{n-3} + ... + 2^n - 2^0 =$
$n \cdot 2^n - (2^{n-1} + 2^{n-2} + 2^{n-3} + ... + 2^0) =$
$n \cdot 2^n - 2^n + 1 =$
$(n-1) \cdot 2^n + 1.$

First of all we have to see that the equality does not hold if all a_i numbers are at least 1 and at least one number is strictly greater than 1. If this happens the 2^n term will be represented at least n times. But, it must be represented only $n-1$ times (see above).

Thus, if at least one number a_i is strictly greater than 1, it means that other numbers a_j must be 0. We will prove that the equality (1) does not hold in this case too.

As discussed above, if one of the coefficients is 0, at least one of the other coefficients must be strictly greater than 1. For instance, if a_1 is set to 0, it means that a_2 must be set to 3 in order to compensate the missing 2^{n-1} term. This is a direct consequence of the fact that $2^{n-1} = 2 \cdot 2^{n-2}$. Of course, we also

have to take into account that 2^{n-2} must be represented once. This is why we have to set a_2 to value 3.

If a_2 is also 0 we have to set a_3 to value 7 (we need $4 \cdot 2^{n-3}$ in order to compensate the missing term 2^{n-1}, we also need $2 \cdot 2^{n-3}$ to compensate the missing term 2^{n-2} and, of course, the term 2^{n-3} must be represented 1 time).

As a general idea: if a particular term (2^{n-j}) is missing (the corresponding coefficient a_j is set to 0), it can be compensate by setting one of the next coefficients to a value of at least 3. But, a coefficient set to 0 means that the term 2^n is missing once, and by setting another coefficient to at least 3 we will get at least 2 extra representations for 2^n. This will mean that right part of the equation (1) will be at least $(n+1) \cdot 2^n + 1$. But, the left part of the equation is only $(n-1) \cdot 2^n + 1$.

With this we have shown that all $a_i (1 \leq i \leq n)$ must be 1 in order to have equality in equation (1). For any other values of a_i the equality (1) does not hold.

An important question is whether this system is the minimal possible (the biggest number is the minimal possible). A partial answer to this question is given in section 3.5.

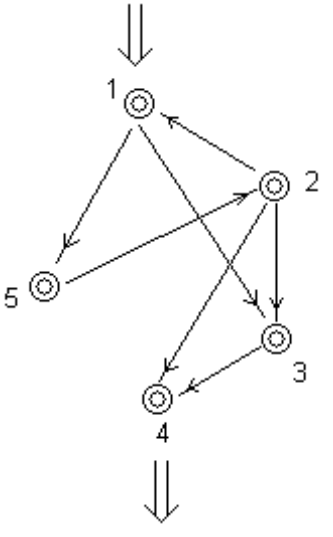


Fig. 1. A directed graph with 5 nodes. Start node is 1 and the destination node is 4. The list of arcs is: (1,3), (1, 5), (2, 1), (2, 3), (2, 4), (3, 4), (5, 2)

3.3 How the System Works

An schematic example of a graph-like system is given in Figure 1.

In the graph depicted in Figure 1 the light will enter in node 1. It will be delayed with a certain amount of time and then it will be divided into 2 rays which will be sent to the nodes 3 and 5. In node 3 the ray will be delayed (with

the amount of time corresponding to node 3) and then it will be sent to node 4. However, this is not a Hamiltonian path because it has visited only the nodes 1, 3 and 4. The other ray which was sent to node 5 (from node 1) can generate a Hamiltonian path by following the route 1, 5, 2, 3 4.

Note that there is also a cycle in the graph: 1, 5, 2. The cycle will make that some particular rays to be trapped within the system. The rays which have passed once through the previously described cycle are not considered Hamiltonian paths because the moments when they arrive at the destination node are greater than the sum of delays introduced by each node.

The partial paths traversed by rays are given below:

1
  1, 3
    1, 3, 4
  1, 5
    1, 5, 2
      1, 5, 2, 1
      .....{paths starting with 1, 5, 2, 1}
      1, 5, 2, 3
        1, 5, 2, 3, 4
      1, 5, 2, 4

3.4 Hardware Implementation of the Labeling System

For implementing the proposed device we need the following components:

- a source of light (laser),
- Several beam-splitters for dividing light rays into multiple subrays. A standard beam-splitter is designed using a half-silvered mirror. For dividing a ray into k subrays we need $k-1$ beam-splitters.
- A high speed photodiode for converting light rays into electrical power. The photodiode is placed in the destination node.
- A tool for detecting fluctuations in the intensity of electric power generated by the photodiode (oscilloscope),
- A set of optical fiber cables having certain lengths. These cables are used for connecting nodes and for delaying the signals within nodes. The length of the cables must obey the rules described in section 3.1. A practical example is given in section 3.6.

3.5 Complexity

This section answers a very important question: *Why the proposed approach is not a polynomial-time solution for the HP problem?*

At the first sight one may be tempted to say that the proposed approach provides a solution in polynomial time to any instance of the HP problem. The reason behind such claim is given by the ability of the proposed device to provide

output to any instance by traversing only once all nodes (O(n) complexity). This could mean that we have found a polynomial-time algorithm for the HP problem. A direct consequence is obtaining solutions, in polynomial time, for all other NP-Complete problems - since there is a polynomial reduction between them [9].

However, this is not our case. As can be seen from Table 1 the delay time increases exponentially with the number of nodes. Even if the ray has to traverse only n nodes (resulting a complexity of O(n)), the total time required by the ray to reach the destination node increases exponentially with the number of nodes.

There are two direct consequences which are derived from here:

- The length of the optical fibers, used for delaying the signals, increases exponentially with the number of nodes,
- The intensity of the signal decreases exponentially with the number of nodes that are traversed.

These two issues are discussed in sections 3.6 and 3.7.

3.6 Problem Size

We are interested in computing the size of the cables required to solve a certain instance of the problem in a small amount of time. This will give as a rough indication on the size of the graphs that can be solved using our system in reasonable time.

This size heavily depends on the accuracy of the measurement tools. The rise-time of the best oscilloscope available on the market is in the range of picoseconds (10^{-12} seconds). This means that we should not have two signals that arrive at 2 consecutive moments at a difference smaller than 10^{-12} seconds.

Knowing that the speed of light is $3 \cdot 10^{11} m/s$ we can easily compute the minimal cable length that should be traversed by the ray in order to be delayed with 10^{-12} seconds. This is obviously 0.3 meters.

This value is the minimal delay that should be introduced by a node in order to ensure that the difference between the moments when two consecutive signals arrive at the destination node is greater or equal to the measurable unit of 10^{-12} seconds. This will also ensure that we will be able to correctly identify whether the signal has arrived in the destination node at a moment equal to the sum of delays introduced by each node. No other signals will arrive within a range of 10^{-12} seconds around that particular moment.

Once we have the length for that minimal delay is quite easy to compute the length of the other cables that are used in order to induce a certain delay.

Recall from section 3.2, Table 1 that a graph with 5 nodes has the following delaying system:

16, 24, 28, 30, 31.

From the previous reasoning line we have deduced that the smaller indivisible unit is 0.3. So, we have to multiply these numbers by 0.3. We obtain:

4.8, 7.2, 8.4, 9.0, 9.3.

These numbers represent the length of the cables that must be used in graph's nodes in order to induce a certain delay.

Note that the delay introduced by the cables connecting the nodes was not taken into account in this example. This is not a limitation of our system. The cables connecting nodes can be set to have some length which must obey the property of delaying system (see section 3.1). Note that all cables must have the same length. In this case if we have a graph with 4 nodes the length of every cable connecting the nodes must be set to 16 units (the shortest possible - in order to reduce the costs). The length of cables within the nodes should be 24, 28, 30 and 31 units.

The largest length in this sequence is 9.3 meters. This length is not very big, but for larger graphs the length of the cables within nodes can be a problem.

Once we have the length for that minimal delay is quite easy to compute the maximal number of nodes that a graph can have in order to find the Hamiltonian path in one second. We know the facts:

- the largest delay has the form $2^n - 1$ (see equation 1),
- the distance traversed by light in 1 second is $3 \cdot 10^8$ meters,
- the shortest delay possible is 0.3 meters.

We simply have to solve the equation:

$$2^n \cdot 0.3 = 3 \cdot 10^8 \tag{2}$$

This number is about 33 nodes. However, the length of the optic fibers used for inducing the largest delay for this graph is huge: about $8 \cdot 10^{11}$ meters. We cannot expect to have such long cables for our experiments.

However, shorter cables (of several hundreds of kilometers) are already available in the internet networks. They can be easily used for our purpose. Assuming that the longest cable that we have is about 300 kilometers we may solve instances with about 17 nodes. The amount of time required to obtain a solution is about 10^{-6} seconds.

Note that the maximal number of nodes can be increased when the precision of our measurement instruments (oscilloscope and photodiode) is increased.

Also note that this difficulty is not specific to our system only. Other major unconventional computation paradigms, trying to solve NP-complete problems share the same fate. For instance, a quantity of DNA equal to the mass of Earth is required to solve HP instances of 200 cities using DNA computers [11].

3.7 Amplifying the Signal

Beam splitters are used in our approach for dividing a ray in two or more subrays. Because of that, the intensity of the signal is decreasing. In the worst case we

have an exponential decrease of the intensity. For instance, in a graph with n nodes, each signal is divided (within each node) into $n-1$ signals. Roughly speaking, the intensity of the signal will decrease n^n times.

This means that, at the destination node, we have to be able to detect very small fluctuations in the intensity of the signal. For this purpose we will use a photomultiplier [8] which is an extremely sensitive detector of light in the ultraviolet, visible and near infrared range. This detector multiplies the signal produced by incident light by as much as 10^8, from which even single photons can be detected.

3.8 Improving the Device by Reducing the Speed of the Signal

The speed of the light in optic fibers is an important parameter in our device. The problem is that the light is too fast for our measurement tools. We have either to increase the precision of our measurement tools or to decrease the speed of light.

It is known that the speed of light traversing a cable is significantly smaller than the speed of light in the void space. Commercially available cables have limit the speed of the ray wave up to 60% from the original speed of light. This means that we can obtain the same delay by using a shorter cable.

However, this method for reducing the speed of light is not enough. The order of magnitude is still the same. This is why we have the search for other methods for reducing that speed. A very interesting solution was proposed in [12,14] which is able to reduce the speed of light by 7 orders of magnitude. This could help our mechanism significantly. However, is still a question how to use this idea for our device because of the complex equipment involved in those experiments [12,14].

By reducing the speed of light by 7 orders of magnitude we can reduce the size of the involved cables by a similar order. This will help us to solve larger instances of the problem.

3.9 Improving the Performance of the Device for Particular Graphs

The labeling system proposed in section 3.1 is a general one. It can be used for any kind of graph (with any number of nodes and arcs). We have shown that this system has a big problem: the value of the involved numbers increase exponentially with the number of nodes in the graph being solved.

But, for particular graphs we can find other labeling systems which are not exponential. For example, the linear graph (see Figure 2 can be solved by our device by using virtually no delays. In this case the moment when the signal arrives in the destination node is equal to sum of delays introduced by the cables connecting the nodes.

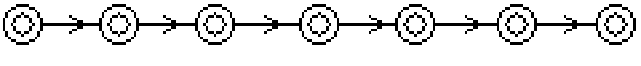


Fig. 2. A linear graph with 7 nodes. No delays are required for the nodes of this graph in order to find an Hamiltonian path

Finding the optimal labeling system for a particular graph is an interesting problem which will be investigated in the near future.

3.10 Technical Challenges

There are many technical challenges that must be solved when implementing the proposed device. Some of them are:

- Cutting the optic fibers to an exact length with high precision. Failing to accomplish this task can lead to errors in detecting a ray which has passed through each node once.
- Finding a high precision oscilloscope. This is an essential step for solving larger instances of the problem (see section 3.6),
- Finding cables long enough so that larger instances of the problem could be solve. This problem might have a simple solution: the internet networks connecting the world cities. It is easy to find cables of hundreds of kilometers connecting distant cities. This will help us to solve instance of more than 10 nodes. However, this solution introduces a difficulty too: cables of certain lengths must be found or the system must be rescaled in order to fit the existing lengths.

4 Conclusions and Further Work

The way in which light can be used for performing useful computations has been suggested in this paper. The techniques are based on the massive parallelism of the light ray.

It has been shown the way in which a light-based device can be used for solving the Hamiltonian path problem. Using the today technology we can build a light-based device which can solve small and medium size instances in several seconds.

Further work directions will be focused on:

- Implementing the proposed hardware,
- Finding optimal labeling systems for particular graphs. This will reduce the length of the involved cables significantly,
- Finding other non-trivial problems which can be solved by using the proposed device,
- Finding other ways to introduce delays in the system. The current solution requires cables that are too long and too expensive,
- Using other type of signals instead of light. A possible candidate would be electric power,
- Finding other ways to implement the system of marking the signals which pass through a particular node. This will be very useful because the currently suggested system, based on delays, is too time consuming.

References

1. Adleman, L.: Molecular computation of solutions to combinatorial problems, Science, Vol. 266, (1994) 1021-1024
2. Agrawal, G.P.: Fiber-optic communication systems, Wiley-Interscience; 3^{rd} edition, (2002)
3. Ascheuer, N.: Hamiltonian path problems in the on-line optimization of flexible manufacturing systems. PhD thesis, TU Berlin, (1995)
4. Černý, V.: Quantum computers and intractable (NP-Complete) computing problems. Phys. Rev. A, Vol. 48, (1993) 116-119
5. Cormen, T.H., Leiserson, C.E., Rivest R.R.: Introduction to algorithms, MIT Press, (1990)
6. Doniach, S., Garel, H., Orland, H.: Phase diagram of a semiflexible polymer chain in a θ solvent: Application to protein folding, Journal Of Chemical Physics, Vol. 105, (1996) 1601-1608
7. Faist, J.: Optoelectronics: silicon shines on, Nature, Vol. 433, (2005) 691-692
8. Flyckt, S.O., Marmonier, C.: Photomultiplier Tubes: Principles and Applications, Photonis, Brive, France, (2002)
9. Garey, M.R., Johnson, D.S.: Computers and intractability: A guide to NP-Completeness, Freeman & Co, San Francisco, CA, (1979)
10. Greenwood, G.W.: Finding solutions to NP problems: Philosophical differences between quantum and evolutionary search algorithms, in Proceedings CEC'2001 (2001) 815-822
11. Hartmanis, J.: On the weight of computations, Bulletin of the EATCS 55, (1995) 136-138
12. Hau L.V., (et al.): Light speed reduction to 17 meters per second in an ultracold atomic gas, Nature, Vol. 397, (1999) 594-598
13. Lenslet website, www.lenslet.com
14. Liu C., (et al.): Observation of coherent optical information storage in an atomic medium using halted light pulses, Nature, Vol. 409, (2001) 490-493
15. MacQueen, J.: Some methods for classification and analysis of multivariate observations, In LeCam, L. M., Neyman, J., (eds.) Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability. University of California press, Berkeley, (1967) 281-297
16. Paniccia, M., Koehl, S.: The silicon solution, IEEE Spectrum, IEEE Press, October (2005)
17. Rong, H., (et al.): A continuous-wave Raman silicon laser, Nature, Vol 433, (2005) 725-728
18. Rong, H., (et al.), An all-silicon Raman laser, Nature, Vol. 433, (2005) 292-294

Optimizing Potential Information Transfer with Self-referential Memory

Mikhail Prokopenko[1], Daniel Polani[2] and Peter Wang[1]

[1] CSIRO Information and Communication Technology Centre
Locked bag 17, North Ryde, NSW 1670, Australia
[2] Department of Computer Science, University of Hertfordshire
Hatfield AL10 9AB, United Kingdom
mikhail.prokopenko@csiro.au

Abstract. This paper investigates an information-theoretic design principle, intended to support an evolution of a memory structure fitting a specific selection pressure: potential information transfer through the structure. The proposed criteria measure how much does associativity in memory add to the information transfer in terms of precision, recall and effectiveness. Maximization of the latter results in holographic memory structures that can be interpreted in self-referential terms. The study introduces an analogy between self-replication and memory retrieval, with DNA as a partially-associative memory containing relevant information. DNA decoding by a complicated protein machinery ("cues" or "keys") may corresponds to an associative recall: i.e., a replicated offspring is an associatively-recalled prototype. The proposed information-theoretic criteria intend to formalize the notion of information transfer involved in self-replication, and enable bio-inspired design of more effective memory structures.

1 Introduction

Bio-inspired models have been suggested and used in many areas of Unconventional Computing: parallel processing such as Cellular Automata (CA) and DNA computation; distributed storage and transmission: e.g., neural networks and associative memory; search and optimization: e.g., genetic algorithms and ant colony optimization (ACO). New metaphors are discovered and applied at an increasing pace, improving computational models in terms of robustness, adaptivity and scalability. However, there is a certain lack of a unifying methodology, or at least a set of guiding principles, underlying many recent developments. This is unsatisfactory not only from a methodological, but also from a pragmatic point of view: if some generic principles are not utilized then specific solutions are likely to be suboptimal.

Existence of such core principles may be supported by an observation that most of the bio-inspired models listed above do not fit into a particular category of conventional computing (memory, communication, processing), but cope with multiple aspects. For instance, CA were shown by Langton [18] to support,

C.S. Calude et al. (Eds.): UC 2006, LNCS 4135, pp. 228–242, 2006.

under certain conditions (*the edge of chaos*), three basic operations of information storage, transmission, and modification, through static, propagating and interacting structures (*blinkers*, *gliders*, *collisions*). ACO algorithms also combine distributed memory, distributed transmission and distributed search, employing stigmergy — the process by which multiple ant-like agents indirectly interact through changes in their environment caused by pheromone deposits [4,5] — and resulting in emergence of optimal solutions. In other words, these fundamental aspects of dealing with information are fused together within these bio-inspired approaches, making them less brittle and more scalable than conventional systems. One compelling explanation is that the motivating biological systems (ranging from cellular tissues to ant colonies) co-evolved the computing components rather than assemble the overall architecture out of separately designed parts [11].

The main question then becomes what are the core principles that interrelate memory, communication, and processing in evolvable computational systems? Answering this question from an information-theoretic viewpoint may also improve comparability of different bio-inspired approaches. In this paper, we propose an information-theoretic design principle, intended to support an evolution of a memory structure fitting a specific selection pressure: potential information transfer through the structure. In doing so we minimize architectural assumptions about memory or processor structures, hoping instead that such dependencies emerge as a result of the optimization of the information processing dynamics. Our preliminary studies, reported here, indicate that the proposed principle is capable of clearly identifying the range and information dynamics of possible memory structures in a very general sense, enabling design of optimal memory.

The following Section points out some relevant background material on unconventional memory organization, as well as intrinsic information-theoretic fitness criteria used in evolvable computational systems. Section 3 describes the proposed measure, followed by experimental results (Section 4) and conclusions (Section 5).

2 Background and Motivation

Moskowitz and Jousselin [20] have shown that, in a general algebraic sense, the nature of the operations carried out by a computer processor actually determine the structure of the computer memory. In particular, they highlighted the hidden group structure of the address space, and pointed out that "when the integer addition law is used to manipulate addresses, this space is a cyclic group, and memory is seen as a linear array". When another composition law is used (e.g., a non-commutative address composition), a hypercubic memory structure fits more, greatly reducing complexity of computations.

Another related concept is associative or content-addressed memory: a memory organization in which the memory is accessed by its content rather than an explicit address. Reference clues or keys are "associated" with actual memory

contents until a desirable match (or set of matches) is found. A well-known example is a self-organizing map (SOM or Kohonen network). It can be interpreted as an associative memory which encodes the input patterns within the nodes of the network (the neural layer), in the form of weight (codebook) vectors of the same dimension and nature as the input patterns [17]. When a partial or corrupted pattern of data (a sensory cue) is presented in the form of a key input-vector, the rest of the pattern (memory) is associated with it. A characteristic of SOM-based associative memory is its self-organizing ordering: neighboring nodes encode similar codebook vectors, preserving topology: neurons that are closer in the neural layer tend to respond to inputs that are closer in the input space. A related approach is advocated by Kanerva [15,16]: a Sparse Distributed Memory (SDM) which is a content addressable, associative memory technique relying on close memory items clustered together: while perceived data sparsely distribute themselves over multiple storage locations, the outcome is a fusion of this distribution. In the auto-associative version of SDM the memory contents and their addresses are from the same space and may be used alternatively. Another well-known example of auto-associative memory reproducing its input pattern as output is the Hopfield neural network [10].

Importance of memory access is discussed by Goertzel [6], who pursues "not a model of how memories are physically stored in the brain or anywhere else, but rather a model of how memory access must work, of how the time required to access different memories in different situations must vary". This pursuit led towards a *structurally associative memory* (STRAM), based on the idea that "if x is more easily accessible than y, those things which are similar to x should in general be more easily accessible than those things which are similar to y" [6]. Goertzel sketched a way of mapping a weighted graph describing STRAM to a physical memory M, by assigning to each pair of elements (x, y) stored by M a distance $D_M(x, y)$ measuring the difficulty of locating x in memory given that y has very recently been located. It was suggested that the distance $D_M(x, y)$ is approximated as a number of links along the shortest path between the graph nodes corresponding to x and y.

It is worth pointing out that our approach does not intend to present just a new measure of associativity or information transfer involved in memory operations, but rather identify an information-theoretic principle contributing to a general methodology. Such a methodology may go beyond computational aspects, including sensing, actuation, and networking in distributed systems, co-evolving under multiple design/selection pressures. We intend to follow the proposal on *information-driven evolutionary design* which suggested to use fitness functions according to generic information-theoretic criteria [25,26,13,14]. The identification of possible intrinsic fitness criteria is also related to the work of Der *et al.* on self-organization of agent behaviors from domain-invariant principles, e.g., homeokinesis [3].

An example of a selection pressure is the acquisition of information from the environment: there is some evidence that pushing the information flow to the information-theoretic limit (i.e., maximization of information transfer in

perception-action loops) can give rise to intricate behaviour, induce a necessary structure in the system, and ultimately be responsible for adaptively reshaping the system [12,13,14]. Other important selection pressures applicable to distributed systems include stability of self-organizing hierarchies [22]; efficiency of multi-cellular communication topologies [23]; efficiency of locomotion and distributed actuation [25,29]. Identifying a selection pressure on potential information transfer involved in memory recall would allow us to contribute to the general methodology of information-driven evolutionary design.

3 Information Transfer: Precision and Recall

Since our task is to identify a very generic principle, we choose to abstract away from implementation details and consider instead an unconstrained deterministic function f from two equally distributed random variables K and X to a random variable Y. The variable K is intended to serve as a "key" or "cue" in accessing the memory X, retrieving, as a result of the mapping f, the outcome Y, i.e., $Y = f(K, X)$. It is important to realize that while we interpret K, X and Y as key, memory and outcome, we do not structurally constrain the variables and the mapping: e.g., there is no requirement that any location x in memory X is accessible by a unique key $k \in K$, etc.

The first constraint that we impose is the criterion:

$$\text{maximization of} \quad \mathcal{P} = I(X; K|Y) \ , \tag{1}$$

where $I(A; B)$ denotes the mutual information between A and B:

$$I(A; B) = \sum_{a \in A} \sum_{b \in B} P(a, b) \ \log \frac{P(a, b)}{P(a)P(b)} \ , \tag{2}$$

where $P(a)$ is the probability that A is in the state a, and $P(a, b)$ is the joint probability. The criterion (1) maximizes the conditional mutual information between key and memory, given the outcome. First of all, we need to clarify that, although K and X are independent and, therefore, mutual information $I(X; K)$ is zero, the conditional mutual information $I(X; K|Y)$ may well be positive. This is analogous to the example of a binary symmetric channel with input X, noise K, and output Y, described by MacKay [19] (we altered the variables names here to avoid confusion): mutual information $I(X; K) = 0$ since input and noise are independent, but $I(X; K|Y) > 0$, because "once you see the output, the unknown input and the unknown noise are intimately related!" [19]. Similarly, the criterion (1) is applied once the outcome is obtained, which means that a possible association between memory and key has been made.

Secondly, we draw an analogy with well-known information retrieval metrics: precision and recall. Precision is a measure of usefulness or *soundness* of the outcome retrieved in response to a query, and is measured as a fraction of the relevant and retrieved items within the retrieved items (aiming at "nothing but the truth"). Recall is a measure of relevance or *completeness* of the retrieved

outcome, and is measured as a ratio of the relevant and retrieved items over the relevant items (aiming at "the whole truth"). A probabilistic interpretation is possible as well [7]: precision may be defined as the conditional probability that an object is relevant given that it is returned by the system, while the recall is the conditional probability that a relevant object is returned: precision = P(relevant|returned), and recall = P(returned|relevant).

Intuitively, the criterion (1) captures the potential $\mathcal{P}$ of precision-driven information transfer. To formalize this intuition, let us apply the chain rule for the mutual information:

$$I(X;Y,K) = I(X;Y) + I(X;K|Y) \ ,$$

producing

$$\mathcal{P} = I(X;K|Y) = I(X;Y,K) - I(X;Y) \ . \tag{3}$$

The alternative representation (3) can be interpreted as follows: how much does a key *add to precision of the outcome by associating with memory*. The equation (3) contrasts two information transfers: one, $I(X;Y)$, does not use associativity, while the other, $I(X;Y,K)$, incorporates it. The difference between the two transfers captures, we believe, the potential information gain in precision. Another useful representation of the criterion (1) can be obtained in terms of entropies $H(\cdot)$, joint entropies $H(\cdot,\cdot)$, and conditional entropies $H(\cdot|\cdot)$:

$$H(A) = -\sum_{a\in A} P(a) \ \log P(a) \ , \tag{4}$$

$$H(A,B) = -\sum_{a\in A}\sum_{b\in B} P(a,b) \ \log P(a,b) \ , \tag{5}$$

$$H(B|A) = H(A,B) - H(A) \ , \tag{6}$$

where $P(a)$ is the probability that A is in the state a, $P(b)$ is the probability that B is in the state b, and $P(a,b)$ is the joint probability. We begin by applying the identity

$$I(X;K|Y) = H(X|Y) - H(X|K,Y)$$

to the right-hand side of the criterion (1). It yields

$$\mathcal{P} = I(X;K|Y) = H(X|Y) - H(X|K,Y) =$$

$$H(X|Y) - [H(X,Y,K) - H(K) - H(Y|K)] =$$

$$H(X|Y) - H(X,Y,K) + H(K) + [H(K,Y) - H(K)] =$$

$$[H(X,Y) - H(Y)] - H(X,Y,K) + H(K,Y) \ .$$

where the last three steps used relationships $H(X|K,Y) = H(X,Y,K) - H(K) - H(Y|K)$, $H(Y|K) = H(K,Y) - H(K)$ and $H(X|Y) = H(X,Y) - H(Y)$ respectively. A further reduction is possible for deterministic functions, where $H(X,Y,K)$ is a constant, making the criterion (1) equivalent to

$$\text{maximization of} \quad \tilde{\mathcal{P}} = H(X,Y) - H(Y) + H(K,Y) \ . \tag{7}$$

The measure $\tilde{\mathcal{P}}$ may, of course, be rewritten as follows:

$$\tilde{\mathcal{P}} = H(X|Y) + H(K,Y) = H(X,Y) + H(K|Y) \ . \tag{8}$$

At this stage we would like to introduce another criterion. We consider

$$\text{maximization of} \quad \mathcal{R} = I(Y;K|X) = I(Y;X,K) - I(Y,X) \ . \tag{9}$$

Intuitively, $\mathcal{R}$ measures how much a key is necessary *to identify the output of the mapping, given the memory*. The criterion (9) captures the potential $\mathcal{R}$ of information transfer involved in the memory recall, and aims to maximize the difference between associative and non-associative information transfer. Using similar unfolding, we obtain

$$\mathcal{R} = I(Y;K|X) = H(Y|X) - H(Y|K,X) =$$

$$H(Y|X) - [H(Y,X,K) - H(K) - H(X|K)] =$$

$$H(Y|X) - H(Y,X,K) + H(K) + [H(X,K) - H(K)] =$$

$$[H(Y,X) - H(X)] - H(Y,X,K) + H(X,K) \ .$$

Since for deterministic functions the last three entropies of the right-hand side are constants (and $H(Y,X,K) = H(X,K)$ anyway), maximization of $\mathcal{R}$ is equivalent to

$$\text{maximization of} \quad \tilde{\mathcal{R}} = H(X,Y) \ . \tag{10}$$

It should be noted that since $Y = f(K,X)$, the expression for $\tilde{R}$ is dependent on K.

The overall effectiveness of information retrieval is typically defined as the harmonic mean (the reciprocal of the arithmetic mean of the reciprocals) of recall and precision — hence, we suggest the criterion:

$$\text{maximization of} \quad \mathcal{E} = \frac{2}{\frac{1}{\mathcal{P}} + \frac{1}{\mathcal{R}}} = \frac{2\mathcal{P}\mathcal{R}}{\mathcal{P} + \mathcal{R}} \ , \tag{11}$$

fusing together the potential information gains in both precision and recall.

In order to highlight different roles played by K and X, we consider here scenarios with varying sizes $\|K\|$ and $\|X\|$, interpreted in the context of several examples: (a) catalog/book indexing and search; (b) pattern association using a neural network; (c) decoding of genotype (DNA) by proteins. The scenarios are as follows:

(S_1) $\|K\| \gg \|X\|$ and $\|X\| \approx \|Y\|$, (S_2) $\|K\| \approx \|X\| \approx \|Y\|$,
(S_3) $\|K\| \ll \|X\|$ and $\|X\| \approx \|Y\|$

where $\|\circ\|$ is the cardinal number of the set $\circ$ (in our case, simply the number of its elements). In the example (a), a library catalogue is a database containing records indexed by the authors, titles, subjects, etc. The explicit "cue" is the key, using which a set of catalogue items $Y' \subseteq Y$ can be found as a result of a query. Typically, $\|K\| \gg \|X\|$, while $\|Y\| \approx \|X\|$: this is our first scenario (S_1).

Similarly, a book can be indexed by associating its content (e.g., pages) with keywords. In this case, $\|K\| \gg \|X\|$ as well, since there may be more keywords than pages, while $\|Y\| \approx \|X\|$ as the number of retrieved pages may approach their total number. However, the scenario (S_2) pushes the scenario (S_1) to the extreme by restricting the number of possible keys (e.g., a limit on queries), while the memory size is unchanged: $\|K\| \approx \|X\|$. This represents a more challenging case with respect to the precision as the relevant items are harder to find.

The example (b) involves an artificial neural network, e.g., a self-organizing map (SOM) implementing associative memory, briefly discussed in section 2. Each neuron in memory (a network node) encodes a retrievable pattern, hence $\|Y\| \approx \|X\|$. Of course, memory updates would lead to an increase in the overall number of returned patterns, highlighting the distinction between cumulative memory capacity and memory size. The SOM handles multiple cues/keys as partial or corrupted patterns of data, associating them with the memory, implying $\|K\| \gg \|X\|$. This also concurs with the first scenario (S_1). Again, restricting the number of possible keys while keeping the memory size is unchanged (the scenario (S_2)) would challenge the system in terms of the precision.

The third scenario (S_3) may correspond to an auto-associative neural network such as the Hopfield network [10] or a Sparse Distributed Memory [15]. A key is interpreted simultaneously by all neurons which interact by updating their weights until a stable network state is reached: this attractor then represents the network output associated with the key. In this case, $\|X\| \gg \|Y\|$ since there is only a limited number of attractor states supported by the network, while $\|K\| \ll \|X\|$ due to high-dimensionality of memory. Interestingly, restricting the memory (reducing $\|X\|$) would challenge precision again, approaching the scenario (S_2) from another direction.

Finally, we consider the case (c) when a genotype (DNA) is decoded by proteins. An individual DNA can be interpreted as associative memory in the sense that it contains *potential information* relevant to the niche occupied by the individual's species. As pointed out by Adami [1], "If you do not know which system your sequence refers to, then whatever is on it cannot be considered information. Instead, it is potential information (a.k.a. entropy)". Decoding a DNA involves a complicated protein machinery (the key), and may corresponds to an associative recall. In this model, a replicated offspring is an associatively-recalled prototype. In the next section we shall interpret all three scenarios within this analogy.

4 Results

The experimental setup is very simple: we intend to satisfy our criteria (1), (9), and (11) by varying possible deterministic functions $Y = f(K, X)$ over finite size domains K, X and Y, for the scenarios (S_1), (S_2) and (S_3). In particular, we consider three sets of integers $\{1, \ldots, \|K\|\}$, $\{1, \ldots, \|X\|\}$ and $\{1, \ldots, \|Y\|\}$, and vary their sizes $\|K\|$, $\|X\|$ and $\|Y\|$ between experiments. For each experiment, we search for deterministic mappings $Y = f(K, X)$ which maximize $\mathcal{P}$, or $\mathcal{R}$, or $\mathcal{E}$ — repeating the search for each of these criteria. We used a simple

genetic algorithm to evolve solutions to the maximization problems. The initial population is generated by random mappings $Y = f_i(K, X)$, for a sufficiently large number of individual mappings, e.g. $1 \leq i \leq 1000$. At each generation, the mappings are evaluated in terms of the criterion in point (either $\mathcal{P}$, or $\mathcal{R}$, or $\mathcal{E}$). We have chosen a generation gap replacement strategy (the entire old population is sorted according to the fitness, and the best 10% are chosen for direct replication in the next generation, employing an elitist selection mechanism), and the multiple-point crossover. We also ensure that the mutation results in a unique individual by re-applying this operator if necessary.

Visualizing evolved mappings f is not revealing, as can be observed from Figure 1. We plot instead an analogue of a 2-dimensional contour, but rather than simply using contours, we connect, for a given height $y \in Y$, all points $(k, x) \in K \times X$ which agree either on k or on x, producing a partial grid. For example, if there are entries $7 = f(1, 4)$, $7 = f(3, 4)$, and $7 = f(1, 6)$, we connect points $(1, 4)$ and $(3, 4)$ as they represent the same memory $x = 4$, as well as points $(1, 4)$ and $(1, 6)$ sharing the same key $k = 1$. A grid-contour combines grids for all values of $y \in Y$ by "overlaying" the grids for all values y.

A random mapping (the zero hypothesis) has no discernable structure for all scenarios (e.g., Figure 2). Let us focus initially on the scenario (S_1). A $\mathcal{P}$-maximizing mapping for this scenario is a structure with dominant horizontal lines (Figure 3). Each horizontal reflects the fact that in the evolved mapping, the same memory is recalled if multiple different keys are associated with it. This, in the context of DNA decoding, corresponds to conservation of DNA (memory) and its robustness to possible decoding errors (multiple keys), ensuring high precision. A $\mathcal{R}$-maximizing mapping maintains the horizontal lines but introduces some vertical lines (Figure 4). Each vertical line means that a key recalls the same content even if associated with different memories. In the context of DNA decoding, this would correspond to junk DNA: redundant code which does not differentiate between offsprings and ensures high recall. Importantly, the effectiveness criterion $\mathcal{E}$ maintains the horizontal lines (robust DNA) but eliminates the vertical lines (no junk DNA), as shown in Figure 3. On the other hand, minimization of $\mathcal{E}$ does the opposite, producing a grid-like structure, i.e., for every association (k_1, x_1) there exists an association (k_2, x_2) such that either $k_1 = k_2$ or $x_1 = x_2$.

The scenario (S_2) pushes the observed tendencies to their limits. A $\mathcal{P}$-maximizing mapping for this scenario is a structure with no lines (Figure 5). There are no entries which share either a key or memory — in other words, both key and memory are necessary. Such a *holographic* outcome illustrates the full precision of associative memory (a perfectly succinct DNA). An $\mathcal{R}$-maximizing mapping has some vertical lines (Figure 6), suggesting that some junk DNA is possible even in the highest recall case. This can be interpreted as a tendency towards the dominance of precision over recall, i.e., robustness of DNA at the expense of redundancy. However, the effectiveness criterion $\mathcal{E}$ eliminates redundancy and results in the holographic structure (Figure 5). This mapping implements a fully

associative memory, where for every pair of a key and memory, fixing a key k and varying memory x (or vice versa) results in a different outcome $y = f(k, x)$.

The results for the scenario (S_3) are not surprising: mappings maximizing $\mathcal{P}$, $\mathcal{R}$ and $\mathcal{E}$ produce structures with only vertical lines. In the context of DNA decoding, this would correspond to highly redundant and error-prone DNA structures. This model would work for reproduction if different arrays collectively store information (as in an SDM or Hopfield network) "retrieving" offspring as a composite result of data fusion, e.g. genetic cross-over.

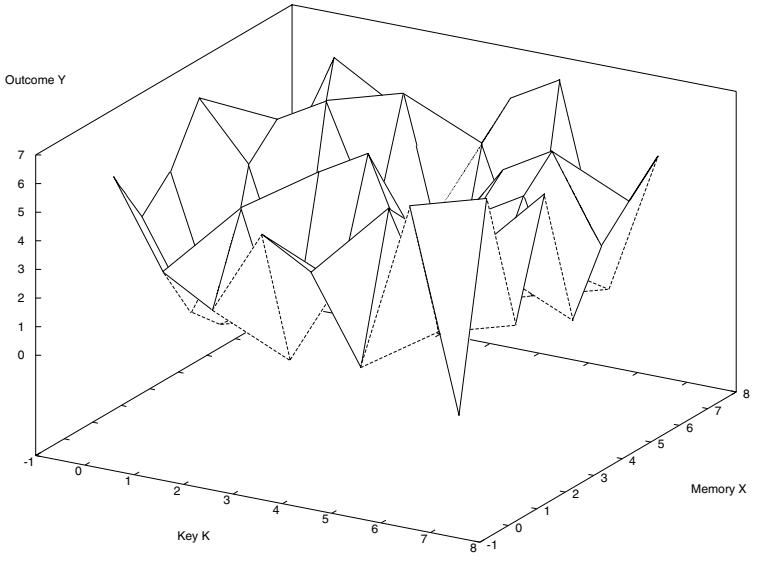


Fig. 1. An evolved mapping: scenario (S_2)

A holographic mapping $Y = f(K, X)$ implementing fully associative memory in the scenario (S_2) (Figure 5) can be interpreted in self-referential terms. Self-referentiality has many interpretations, ranging from programming data structures (a self-referential structure contains a pointer to a structure of the same type) to cognitive neuroscience: the self is a cognitive structure with special mnemonic abilities, leading to "the enhanced memorability of material processed in relation to self" [8,27], suggesting that a self-referential memory — a memory about the self — is not ordinary. According to the well-known interpretation of Hofstadter [9], a self-referential system can be characterised by emergent behaviour and tangled hierarchies exhibiting Strange Loops: "an interaction between levels in which the top level reaches back down towards the bottom level and influences it, while at the same time being itself determined by the bottom level". We shall adopt a weaker interpretation of self-referential memory: the memory using a model of itself. This limited a-model-within-the-model view is not intended to preclude emergence of tangled hierarchies, or references to the cognitive self of the agent using this memory.

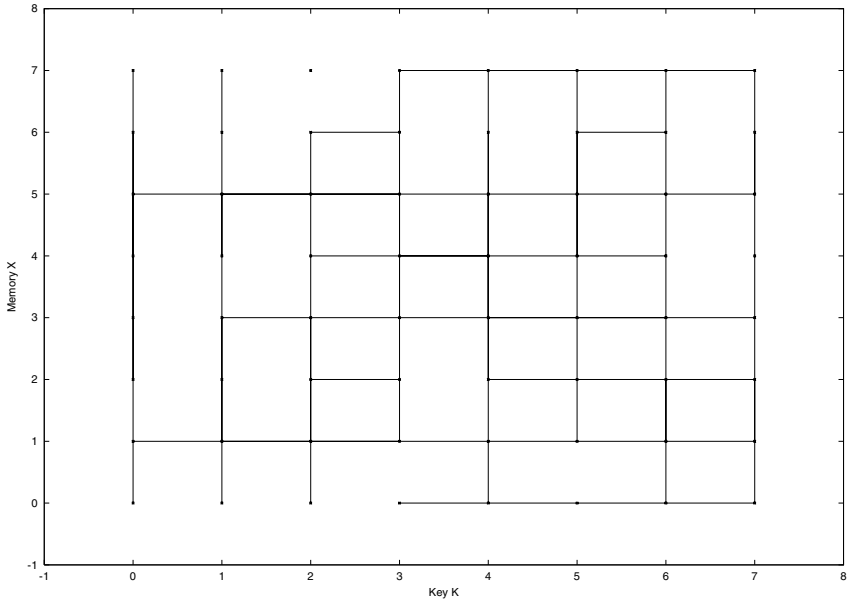


Fig. 2. Grid-contour of a random mapping: scenario (S_2)

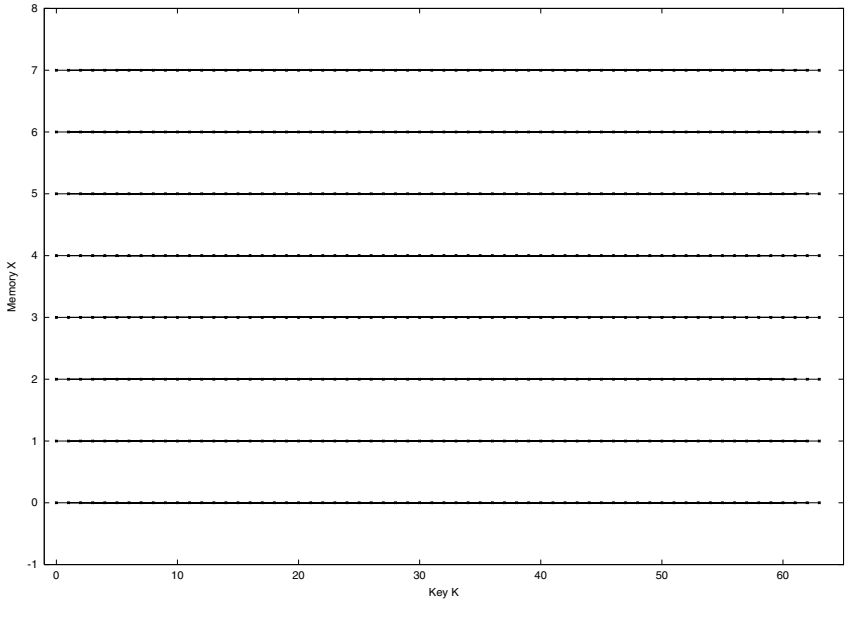


Fig. 3. Grid-contour of a $\mathcal{P}$-maximizing mapping, as well as an $\mathcal{E}$-maximizing mapping: scenario (S_1)

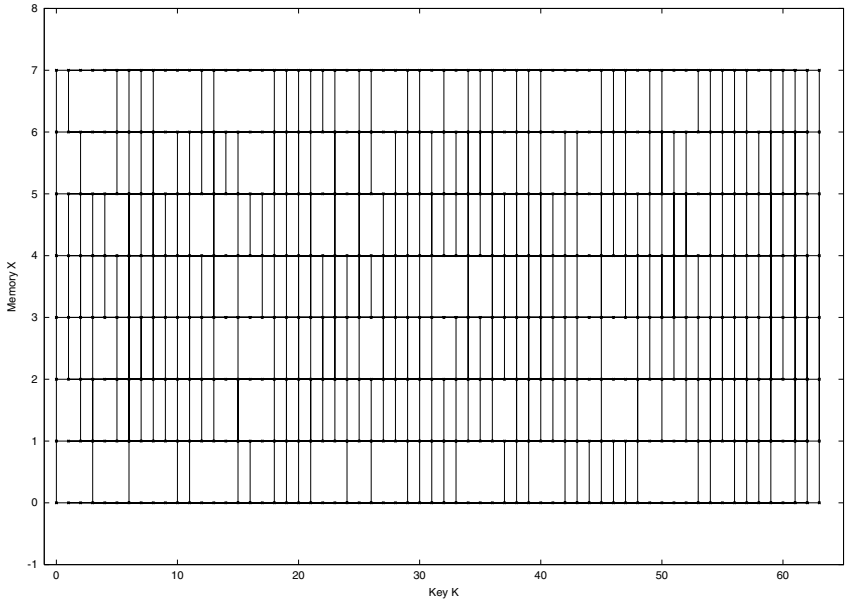


Fig. 4. Grid-contour of a $\mathcal{R}$-maximizing mapping: scenario (S_1)

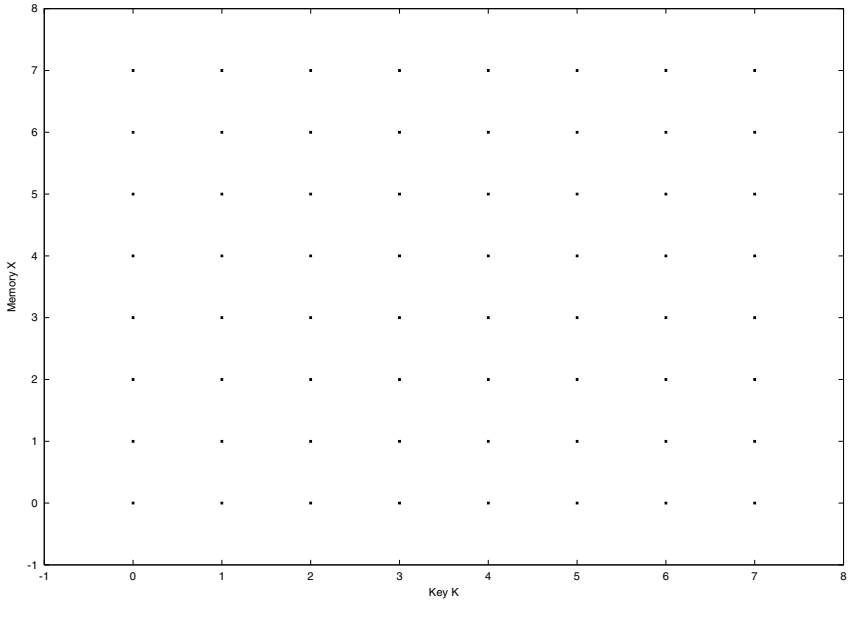


Fig. 5. Grid-contour of a holographic $\mathcal{P}$-maximizing mapping, as well as an $\mathcal{E}$-maximizing mapping: scenario (S_2). Its 3-dimensional counterpart is shown in Figure 1.

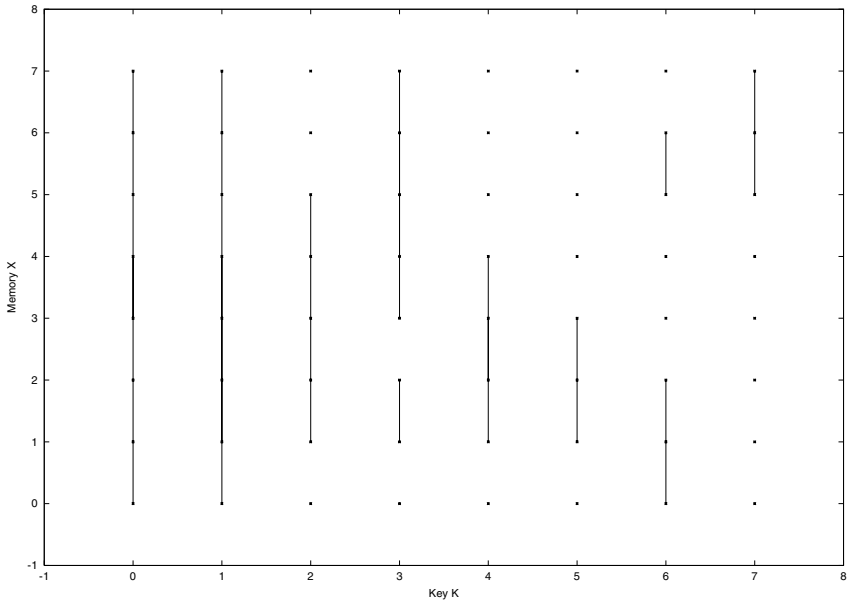


Fig. 6. Grid-contour of a $\mathcal{R}$-maximizing mapping: scenario (S_2)

We begin by observing that, on the one hand, for $y_{1,1} = f(k_1, x_1)$ and $y_{1,2} = f(k_1, x_2)$ (i.e., the same key and varying memory), the outcomes differ: $y_{1,1} \neq y_{1,2}$. This means that every memory is sufficiently sensitive to its own content/location, and there is no redundant information in the associated key: the difference in the recall is due to different memory. On the other hand, for $y_{1,1} = f(k_1, x_1)$ and $y_{2,1} = f(k_2, x_1)$ (i.e., varying key and the same memory), the outcomes differ as well: $y_{1,1} \neq y_{2,1}$. This means that every key is sufficiently informative to produce different outcomes upon association with the same memory. In other words, every memory content is sufficiently sensitive to each key (as well as to its content/location), and therefore, encodes information about all possible keys. If the memory was simply mirroring each associated key, it would not be sensitive to its own content/location. Hence, each memory content *uniquely* encodes information about all possible keys, e.g., in an array of permuted keys. This information *per se* is a memory model. Hence, each memory content maintains a possible model of itself. We believe that it is precisely this self-referentiality that leads to *holographic-ness* and optimizes effectiveness of the considered information transfer in terms of precision and recall. The self-referentiality emerges under the pressures imposed by restricting the number of queries and outcomes to the memory size: the scenario (S_2). If one of these pressures is relaxed, self-referentiality is not needed and a memory does not have to encode information about all possible keys: hence, the presence of horizontal lines in the optimal structures for the scenario (S_1), or vertical lines for the scenario (S_3).

5 Discussion and Conclusions

The evolved mappings $Y = f(K, X)$ maximize precision, recall and effectiveness of the potential information transfer throughout associative memory. Maximization results in holographic structures that can be interpreted in self-referential terms. On the contrary, minimization of the proposed criteria result in full-grid structures which do not require associativity as it does not affect the involved information transfer.

It was conjectured (e.g., [21,24]) that the degree of self-referentiality employed by a self-replicating multi-cellular organism is related to efficiency of its self-inspection and self-repair — and may be quantitatively measured in order to evolve more efficient processes. This conjecture was tested in this work in terms of memory structures and the information transfer. Continuing our analogy with DNA as an associative memory, it is interesting to observe that real-life examples of DNA are not approaching the maximum information transfer, as evidenced by their non-perfect error recovery and significant redundancy. Thus, in terms of self-replication, the maximum potential is not realized — it would require higher precision and higher recall, culminating in a perfectly-associative memory. Interestingly, another extreme, lower precision and/or lower recall, can be pointed out already. We believe that a suitable example is the self-replication mechanism exhibited by mineral crystals in the absence of biological enzymes, as advocated by Cairns-Smith [2]: clay crystals can store information as a pattern of inhomogeneities that are propagated from layer to layer, with few errors; they can reproduce by random fragmentation; and they can express a variety of morphological phenotypes. Following this intuition, Schulman and Winfree recently proposed a method of error-correcting self-replication that works by similar growth and fragmentation of algorithmic DNA crystals [28]: "crystal growth extends the layers and copies the sequence of orientations, which may be considered its genotype. ... splitting of a crystal can yield multiple pieces, each containing at least one copy of the entire genotype". Such self-replication can be considered as non-associative memory recall, where a key is not necessary at all, and neither the point of crystal fragmentation nor surrounding environmental conditions are important. In other words, Cairns-Smith model of crystal self-replication is near the low-precision and low-recall extreme, while a self-referential associative memory would implement the highest-effectiveness case.

Adami advocated the view that "evolution increases the amount of information a population harbors about its niche" [1]. The information-theoretic criteria proposed in this work may further formalize the notion of information transfer involved in self-replication, and enable bio-inspired design of more effective memory structures.

References

1. Adami, C., 2002, What is complexity?, *Bioessays*, 24(12), 1085–1094.
2. Cairns-Smith, A.G., 1966, The origin of life and the nature of the primitive gene, *Journal of Theoretical Biology*, 10, 53–88.

3. Der, R., Steinmetz, U., Pasemann, F., 1999, Homeokinesis — A new principle to back up evolution with learning, *Concurrent Systems Engineering Series*, 55, 43–47.
4. Dorigo, M., Maniezzo, V., Colorni, A., 1996, The Ant System: Optimization by a colony of cooperating agents, *IEEE Transactions on Systems, Man, and Cybernetics, B*, 26(1), 1–13.
5. Dorigo, M., and Di Caro, G., 1999, Ant Colony Optimization: A new meta-heuristic, in: *Proceedings of 1999 Congress on Evolutionary Computation, Washington DC*, 1470–1477, IEEE Press, Piscataway, NJ.
6. Goertzel, B., 1993, *The structure of intelligence: A new mathematical model of mind*, New York: Springer-Verlag.
7. Goutte, C., and Gaussier, E., 2005, A probabilistic interpretation of precision, recall and F-score, with implication for evaluation, in: *Proceedings of the 27th European Conference on Information Retrieval*, Santiago de Compostela, Spain.
8. Heatherton, T.F., Macrae, C.N., Kelley, W.M., 2004, What the social brain sciences can tell us about the self, *Current Directions in Psychological Science*, 13(5), 190–193.
9. Hofstadter, D. R., 1989, *Gödel, Escher, Bach: An eternal golden braid*, New York: Vintage Books.
10. Hopfield, J.J., 1982, Neural networks and physical systems with emergent collective computation abilities, *Proceedings of the National Academy of Science*, 79, 2554–2558.
11. Goldsmith, R.S., and Miller, J.F., 2003, Cooperative co-evolution of robot control, sensor relevance and placement, in: *Proccedings of EPSRC Evolvability and Sensor Evolution Symposium*, Birmingham, U.K.
12. Klyubin, A. S., Polani, D., Nehaniv, C. L., 2004, Organization of the information flow in the perception-action loop of evolved agents, in: *Proceedings of 2004 NASA/DoD Conference on Evolvable Hardware*, IEEE Computer Society, 177–180.
13. Klyubin, A. S., Polani, D., and Nehaniv, C. L., 2005. Empowerment: A universal agent-centric measure of control, in: *Proceedings of The 2005 IEEE Congress on Evolutionary Computation*, IEEE Press, 128–135.
14. Klyubin, A. S., Polani, D., Nehaniv, C. L., 2005, All else being equal be empowered, in: Capcarrère, M.S., Freitas, A.A., Bentley, P.J., Johnson, C.G., Timmis, J. (eds.), *Advances in Artificial Life, 8th European Conference, ECAL, 2005, Canterbury, UK, September 5-9, 2005, Proceedings*, 744–753, Springer, Lecture Notes in Computer Science, Vol. 3630.
15. Kanerva, P., 1988, *Sparse Distributed Memory*, Cambridge, Mass.: MIT Press.
16. Kanerva, P., 1993, Sparse Distributed Memory and related models. In M.H. Hassoun (ed.), *Associative Neural Memories: Theory and Implementation*, New York: Oxford University Press, 50–76.
17. Kohonen, T., 1984, *Self-organization and associative memory*, Berlin, Springer-Verlag.
18. Langton, C., 1991, Computation at the Edge of Chaos: Phase transitions and emergent computation. In S. Forest (ed.), *Emergent Computation*, MIT.
19. MacKay, D. J. C., 2003, *Information theory, inference and learning algorithms*, Cambridge University Press.
20. Moskowitz, J.P., and Jousselin, C., 1989, An algebraic memory model. *ACM SIGARCH Computer Architecture News archive*, 17(1), 55–62, ACM Press, New York, NY, USA.

21. Prokopenko, M., and Wang, P., 2004, On Self-referential shape replication in robust aerospace vehicles, in Pollack, J. (ed.) *Artificial Life IX: Proceedings of The 9th International Conference on the Simulation and Synthesis of Living Systems*, Boston, USA, MIT Press, 27–32.
22. Prokopenko, M., Wang, P., Price, D.C., Valencia, P., Foreman, M., Farmer, A.J., 2005, Self-organizing hierarchies in sensor and communication networks. *Artificial Life*, Special Issue on Dynamic Hierarchies, 11(4), 407–426.
23. Prokopenko, M., Wang, P., Foreman, M., Valencia, P., Price, D.C., Poulton, G. T., 2005, On connectivity of reconfigurable impact networks in Ageless Aerospace Vehicles. *The Journal of Robotics and Autonomous Systems*, 53(1), 36–58.
24. Prokopenko, M., Poulton, G., Price, D. C., Wang, P., Valencia, P., Hoschke, N., Farmer, A. J., Hedley, M., Lewis, C., Scott, D. A., 2006, Self-organising impact sensing networks in robust aerospace vehicles, in Fulcher, J. (ed.) *Advances in Applied Artificial Intelligence*, Idea Group, 186–233.
25. Prokopenko, M., Gerasimov, V., Tanev, I., 2006, Measuring spatiotemporal coordination in a modular robotic system, in: Rocha, L.M., Yaeger, L.S., Bedau, M.A., Floreano, D., Goldstone, R.L., Vespignani, A., (eds.). *Artificial Life X: Proceedings of The 10th International Conference on the Simulation and Synthesis of Living Systems*, Bloomington IN, USA.
26. Prokopenko, M., Gerasimov, V., Tanev, I., 2006, Evolving spatiotemporal coordination in a modular robotic system, in *Proceedings of The 9th International Conference on the Simulation of Adaptive Behavior* (to appear), Rome, Italy.
27. Rogers, T.B., Kuiper, N.A., Kirker, W.S., 1977, Self-reference and the encoding of personal information, *Journal of Personality and Social Psychology*, 35(9), 677–688.
28. Schulman, R., and Winfree, E., 2005, Self-replication and evolution of DNA crystals, in: M. S. Capcarrère, A.A. Freitas, P.J. Bentley, C.G. Johnson, J, Timmis (eds.) *Advances in Artificial Life, 8th European Conference, ECAL 2005, Canterbury, UK, September 5-9, 2005, Proceedings*, 734–743, Springer.
29. Tanev, I., Ray, T., Buller, A., 2005, Automated evolutionary design, robustness, and adaptation of sidewinding locomotion of a simulated snake-like robot, *IEEE Transactions On Robotics*, 21, 632–645.

On the Power of Bio-Turing Machines

H. Ramesh[1], Shankara Narayanan Krishna[2], and Raghavan Rama[1]

[1] Department of Mathematics,
IIT Madras, Chennai, India - 600 036
{ramesh_h, ramar}@iitm.ac.in
[2] Department of Computer Science and Engineering,
IIT Bombay, Powai, Mumbai, India- 400 076
krishnas@cse.iitb.ac.in

Abstract. In this paper, we continue the study of Bio-Turing machines introduced in [1]. It was proved in [1] that using two differentiated cells, and using antiport rules of weight 2, one can recognize the family $1RE$. We show here that with just one differentiated cell, $1RE$ can be characterized, by using antiport rules of weight 2, or by using symport rules of weight 3. We also prove that RE can be characterized using arbitrary alphabets, using 2 differentiated cells, and antiport rules of weight 2. Finally, we examine the computational power when there are no differentiated cells and show that non-regular languages can be accepted.

1 Introduction

Bio-Turing machines were introduced by F.Bernardini et al in [1]. The motivation came from the observation that the notion of a "cell" used in Turing machines is very restricted and local. So it is natural to consider Turing machines where the cells are "real cells", that is, membrane containing multisets of objects, arranged in a linear manner. The objects evolve by means of symport/antiport rules [5] which is well known in membrane computing. In some sense, we have a tissue-like system, but we have an "infinite tape", hence an infinite string of membranes, and no interaction with the environment.

The intuition is that the tape of the machine has first k cells (the tape is finite to the left and infinite to the right) *differentiated*. They have different contents and different rules. All other cells are *non-differentiated*, which have the same contents and rules. An input is provided in the form of symbol objects introduced in first cells of the tape. The string of these symbols is accepted if the computation halts. It was known that these devices are universal. More details and some examples can be found in [1].

This paper is organized as follows: We give some preliminaries in Section 2. In Sections 3 and 4, we recall the definition of bio-Turing machines and investigate the computational power while using (i) a single differentiated cell, (ii) two differentiated cells and arbitrary alphabet, and (iii) no differentiated cells.

C.S. Calude et al. (Eds.): UC 2006, LNCS 4135, pp. 243–253, 2006.

2 Preliminaries

For the basic elements of formal language theory needed, we refer to any monograph in this area, and the details related to membrane computing can be found in [6]. We just list a few notions and notations here.

By REG, CF, CS, RE we denote the families of regular, context-free, context sensitive and recursively enumerable languages respectively. If the languages are over alphabets with at most $n \geq 1$ symbols, then we write $nREG, nCF, nCS$, nRE, respectively.

As in [1], we consider one membrane cells, communicating through symport/antiport rules. For uniformity, we use only antiport rules, of the form $(i, x; y, j)$ where i, j are labels of cells and x, y are multisets of symbol-objects, with the meaning that, the objects indicated by x pass from cell i to cell j and, at the same time, the objects indicated by y pass from cell j to cell i. One of the multisets x, y can be empty, and this corresponds to the case of a symport rule. The maximal length of x or y is called the *weight* of the rule $(i, x; y, j)$.

The proofs about membrane systems in this paper are based on the concept of Minsky's register machine [4]. Such a machine runs a program consisting of numbered instructions of several simple types. Several variants of register machines with different number of registers and different instruction sets were shown to be computationally universal (e.g., see [4]).

An *n-register machine* is a construct $M = (n, H, l_0, l_h, I)$, where:

- n is the number of registers,
- H is the set of instruction labels,
- l_0 is the initial label,
- l_h is the final label, and
- I is a set of labelled instructions of the form $l_i : (op(r), l_j, l_k)$, where $op(r)$ is an operation on register r of M, l_i, l_j, l_k are labels from the set H (which labels the instructions in a one-to-one manner),

The machine is capable of the following instructions:

$(ADD(r), l_j, l_k)$: Add one to the contents of register r and proceed to instruction l_j or to instruction l_k; in the deterministic variants usually considered in the literature we demand $l_j = l_k$.
$(SUB(r), l_j, l_k)$: If register r is not empty, then subtract one from its contents and go to instruction l_j, otherwise proceed to instruction l_k.
halt: Stop the machine. This additional instruction can only be assigned to the final label l_h.

In their *deterministic variant*, such n-register machines can be used to compute any partial recursive function $f : \mathbf{N}^\alpha \to \mathbf{N}^\beta$; starting with $(n_1, \ldots, n_\alpha) \in \mathbf{N}^\alpha$ in registers 1 to α, M has computed $f(n_1, \ldots, n_\alpha) = (r_1, \ldots, r_\beta)$ if it halts in the final label l_h with registers 1 to β containing r_1 to r_β. If the final label cannot be reached, then $f(n_1, \ldots, n_\alpha)$ remains undefined.

A deterministic m-register machine can also analyze an input $(n_1, \ldots, n_\alpha) \in \mathbf{N}_0^\alpha$ in registers 1 to α, which is recognized if the register machine finally stops

by the halt instruction with all its registers being empty. If the machine does not halt, the analysis was not successful. In their *non-deterministic variant*, n-register machines can compute any recursively enumerable set of non-negative integers (or of vectors of non-negative integers). Starting with all registers being empty, we consider a computation of the n-register machine to be successful, if it halts with the result being contained in the first (β) register(s) and with all other registers being empty.

A register machine can also be used for defining a language, in the following way. If $V = \{a_1, \ldots, a_k\}$, then each string $w \in V^*$ can be interpreted as a number in base $k+1$. Specifically, if $w = a_{i_1} a_{i_2} \ldots a_{i_n}$, $1 \leq i_j \leq k, 1 \leq j \leq n$, then $val(w) = i_1(k+1)^{n-1} + \ldots + i_{n-1}(k+1) + i_n$. Then, we have:

Proposition 1. *If $L \subseteq V^*$, $card(V) = k, L \in RE$, then a 3-register machine M exists such that for every $w \in V^*$ we have $w \in L$ if and only if M halts when starting with $val_{k+1}(w)$ in its first register; in the halting step, all registers of the machine are empty.*

3 Bio-Turing Machines

We recall the definition from [1]. A *bio-Turing machine* is a construct

$$\Pi = (O, \$, k, w_1, \ldots, w_k, w, R)$$

where: (i) O is an alphabet, (ii) \$ is a special symbol not in O, used as an end marker, (iii) $k \geq 1$ is the *degree* of the machine, (iv) $w_1, \ldots, w_k$ are strings over O, representing the initial contents of the first k cells, (v) w is the content of cells $k+1, k+2, \ldots$ (the same for all cells from cell $k+1$ to infinity), and (vi) R is a finite set of rules of the following two types: (a) $(i, x; y, i+1)$, for $1 \leq i \leq k$, $x, y \in O^*$, and (b) $(*, x; y, * + 1)$, for $x, y \in O^*$.

The tape of the machine is bounded to the left and infinite to the right. We call the first k cells *differentiated*, they have different contents and rules, while all other cells are *non-differentiated*, they have the same contents and rules. Any rule of the form $(*, x; y * +1)$ can be used to exchange the multisets x, y between cells $j, j+1$ with $j > k$.

The input string $w = a_{i_1} \ldots a_{i_n}$ is introduced in the first n cells and the end marker \$ in cell $n+1$. From the initial configuration the computation proceeds as usual in membrane computing, viz., by using the rules from each cell in a non-deterministic maximally parallel way. If the computation halts, then the input string $w = a_{i_1} \ldots a_{i_n}$ is accepted.

The language of all strings recognized by the machine Π is denoted by $L(\Pi)$. The family of all languages recognized by bio-Turing machines with at most k differentiated cells and using the rules of *weight* at most r is denoted by $LTP_k(anti_r)$. If any of the parameters k, r is not bounded, it is replaced with $*$. When considering languages over an alphabet with at most $n \geq 1$ symbols we denote the corresponding family by $nLTP_k(anti_r)$. It has been proved in [1] that $1RE = 1LTP_k(anti_r), k, r \geq 2$.

4 Universality Results

We first show that universality can be obtained with just one differentiated cell, and antiport rules of weight 2. In the proofs of Theorems 1 and 2, we use a register machine $M = (m, H, l_0, l_h, I)$. Adding a new register to M, which we call register 0, as well as two new instructions $s_0 : (SUB(0), s_1, l_0)$ and $s_1 : (ADD(1), s_0)$, where s_0, s_1 are new labels, we obtain a new register machine M'. Let us consider $M' = (m+1, H \cup \{s_0, s_1\}, s_0, l_h, I')$ and assume that we start with some number n in register 0, while all other registers are empty. Using instructions s_0, s_1, the number n is shifted to register 1. It follows that $N(M) = N(M')$, and so, we can follow the instructions of M, never again looking at register 0.

Theorem 1. *$1RE = 1LTP_k(anti_r)$ for all $k \geq 1, r \geq 2$.*

Proof. Given the register machine M as above, let us construct a bio-Turing machine Π which recognizes a_0^n iff $n \in N(M)$. Take

$$\Pi = (O, \$, 1, ded'ed''e'', g^2z^2w', R)$$

where

$$O = E \cup \{a_0, g, d, e, d', e', d'', e'', z\},$$
$$E = \{a_i \mid 1 \leq i \leq m\} \cup \{l_i, l_i', l_i'', l_i''', L_i \mid 0 \leq i \leq h\}$$

and w' consists of symbols of E once. Thus, the initial configuration looks like :

$a_0ded'e'd''e''$	a_0ggzzw'	a_0ggzzw'	$\ldots$	a_0ggzzw'	$\$ggzzw'$	$ggzzw'$	$\ldots$	$ggzzw'$	$\ldots$

The set of rules are as follows:

1	$(*, \lambda; x, * + 1), x \in \{a_0, \$\}$	2	$(1, \lambda; a_0, 2)$
3	$(1, de; \$, 2)$	4	$(*, d; \lambda, * + 1)$
5	$(*, e; \lambda, * + 1)$	6	$(*, \lambda; d\alpha, * + 1), \alpha \in E$
7	$(*, \lambda; e\alpha, * + 1), \alpha \in E$	8	$(1, d'e'; de, 2)$
9	$(1, d''; d'l_0, 2)$	10	$(1, e''; e'l_0')$
11	$(1, d''; d'g, 2)$	12	$(1, e''; e'g, 2)$
13	$(1, g; z, 2)$	14	$(z, 1; g, 2)$

1. Initialization : Bringing in all copies of a_0 inside cell 1.
   - We plan to simulate the register machine in cell 1. To this end, we first bring all the a_0's into cell 1. This is done by using the rules 1 and 2. This makes all the a_0's move into cell 1.
   - When \$ arrives in cell 1, we know that the entire string has moved into cell 1. After this, we start moving all the necessary symbols required for simulation into cell 2 from the infinitely many cells to the right. This is done by using rule 3. This makes de move out of cell 1. Using d, e we start collecting objects required for simulation. This is accomplished by the rules 4 to 7. These rules allow d, e to move right to any distance, and while coming back, they fetch symbols of E.

- The movement of d, e should be stopped after some time, since otherwise, an infinite computation will be induced. This is done by rule 8. This makes symbols d', e' move out of cell 1 together. The symbols d, e return to cell 1 together.
- Next, we start the simulation of the register machine by bringing in cell 1, objects representing the initial instruction. We use rules 9 and 10. This sends out d'', e'' from cell 1, and brings inside cell 1, the objects l_0, l_0' representing the initial instruction. Note however, that in case the objects l_0, l_0' are not in cell 2, we use the rules 11 and 12, bringing in a copy of g from cell 2 to cell 1. This further induces an infinite computation due to the rules 13 and 14.

2. Simulation an ADD instruction $l_i : (ADD(r), l_j)$. This is handled by the following rules: $(1, l_i; a_r, 2), (1, l_i'; l_j l_j', 2)$. Thus, while l_i brings in the object a_r from cell 2, l_i' brings in l_j as well as a primed version of it. Now, using l_j, l_j', we can simulate instruction j. The only thing that needs to be handled here is the possibility of $a_r, l_j l_j'$ not being in cell 2. This is done by adding the rules $(1, l_i; z, 2), (1, l_i'; z, 2)$, which brings into cell 1, the object z and induces an infinite computation by rules 13, 14.
3. Simulation of a SUB instruction $l_i : (SUB(r), l_j, l_k)$.

1. $(1, l_i'; l_i'', 2)$	
2. $(1, l_i''; 2, L_i)$	
Register 1 non-zero	Register 1 is zero
3. $(1, l_i a_r; l_j''', 2)$ 4. $(1, L_i l_j'''; 2, l_j l_j')$	5. $(1, l_i L_i; l_k l_k', 2)$

To start with, we always use rule 1. This exchanges l_i' in cell 1 with l_i'' of cell 2. If register i was non-zero, then rule 3 will be applicable in parallel with rule 1. This would bring in l_j''' from cell 2 in exchange for $a_r l_i$ of cell 1. Thus, we would have at the end of step 1, l_i'', l_j''' in cell 1. Then rule 2 is used, which brings in cell 1, L_i in exchange for l_i''. Finally, rule 4 is used, bringing in cell 1, $l_j l_j'$. However, if register i was zero, then rule 3 will not be applicable, and hence, rule 4 also will not be applicable. Then the only rule to be used will be rules 1, 2, 5.

4. Halting : Finally, if the register machine halts, we will have l_h in cell 1 and then, there are no more instructions to be simulated. It is clear from the above set of instructions that if Π halts, then the register machine must have entered l_h and conversely, if the register machine halts, then a halting configuration can be arrived at in Π by following the rules correctly, and cell 1 will contain l_h. □

Next, we prove the universality of bio-Turing machines with only symport rules. We obtain the universality with only 1 differentiated cell and symport rules of weight at most 3.

Theorem 2. *$1RE = 1LTP_k(sym_r)$ for all $k \geq 1, r \geq 3$.*

Proof. We prove the inclusion $1RE \subseteq 1LTP_1(sym_3)$. To prove the above inclusion, let us consider an arbitrary register machine $M = (m, H, l_0, l_h, I)$, with contents n in its register 1 and all other registers empty.

Let us construct a bio-Turing machine Π which recognizes strings a_0^n if and only if $n \in N(M)$, which will prove our claim. Let $\Pi = (O, \$, 1, efw_1, dgw, R)$ where:

$$O = E \cup \{a_0, e, f\} \cup \{l^{'}, l^{iv}, l^{v}, l^{viii}, l^{ix}\},$$
$$E = \{a_i \mid 1 \leq i \leq m\} \cup \{l, l^{''}, l^{'''}, l^{vi}, l^{vii} \mid l \in H\} \cup \{d, g\},$$
w_1 contains one occurrence of each $l^{'}, l^{iv}, l^{v}, l^{viii}, l^{ix}$ for each $l \in H$,
w contains all symbols from E exactly once,

The initial configuration is

a_0efw_1	a_0dgw	a_0dgw	$\ldots$	a_0dgw	$\$dgw$	dgw	$\ldots$	dgw	$\ldots$

The set R of rules is constructed as follows.

1	$(*, \lambda; x, * + 1), x_0 \in \{a_0, \$\}$	2	$(1, \lambda; a_0, 2)$
3	$(1, \lambda; d\$, 2)$	4	$(1, de; \lambda, 2)$
5	$(*, e; \lambda, * + 1)$	6	$(*, \lambda; e\alpha, * + 1), \alpha \in E$
7	$(1, \lambda; el_0, 2)$ or $(1, \lambda; eg, 2)$	8	$(1, fg; \lambda, 2)$
9	$(*, f; \lambda, * + 1)$		

Initialization : All copies of a_0 are brought into cell 1 by rules 1,2. We bring \$ in cell 1, by means of rule 3. Next, e enters cell 2, by means of rule 4. We now use the object e in order to bring in cell 2 arbitrarily many objects from the set E, and to this aim, use rules 5, 6. At some moment, we stop this process, by bringing e back to cell 1 together with l_0, the starting label of M, by means of the rule 7. The possibility of some object not present in cell 2 should lead to an infinite computation. This can be done by rules 8, 9.

1. Now we simulate the instructions of M. An ADD instruction l_1 $(ADD(r), l_2)$ is simulated by the following rules: $(1, l_1 l_1'; \lambda, 2), (1, \lambda; l_1' l_2 a_r, 2)$. The object l_1 exits from cell 1 together with the carrier l_1', which comes back together with the correct label l_2 and one copy of a_r. In case l_2 or a_r is not present in cell 2, use $(1, \lambda; l_1' g, 2)$, which will induce an infinite computation.
2. Simulation of a SUB instruction $l_1 : (SUB(r), l_2, l_3)$: We start with rule 1, the object l_1 exits cell 1 together with l_1'. Rule 2 brings the objects $l_1^{''}$ and $l_1^{'''}$ into cell 2. Rules 3, 3' are applied in parallel; $l_1^{''}$ checks whether any a_r is present in cell 1; in the affirmative case it exits cell 1 together with l_1^{iv}, at the same time $l_1^{'''}$ exits cell 1 together with the object l_1^{v}. Next, rules 4, 4' are applied in parallel; l_1^{iv} returns to cell 1 together with l_1^{vi}, l_1^{v} returns with l_1^{vii}. Next, 5" (or 5') is used , by which l_1^{vii} exits cell 1 with $l_1^{''}, l_1^{viii}$ (l_1^{vi} and l_1^{ix}), depending on whether r was empty or not. Finally, 6" or 6' is used, and objects l_1^{viii}, l_1^{ix} return to cell 1 with either l_2 or l_3.

1. $(1, l_1 l_1'; \lambda, 2)$, 2. $(1, \lambda; l_1' l_1'' l_1''', 2)$ or $(1, \lambda; l_1' g, 2)$	
3. $(1, l_1''' l_1^v; \lambda, 2)$, 4. $(1, \lambda; l_1^v l_1^{vii}, 2)$ or $(1, \lambda; l_1^v g, 2)$	
Register 1 non-zero	Register 1 is zero
3'. $(1, l_1'' l_1^{iv} a_r; \lambda, 2)$ 4'. $(1, \lambda; l_1^{iv} l_1^{vi}, 2)$ or $(1, \lambda; l_1^{iv} g, 2)$ 5'. $(1, l_1^{viii} l_1^{vi} l_1^{ix}; \lambda, 2)$ 6'. $(1, \lambda; l_1^{ix} l_2, 2)$ or $(1, \lambda; l_1^{ix} g, 2)$	5". $(1, l_1^{vii} l_1'' l_1^{viii}; \lambda, 2)$ 6". $(1, \lambda; l_1^{viii} l_3, 2)$ or $(1, \lambda; l_1^{viii} g, 2)$

3. Halting : When M halts, then we have l_h in cell 1, and since there are no more instructions to be simulated, Π halts. Conversely, for every halting computation in Π we can find a halting computation in M. □

We now investigate the computational capacity of bio-Turing machines in the arbitrary case. It was conjectured in [1] that one can even recognize languages over arbitrary alphabets. We settle this in an affirmative way here by proving the following theorem.

Theorem 3. *$RE = LTP_k(anti_r)$ for all $k \geq 2$ and $r \geq 2$.*

Proof. Consider a language $L \subseteq V^*$, for some $V = \{b_1, b_2, \ldots, b_k\}$, which is accepted by a register machine $M = (n, H, q_s, q_h, I)$.

We will construct a bio-Turing machine Π which recognizes strings $w \in V^*, w \in L$ if and only if M halts when started with $val_{k+1}(w)$ in its first register; in the halting step, all registers of the machine M are empty. This will imply any recursively enumerable language over arbitrary alphabets can be recognized in this way.

$$\Pi = (O, \$, 2, w_1, w_2, zw, R)$$

where:

$$\begin{aligned}
O &= E \cup \{e, f, g, z, c_1, c_2, \ldots, c_k, b_1, b_2, \ldots, b_k\},\\
E &= \{a_i \mid 1 \leq i \leq n+2\} \cup \{l, \bar{l}, l', l'', l''', l^{iv} \mid l \in H\}\\
&\quad \cup \{q_{s,j}, q_{s,j,t} \mid 1 \leq i \leq k, 1 \leq t \leq j-1\} \cup \{c, q_{s,any}, q_{move}\},\\
w_1 &= z,\\
w_2 &= efggzc_1c_2 \ldots c_k,\\
w &\quad \text{contains all symbols from } E \text{ exactly once,}
\end{aligned}$$

and the set R of rules is constructed as follows.

Idea of construction is the following: Starting in the initial configuration, we bring arbitrarily many copies of E in cell 2; at some moment, we stop this and we introduce c in cell 1, triggering in this way the computation of $val_{k+1}(w)$. Whenever a symbol b_j is introduced in cell 1, j copies of a_{n+1} are also introduced. Assume that we have some m copies of a_{n+2} present in cell 1. We multiply this number m with $k+1$, passing from the m copies of a_{n+2} to km copies of a_{n+1}. In this way, we have $km + j$ copies of a_{n+1}, which corresponds to the value

in base $k+1$ of the string we introduce in the system. After completing this, we change all objects a_{n+1} with objects a_{n+2} and we continue. The presence of the state-object q_{move} in cell 1 will trigger the register machine M_{move} which changes all symbols a_{n+2} with a_1, and ends in the label q_s(the initial label of M), which will start the computation of M.

The contents of each register j of M will be represented by the number of copies of a_j present in cell 1. The computation in Π will halt only if the computation in M halts, and conversely, for every halting computation in M we can find a halting computation in Π.

1. We use the object e in order to bring in cell 2 arbitrarily many objects from the set E, and to this aim we use the rules

$$\begin{array}{ll} (2, ef; \lambda, 3), & (1, \lambda; eg, 2), (1, g; g, 2), \\ (*, e; \lambda, * + 1), & \\ (*, \lambda; e\alpha, * + 1), (2, \lambda; \alpha, 3), & \text{for all } \alpha \in E. \end{array}$$

   The object f remains in cell 3, while e goes to the right at any distance, comes back with any symbol α, and the process is repeated an arbitrary number of time. All objects α brought to cell 3 are sent immediately to cell 2.
2. At some moment, we stop this process, by bringing e back to cell 2, by means of the rule

$$(2, \lambda; e, 3).$$

3. We now use the object e in order to bring c in cell 1, and to this aim we use the following rules

$$(1, \lambda; ce, 2), \quad (1, \lambda; eg, 2), \quad (1, g; g, 2).$$

   From now on, no rule from the previous groups can be used, because the necessary pairs of objects are not in the right places.
4. The introduction of b_j and j copies of a_{n+1} is done by the following rules

$$\begin{array}{lll} (1, cb_j; c_j, 2), & & \\ (1, c_j; b_j q_{s,j}, 2), & (1, c_j; g, 2), & \\ (1, q_{s,j}; q_{s,j,1} a_{n+1}, 2), & (1, q_{s,j}; g, 2) & \text{for } 1 \leq j \leq k, \\ (1, q_{s,j,t}; q_{s,j,t+1} a_{n+1}, 2), & (1, q_{s,j,t}; g, 2), & \text{for } 1 \leq j \leq k \\ & & \text{and } 1 \leq t \leq j-1, \\ (1, q_{s,any}; b_j q_{s,j}, 2), & (1, q_{s,any}; g, 2), & \text{for } 1 \leq j \leq k. \end{array}$$

   When $q_{s,j,j}$ is introduced in cell 1, we pass to the multiplication by $k+1$ of the available copies of a_{n+2}. This can be done by a register machine $M_j = (2, I_j, q_{s,j,j}, q_{s,any})$, with the following instructions:

$$\begin{array}{rl} q_{s,j,j} : & (S(n+2), q_{j,1}, q_j^{'}), \\ q_{j,t} : & (A(n+1), q_{j,t+1}, q_{j,t+1}), \text{ for all } 1 \leq t \leq k, \\ q_{j,k+1} : & (A(n+1), q_{s,j,j}, q_{s,j,j}), \\ q_j^{'} : & (S(n+1), q_j^{''}, q_{s,any}), \\ q_j^{''} : & (A(n+2), q_j^{'}, q_j^{'}). \end{array}$$

The two registers of M_j were numbered with $n+1, n+2$ because M_j will be integrated in a large machine, with $n+2$ registers.

5. In order to move b_j's and \$ towards cell 2, we use the following rules:

$$\begin{array}{lll} (2,\lambda;b_j,3), & (*,\lambda;b_j,*+1), & (2,\lambda;\$,3), \qquad\quad (*,\lambda;\$,*+1) \\ (1,z;b_ib_j,2), & (2,z;b_ib_j,3), & (*,z;b_ib_j,*+1), \text{ for } 1\leq i,j\leq k \\ (1,z;\$b_j,2), & (2,z;\$b_j,3), & (*,z;\$b_j,*+1), \text{ for } 1\leq i,j\leq k \\ (2,zz;\lambda,3), & (*,zz;\lambda,*+1). & \end{array}$$

The idea is to use the object z to sense the appearance of any b_ib_j in the adjacent cell. If it founds any pair, then the object z moves to the next cell from where it moves to right forever and the system never halts. This is to ensure the order of the symbols in the input string.

6. We bring \$ in cell 1, by means of the rule

$$(1,\lambda;\$q_{move},2).$$

This will trigger the register machine M_{move}(whose initial label is q_{move}), which changes all symbols a_{n+2} with a_1, and ends in the label q_s(which is the initial label of M).

7. We simulate the register machine, by means of the following rules. An ADD instruction $l_1 : (ADD(r), l_2) \in R$ is simulated by using the following rules:

$$(1,l_1;l_2a_r,2), \qquad (1,l_1;g,2).$$

The label l_1 in cell 1 is exchanged with l_2a_r. If cell 2 does not contain the necessary objects l_2, a_r, then we have to use the rule $(1,l_1;g,2)$ and the computation will never halt.

8. A SUB instruction $l_1 : (SUB(r), l_2, l_3) \in R$ is simulated by means of the following rules:

$$\begin{array}{ll} (1,l_1;\bar{l}_1l_1^{'''},2), & (1,l_1;g,2), \\ (1,\bar{l}_1l_1^{'''};l_1^{'}l_1^{''},2), & (1,\bar{l}_1l_1^{'''};g,2), \\ (1,l_1^{'}a_r;l_1^{'''},2), & \\ (1,l_1^{''};l_1^{iv},2), & (1,l_1^{''};g,2), \\ (1,l_1^{iv}l_1^{'''};l_2,2), & (1,l_1^{iv}l_1^{'''};g,2), \\ (1,l_1^{iv}l_1^{'};l_3,2), & (1,l_1^{iv}l_1^{'};g,2). \end{array}$$

The first two pairs of rules are meant to ensure that $l_1^{'''}$ is present in cell 2; if this is not the case, then the trap object g comes to cell 1 and the computation will not halt. After making sure that $l_1^{'''}$ is present in cell 2 and if at least one copy of a_r is in cell 1, then we use the rule $(1,l_1^{'}a_r;l_1^{'''},2)$, and at the same time $l_1^{''}$ is exchanged with l_1^{iv}. If l_1^{iv} finds $l_1^{'''}$ in cell 1, then we will bring l_2 in cell 1, otherwise the object l_3 is brought to cell 1. Note that all the above rules except the third one have companion rule, hence they have to be used, otherwise the computation never stops. This is to make sure that if the computation in Π halts, then it is not due to the shortage of necessary objects in cell 2. If the computation in M stops, then also the computation in Π stops. Conversely for every halting computation in M we can find a halting computation in Π, which proves our result. □

Having seen some universality results using one and two differentiated cells, we examine the power of machines with no differentiated cells. All cells have the same w, and there is some input written on the first n cells, $n \geq 0$. We now show that even these systems are capable of recognizing complex languages, which is rather surprising.

Theorem 4. $3LTP_0(anti_2) - REG \neq \emptyset$.

Proof. Construct the machine $\Pi = (O, \$, 0, w, R)$ with $O = \{a, b, d, g\}, w = \{g\}$ and rules R as follows:

1. $(*, d; a, * + 1)$
2. $(*, b; \$, * + 1)$
3. $(*, dg; b, * + 1)$
4. $(*, dg; d, * + 1)$
5. $(*, a; \$g, * + 1)$
6. $(*, gg; \lambda, * + 1)$

It can be seen that $L(\Pi) = b^* \cup \{wda^n b^n \mid w \in \{a, b\}^*, n \geq 0\} \cup \{wda^n \alpha b^n \mid \alpha \neq d, w \in \{a, b\}^*, n \geq 0\}$.

The working details of the machine are as follows: From the rules, it can be seen that the symbol d moves forward by exchanging an input symbol a, and the end marker \$ moves backward by exchanging an input symbol b. An infinite computation is induced if \$ comes next to symbol a, or if d comes before a b. Under these conditions, two symbols g are put in the same cell, which produces an infinite computation by rule 6. Thus, it is clear that g cannot be part of the input. The input clearly has to be over $\{d, a, b\}$. We cannot have two d's in the input, since this would again induce an infinite computation, by rule 4.

Since the symbol d moves only forward (till it comes before \$) and \$ backward (till it comes next to d), we can consider strings of the kind $wdw'\$$, where $w \in \{a, b\}^*$. The claim now is that w' is either of the form $a^n b^n, n \geq 0$ or $a^n \alpha b^n, \alpha \neq d, n \geq 0$. Clearly, if $w' = a^i b^j, i \neq j, j + 1$, then either \$ will come across an a or d will come across a b, giving an infinite computation. Thus, the possible forms for w' are $a^n b^n, a^{n+1} b^n, a^n b^{n+1}, n \geq 0$. Further, we can have a string over b^* as the input, since this would make \$ move backward till the first b, and then stop. It is not possible to have a's in the input without d, since the \$ would come next to some a at some point of time, and induce an infinite loop. Thus, clearly, $L(\Pi) = b^* \cup \{wda^n b^n \mid w \in \{a, b\}^*, n \geq 0\} \cup \{wda^n \alpha b^n \mid \alpha \neq d, w \in \{a, b\}^*, n \geq 0\}$, which is not regular. □

5 Conclusion

We have investigated the power of bio-Turing machines using zero, one and two differentiated cells respectively. The universality results with only one differentiated cell are interesting, and we conjecture that the same results will hold good for any arbitrary alphabet. Further, we believe that even in the one alphabet case, as far as universality goes, Theorems 1,2 are optimal with respect to the

number of differentiated cells used. It is of definite interest to explore whether the weight of the rules used can be reduced. Further, more examples of complex languages using zero differentiated cells needs to be worked out to understand the power of the model.

References

1. F. Bernardini, M. Gheorghe, N. Krasnogor, Gh. Păun, *Turing Machines with Cells on the Tape*, Proc. of the ESF Exploratory Workshop on Cellular Computing (Complexity Aspects), Sevilla (Spain), Jan. 31 - Feb. 2, 2005, 63–74.
2. R. Freund, Gh. Păun, On deterministic P systems. Submitted, 2004.
3. R. Freund, C. Martin-Vide, A. Obtulowicz, Gh. Păun, On three classes of automata-like P systems, Proc. DLT2003, LNCS 2710, Springer, Berlin, 2003, 292-303.
4. M.L. Minsky, *Computation: Finite and Infinite Machines*. Prentice Hall, Englewood Cliffs, New Jersey, USA 1967.
5. A. Păun, Gh. Păun, The Power of Communication: P Systems with Symport/Antiport, New Generation Computers, 20, 3 (2002), 295–306
6. Gh. Păun, *Membrane Computing: An Introduction*, Springer-Verlag, Berlin, 2002.

Ergodic Dynamics for Large-Scale Distributed Robot Systems

Dylan A. Shell and Maja J. Matarić

Department of Computer Science
University of Southern California
Los Angeles, CA USA 90089-0781
{shell, mataric}@usc.edu

Abstract. Intelligent autonomous robotics is a promising area with many potential applications that could benefit from non-traditional models of computation. Information processing systems interfaced with the real world must deal with a continuous and uncertain environment, and must cope with interactions across a range of time-scales. Robotics problems resist existing tools and, consequently, new perspectives are needed to address these challenges. Toward that end, we describe a dynamics-based model for computing in large-scale distributed robot systems. The proposed method employs a compositional approach, constructing robot controllers from ergodic processes. We describe application of the method to two multi-robot tasks: decentralised task allocation, and collective strategy selection.

1 Introduction

Intelligent autonomous robotics is the study of autonomous agents coupled with the physical world. Robots are equipped with sensors to perceive their environment and actuators that allow aspects of the agent, and often parts of the environment, to be controlled. Even robots with limited information processing capabilities can exhibit complex, nontrivial behaviour due to the feedback introduced between the robot and its environment. Systems like autonomous robots, that interface with the world, introduce several unique challenges. We highlight those challenges that may be best addressed by non-traditional computing ideas, illustrating that robotics is a potentially rich application area for broad, alternative notions of computation.

This paper also considers the specific problem of producing predictable task-oriented collective behaviours in systems of many simple robots, or so-called *swarm* robot systems. The word "swarm" is a reference to those natural systems which inspire the research. Several biological systems are capable of operating in distributed and decentralised ways, exploiting synergism of simple individuals, and achieving robustness through massive redundancy. We wish to construct artificial systems with similar features, which is the subject of ongoing research.

We describe a method for programming swarm robot systems applicable to a range of tasks domains. The method explores a dynamics-based formalism primarily using the composition of elementary processes, each possessing the ergodic

C.S. Calude et al. (Eds.): UC 2006, LNCS 4135, pp. 254–266, 2006.

property. Statistical mechanics techniques allow a link to be established between individual (microscopic) interactions and the group (macroscopic) behaviour. The result is an approach for structuring collective interactions in loosely-coupled distributed systems, which emphasises equilibrium states rather than the steps necessary to achieve particular operations.

In order to demonstrate our method, we consider two complementary variations on an entomologically-inspired multi-robot foraging task. We show that our method is sufficient to enable group coordination in both tasks. This also serves to demonstrate the types of capabilities of interest within the multi-robot community. We present data from large-scale simulations.

2 Robotics and Its Challenges

Researchers from several disciplines have shown an interest in synthesising, analysing and studying autonomous system behaviour. The cybernetics community, with its basis in control theoretic ideas, produced early pioneering systems using analogue electronic methods (e.g., [29]). Later, some within the AI community assumed the goal of constructing artificial beings capable of exhibiting intelligent behaviour. [4] recently produced a comprehensive history and discussion of the field.

2.1 Intelligent Robotics

For several years the AI community attempted to apply search and planning techniques to robotics problems. In the late 1980's arguments against the traditional AI methods were put forward by Brooks and his collaborators (see for example [8]). The arguments highlighted assumptions which ignored key aspects of robot systems, such as the implicit belief in the physical symbol hypothesis [23], and an assumption that symbolic search could become tractable.

Robots inhabit a continuous world. Traditional planning methods however, require discretization. Any robot relying on a discrete representation for its successful functioning may become brittle, with small errors causing the robot's beliefs about the world to diverge from reality. The result is execution of inappropriate actions. Some viewed this as a fundamental shortcoming of symbolic representations. Subsequent researchers explored alternatives, like connectionist methods [22], while others have introduced randomised algorithms for planning over continuous spaces [18], or highly unorthodox methods [1]. Currently, probabilistic techniques for explicitly representing and reasoning about the world are the most popular [25]. However, the world is generally a dynamic place and few explicit representations are able to provide machinery for updating models over time.

The physical world is fraught with uncertainty. In addition to the noisy sensors and inaccurate actuators, no robot can reasonably expect to have complete information about the environment in which it finds itself. Uncertainty presents a challenge to the robot as well as the researcher intending to model the world. Many of the methods for designing industrial robots are inappropriate when

robots are expected to operate in unstructured environments. For example, an assumption of known friction constants becomes tenuous.

Robotic systems operate across disparate time-scales. The physical dynamics, hardware interfaces, and high-level computation typically each execute with frequencies that are orders of magnitudes apart. The task specification itself may have non-trivial timing requirements. In all but the most benign environments, robots will have real-time constraints in order to ensure survival and maintain safety through obstacle avoidance. Often the robot has multiple high-level goals (typically several will be time-extended) that must be reconciled with the lower-level constraints.

The choice of computing model dictates the way one approaches any particular problem. [7] has argued that a dominant computing model has effected thinking about intelligence itself. A re-examination in this context has changed the perception of robots and their role in robot-environment interactions. However, beyond a few biological inspirations and proposals for novel controller architectures, a comprehensive re-examination of alternative computing paradigms is yet to come to robotics.

2.2 Multi-robot Systems

There are numerous task domains in which multiple robots operating concurrently offer advantages over a single robot. A task that is impossible for an individual may be feasible for a group. Also, additional robots may improve performance—although typically only up to a point. After that, additional robots aggravate resource conflicts, thus making sophisticated allocation and management policies necessary. Unlike traditional distributed systems, communication cannot always be assumed. Explicit wireless communication may be ephemeral. Further, the notion of implicit communication, through a shared environment, rather than through a dedicated communication channel, is particularly relevant in multi-robot systems.

Swarm systems follow biological inspirations. They are usually homogeneous groups of simple robots, which are individually *minimalist* [10] in that they are an attempt to find the smallest set of capabilities necessary to achieve a particular task. Frequently, implicit communication is the only form of interaction among the robots. Examples of implicitly coordinated systems include instances of puck clustering and sorting. Simulations have been used to hypothesise about the sensing and computational requirements on individual ants [9]; experiments with real robots showed that fewer sensing capabilities are required from individuals because physical dynamics can aid the clustering process [3]. Further robot experiments showed that overall performance depends critically on several physical features of the experiment and the robots themselves [14]. Minimalist cooperative box-pushing [17], again inspired by social ants, has been demonstrated on physical robots. In those results, positive feedback has lead to self-organised task-achievement [6] of the group as a whole. Specific mechanisms from nature have also been generalised, such as the use of pheromone trails, which have been employed in robotics [21,26].

In stark contrast to swarms are multi-robot systems employing explicit coordination methods [16]. These systems use explicit communicative acts generated by a distributed algorithm executing as a layer within an otherwise standard single-robot controller. Such designs may target heterogeneous robot systems, often with fewer robots (≤ 10) than contained in swarms. The physical properties of the robots are not directly considered, whereas in swarm systems, robot physics may be critical for task-achievement. Programming explicitly coordinated systems is algorithmic, while swarms are often better treated as dynamic systems.

When considering a large swarm of robots, in addition to distinct time-scales, the system can be characterised at multiple spatial scales. Local microscopic interactions occur between individual robots. Since the individual robots act autonomously, programming must ultimately be grounded at the microscopic scale. Descriptions of the properties of the complete system (or large parts, relative to the radius of communication) are considered the macroscopic level-of-detail. Collective properties are exposed at this level by considering average system behaviour. Structured local dynamics can produce complex global phenomena. Simulated distributed systems of simple interacting agents have exhibited global behaviour ranging from point and periodic attractors to chaos [13] and, of course, there are classic examples of those capable of universal computation [5].

Challenges in dealing with robot systems, and other systems that interface computation with the physical world, have a large number of potential applications. Visions of a future with ubiquitous computing [28] are far more likely to be realised if we have an appropriate computational model. We believe that such a model would tolerate random failures, allow multiple levels-of-description, and be capable of dealing with approximate and incomplete information. The key question is whether such a model exists, and if so, whether the model is sufficiently expressive. A model which satisfies the requirements, but which sacrifices the ability to perform task-oriented computation, is of little use.

3 Large-Scale Multi-robot Systems

Next, we consider the problem of prescribing a control and communication policy for homogeneous large-scale systems with hundreds of robots, i.e., robot swarms. Such systems are already conceivable within a research setting, but several engineering issues must be overcome in order for such systems to become common. Hundreds of robots would be demanding for most existing methods, as few researchers evaluate scaling properties for more than twenty robots. It is at these numbers of robots for which the method we propose here begins to become feasible.

The method achieves distributed computation through the interactions of coupled processes with the ergodic property. This places a condition on the temporal structure of the process dynamics so that all accessible regions of the processes phase-space can be explored (see pg. 259 for the formal definition). The ergodic property permits programming to occur at an elevated

level-of-description. The underlying philosophy is that microscopic details are not always necessary for controller construction.

3.1 Definitions

We take a *process* P to be a tuple comprising a state-space definition S and a dynamics function Φ, writing $P = \langle S, \Phi \rangle$. The set S need not be finite (nor countable), and represents all possible states a process could occur in. Also Φ is a generally non-deterministic dynamics that produces the process trajectory $s(t) \in S$ for all non-negative $t \in \mathbb{Z}^+$. Suppose such a trajectory is produced by each of the n robots executing process P. A global snap-shot is given by the state vector $\boldsymbol{s}(t) = [s_1(t), s_2(t), \cdots, s_n(t)]$. The *microstate* vector $\boldsymbol{s}(t)$ captures the complete microscopic details of process P across all of the n robots for time t. The system-wide evolution can be interpreted in terms of vectorised dynamics function $\boldsymbol{\Phi}$ constructed by n copies of Φ operating on each component in $\boldsymbol{s}(t)$ to yield a $\boldsymbol{s}(t + \delta t)$.

We consider a restriction to the general $\boldsymbol{\Phi} : \boldsymbol{S} \to \boldsymbol{S}$ so that the i^{th} element may depend only on nearby robots, that is, robots within a given communication disk. Each robot requires only local information in order to update it's internal state for each process, and so all the dynamics functions we consider will have this form.

An arbitrary function $\mathbf{G} : \boldsymbol{S} \to \mathbb{R}$ produces an associated equivalence relation $\boldsymbol{s} \underset{\mathbf{G}}{\sim} \boldsymbol{r} \iff \mathbf{G}(\boldsymbol{s}) = \mathbf{G}(\boldsymbol{r})$ and hence partitions $\boldsymbol{S}$ into equi-$\mathbf{G}$-valued equivalence classes. Each class is called a *macrostate*. Any suitable $\mathbf{G}$ gives a macroscopic view of the system in which any two microstates which are equivalent with respect to $\underset{\mathbf{G}}{\sim}$ are identified. A convenient interpretation is that any two such states appear indistinguishable to an observer capable only of measuring $\mathbf{G}$. One question frequently asked is the relative number of states with $\mathbf{G}(\cdot) = C_1$ and $\mathbf{G}(\cdot) = C_2$. The set of microstates can be compared through the *entropy*, calculated–following Boltzmann–as a log function of the set cardinality $\mathcal{S}(X) \triangleq \ln ||X||$.

3.2 Ergodicity

In a many degree-of-freedom dynamic system two factors direct the overall system behaviour. First, the phase-space $\boldsymbol{S}$, that is, the conceivable states available to the system. Second, the spatio-temporal dynamics $\boldsymbol{\Phi}$ through which the system explores those states. In designing a system expected to perform information processing, the vast majority of approaches (and thinking) is directed toward constructing sophisticated spatio-temporal dynamics. As already mentioned, even simple interaction rules can produce complex collective dynamics. These complexities make the selection of rules in order perform a given task extremely difficult. Prediction of the system may require exact initial conditions, perfect models, or may even be undecidable.

We propose that the dynamics $\boldsymbol{\Phi}$ be chosen so as to enable prediction. Task-oriented computation is performed through changes to the structure of $\boldsymbol{S}$. Next, we define what is meant by "enable prediction."

The average macrostate $\mathbf{G}$ likely to be externally observed from time 0 to T is given by

$$\overline{\mathbf{G}}_T \triangleq \frac{1}{T} \int_0^T \mathbf{G}\left(\boldsymbol{\Phi}^t\left(\boldsymbol{s}\left(0\right)\right)\right)\, dt, \tag{1}$$

where $\boldsymbol{\Phi}^t\left(\boldsymbol{s}\left(0\right)\right) = \overbrace{\boldsymbol{\Phi}\left(\boldsymbol{\Phi}\left(\cdots \boldsymbol{\Phi}\left(\boldsymbol{s}\left(0\right)\right)\right)\right)}^{t \text{ times}}$ generates a single trajectory from initial condition $\boldsymbol{s}\left(0\right)$.

Dynamics possessing the ergodicity property induce a probability measure on the phase-space. More formally a system that exhibits *ergodic dynamics* has the properties that:

1. A measure $\mathbf{P} : \boldsymbol{S} \rightarrow (0, 1]$ exists with $\int_{\boldsymbol{s} \in \boldsymbol{S}} \mathbf{P}(\boldsymbol{s})\, d\boldsymbol{s} = 1$
2. With any initial conditions the system will evolve exploring $\boldsymbol{S}$ entirely given sufficient time. The probability of finding the system in states $\boldsymbol{B} \subseteq \boldsymbol{S}$ is given by the probability mass of $\mathbf{P}$ that lies within the sub-volume $\boldsymbol{B}$ of the phase-space. That is $\Pr\left(\boldsymbol{s}\left(t\right) \in \boldsymbol{B}\right) = \int_{\boldsymbol{r} \in \boldsymbol{B}} \mathbf{P}(\boldsymbol{r})\, d\boldsymbol{r}$.

The key here is that the current microstate places very "loose" restrictions on future states, the dynamics is free to explore the state space. Long-term history and initial conditions are not important in predicting the system's trajectory. Dynamics functions typically produce short-term temporal regularity. In such cases, analysis must consider durations of sufficient length.

Measure $\mathbf{P}$ allows for the phase-space average of $\mathbf{G}$ defined as

$$\langle \mathbf{G} \rangle \triangleq \int_{\boldsymbol{s} \in \boldsymbol{S}} \mathbf{P}\left(\boldsymbol{s}\right) \mathbf{G}\left(\boldsymbol{s}\right)\, d\boldsymbol{s}. \tag{2}$$

For ergodic processes long time-averages equal phase-space averages,

$$\lim_{T \rightarrow \infty} \overline{\mathbf{G}}_T = \langle \mathbf{G} \rangle. \tag{3}$$

In other words the macroscopic behaviour of such processes can be described by modelling the phase-space rather than resorting to simulation of the dynamics. The value of the two means identified in Eqn. 3 is the called equilibrium value of $\mathbf{G}$.

3.3 Coupling Processes

The definitions thus far describe a method for predicting the mean macrostate of a process executing on an homogeneous system of robots. They also show the probabilistic nature of the law of increasing entropy. But in order to perform any useful processing, multiple connected processes must be considered. First, we must generalise the notion of a process by permitting parametrisation. Let $P(m) = \langle S(m), \Phi(m) \rangle$ denote one such process in which the phase-space and

dynamics are variable and each depend on m. As already mentioned, it is the phase-space which we are most interested in.

Suppose the robots are executing two parametrised ergodic processes: $P^0(m) = \langle S^0(m), \Phi^0 \rangle$, $P^1(n) = \langle S^1(n), \Phi^1 \rangle$. Two state vectors $\boldsymbol{s}^1(t)$ and $\boldsymbol{s}^2(t)$, one for each of the processes, gives the complete state of the system. Eqn. 2 can be applied to each independent parametrisation (e.g., $n = n_0, m = m_0$) to calculate expected behaviour.

The parametrised processes are coupled by defining a relation on the parameters. The previous processes can be constrained so that $m + n = C$. A known C implies that a single degree-of-freedom describes the parametrisation. We call this a *macroscopic degree-of-freedom.* A dynamics can be defined that operates on parameters m and n that respects the conservation constraint. If this coupling dynamics is slow compared with P^0 and P^1 then the two processes can be suitably modelled as a single composite process. This composite process can be predicted as before, provided behaviour is analysed on long time-scales compared to the coupling dynamics. The constraint relation structures a composite phase-space from P^0 and P^1.

Clearly this procedure be applied repeatedly and recursively, while time-scales remain suitable.

4 Example Problem Domains

Foraging is a canonical and one of the most widely studied tasks in distributed robotics [2,12,20]. It is entomologically-inspired and requires robots to locate items (called pucks) scattered throughout an environment, and transport them back to a central location (called the home region).

4.1 Domain 1: Task Allocation

We consider a variation of foraging in which two varieties of puck exist, call them type A and type B. Like [15], we consider the case in which each robot may forage only one variety of puck at a time. The task allocation problem involves switching robots from one variety to another in order to emulate the fractional distribution of pucks within the environment. We measure the effectiveness of the allocation by comparing the distribution of pucks within the environment with the proportion of the robots foraging each type.

The robot controllers include traditional behaviours like obstacle avoidance, basic searching, homing, etc. Two ergodic processes are layered above these behaviours in order to provide each robot with a local strategy for choosing the variety of puck to forage.

For n robots, define $P_a(n, m_a) = \langle \{0, \cdots, m_a\}, \Phi_a \rangle$ where

$$\Phi_a \left[s_k(t), s^{l \in A_k}(t) \right] = s_k(t) - \sum_{i \in A_k} \omega_{k,i}(s_k(t), s_i(t)) + \sum_{j \in A_k} \omega_{j,k}(s_j(t), s_k(t))$$

where $\omega_{k,i}(s_k(t), s_i(t))$ for $i \in A_k$ are uniformly drawn integer random variables constrained so that $\sum_{i \in A_k} \omega_{k,i}(s_k(t), s_i(t)) \leq \alpha s_k(t)$, $(\alpha = 10^{-2})$. The set A_k gives the indices of robots within communication distance of robot k (which technically depend on t). We require symmetrical neighbourhood sets so that $i \in A_k \implies k \in A_i$. The dynamics rule says that robot i distributes random portions of its current state value among its neighbours. Despite using only local communication, this interaction rule conserves the global property, i.e., $\forall t, \sum_i^n s_i(t) = K$. We set initial conditions so that $K = m_a$.

This process has two defining parameters: n and m_a. In the case where $n \ll m_a$, the total number of states is a simple combinatorial exercise,

$$||\boldsymbol{S}(n, m_a)|| = \binom{m_a + n - 1}{n - 1}.$$

Define $\mathbf{G_j}([s^1, \ldots, s^n]) = s^j$. To get the average state for robot j, we apply Eqn. 2. Formally proving that Φ_a is ergodic is beyond the scope of this paper, but observe that the dynamics is symmetrical with respect to robots: no single robot is favoured over another. The dynamics randomly explores the configuration available to it. The full density function is unnecessary to calculate $\langle \mathbf{G_j} \rangle = \frac{m_a}{n}$.

A second process $P_b(n, m_b)$ is defined identically. Dynamics Φ_a and Φ_b explore their respective state spaces with expected state calculable in terms of n, m_a and m_b. The system is initialised with $m_a = 10^5$, $m_b = 10^5$. These m_a and m_b values are adjusted on each puck observation when the robot alters the local state of the processes $s^a(t)$ and $s^b(t)$. Observation of an A puck causes the following transition: $s^a(t + \delta t) = (1 - \gamma)s^a(t)$ and $s^b(t + \delta t) = \gamma s^a(t) + s^b(t)$ where γ is a tunable parameter. The converse happens on observing a B puck. Each robot independently decides which type of puck to forage using $s^a(t)$ and $s^b(t)$, the local states of the two processes. Pucks of variety A are chosen with probability $pr_{a-puck} = s^a(t)/(s^a(t) + s^b(t))$. Thus, the transitions simply skew the probability by a factor of γ.

Robots randomly encounter either type A or B pucks, making observations of each type in proportion with the puck distribution. These observations are smoothed by the dynamics of the two processes. Low probability observations

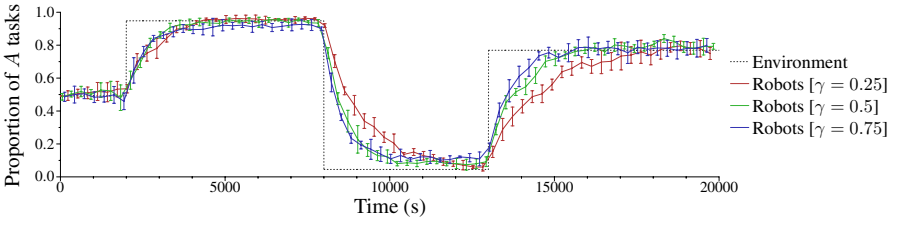


Fig. 1. Performance of task allocation processes. The vertical axis gives the proportion of tasks (for the broken line), and the division of robots among tasks (the solid lines). Plots show mean and standard deviation for 5 runs.

(e.g., observing ten types of minority puck) are averaged out over the entire group of robots.

We simulated 100 robots within a 64m×64m arena. Initially, 3000 pucks were randomly scattered throughout the arena. Pucks started in an initial 50%/50% distribution. Robots explored from random initial locations. After stumbling on an appropriate puck, the robot would transport it to the home region. For each puck foraged, a new one, of the same type, was introduced at a random location. Thus the puck density was maintained throughout.

The puck distribution was altered at three stages, at $t = 2000$ it was changed to 95%/5%, at $t = 8000$ to 5%/95%, and at $t = 13000$ to 75%/25%. In Figure 1, the dotted line shows the puck distribution. The plot shows experimental runs with three different settings for the γ parameter. The system shows hysteresis and a response time dependant on γ. In all cases, however, the system adapts so as to find a distribution applicable for the environmental conditions.

4.2 Domain 2: Collective Strategy Selection

Interference can adversely affect task performance in a multi-robot system. It is a particularly important factor in large-scale robot systems. Robot foraging is one of few problem domains for which interference has been well-studied (e.g., [12]) and is suitable for large-scale systems. Efficient strategy selection can ameliorate the negative effects inter-robot interference. More generally, however, this form of distributed decision-making underlies many tasks.

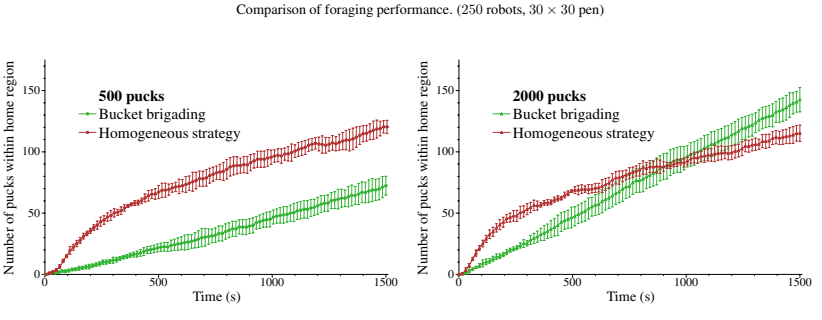


Fig. 2. Plot of the number of pucks within home region versus time for both foraging strategies (with 250 robots). Left figure has low puck density, right has high puck density. Plots show mean and standard deviation for 5 independent simulation runs.

In this domain we consider two distinct strategies for foraging (with a single puck type). The first, *homogeneous method*, involves each robot searching for a puck, locating a puck, and delivering that puck home [12]. In the alternative, *bucket brigading method* [19], each robot deals with only those pucks in a particular sub-region of the environment (also termed the caste method [12]). Figure 2 shows performance data for simulations with 250 robots within a 25m×25m

arena. The left graphs show data for 500 pucks. With that puck density the homogeneous strategy out performs bucket brigading. But with higher puck density (right graph is for 2000 pucks) the homogeneous strategy is inferior to bucket brigading. The homogeneous strategy results in considerable interference around the home region with high puck densities because a large number of pucks results in a short mean time between puck discoveries. Bucket brigading becomes more effective with increasing puck density because the robots have no perception of pucks at a distance. On the other hand, with too few pucks each robots takes a long time to find pucks dropped by its peers.

Each robot's controller consists of an implementation of both foraging strategies, with a binary variable to control the one currently in use. This variable provides an interface to the two coupled ergodic processes. The first is the process defined in the preceding section; call it $P_a(n, m_a)$. The second is defined as $P_c(n, m_c, e_c) = \langle \{-1, +1\}, \Phi_c \rangle$. Again that is for n robots. Here $m_c = \sum_{i=1}^{n} s_k(t)$. Less obviously, $e_c = \sum_{j,k,\ s.t.\ j \in A_k} -s_j(t)s_k(t)$ measures the number of neighbours that have the same state. The value of m_c gives a measure of "agreement" among robots, while e_c a measure of "frustration." A range of m_c values are applicable for a given e_c.

The dynamics Φ_c is constructed so that e_c remains constant. The process has only two states and the transition from one state to another (e.g., $+1$ to -1) is called a flip. Two robots within communication range, say i and j, calculate δe_i^c and δe_j^c, these are the changes in e_c that would result if robot i and j flip states. Both can be calculated using local information. If $\delta e_i^c = -\delta e_j^c$ then the two robots carry out this flip operation.

Both P_c and P_a execute a coupling dynamics that operates on the e_c and m_a parameters. The process on robot i may flip with a resulting δe_i^c, provided that this can be supplanted by subtracting an equivalent value from P_a. Thus a large m_a value can result in an increased e_c and hence effect the robot's "agreement" and, similarly, small m_a results in decreased e_c. The manner by which this change occurs is crucial for strategy selection.

Analysis of P_c's macroscopic behaviour is less obvious than P_a. The process has the same structure as the Ising ferromagnetic model [11], which has been well studied in the limit of infinite system size, and under controlled temperature conditions. In the limit the model exhibits a symmetry breaking phase-transition from mixed spin values (with $m_c \sim 0$) to a state with alignment ($m_c = 1$ or $m_c = -1$).

During the foraging task, local sensing of task progress enables the m_a value to be tuned appropriately. While bucket brigading, each robot kept track of the time between puck discoveries, a noisy local measurement of puck density. For homogeneous foraging, interference was estimated by measuring distance travelled for a short time. Both of these decreased the local value of $s^a(t)$ on each robot. Each time a puck was dropped over the home region $s^a(t)$ was decreased. The system achieves a steady-state between m_a creation (through interference, obstacle avoidance) and m_a deletion (from pucks homed). Different

puck densities have different steady-states, and a mismatch of puck density and strategy is sufficient to drive the phase transition of P_c.

5 Discussion

Statistical mechanics methods enable predictions of behaviour by characterising macroscopically identical systems. These *ensemble* predictions are feasible in systems with hundreds of robots, and with increasing system size predictions become progressively accurate. We believe that, for large systems, existing metaphors — e.g., message-passing — break down and must be superseded. We thus explore and exploit the benefits afforded by the large number of degrees-of-freedom, in which we include both aspects of the physical robot and controller state-space.

The two foraging domains were carefully chosen. The "decentralised task allocation" domain required estimation of a continuous quantity. It is typical of the calculations and optimisation that might be performed by an implicitly coordinated swarm system. Small local estimates can be used to make a decision, and that can shared with other agents easily. This notion seems similar to the Downhill Principle [27], but in a distributed sense.

On the other hand, explicitly coordinated systems often solve discrete problems with hard constraints. For such problems, gradient techniques are not useful. The "collective strategy selection" was intended to demonstration that discrete notions can also be feasibly tackled with the ergodic process approach. Since communication times are not zero, the system does take time to transition between states. The transition is all-or-nothing, with infinite size.

An unconventional aspect of the proposed methodology is the focus on equilibrium solutions rather than dynamics in computing. In a sense this is not unlike the electronic analogue computers of the past, in which the initial transients were ignored, with attention paid to steady-state solutions [24]. In robot systems, it is envisioned that changes within environmental factors would trigger adaptation within the ergodic processes, as they reach a new equilibrium.

6 Conclusion

This paper has argued that non-conventional computing models may be a way to elegantly address the challenges raised by physically situated robots. We defined our own compositional method for synthesising controllers for large-scale multi-robot systems and proposed the use of ergodic processes as elemental distributed building blocks. This contrasts with current methods that produce task-oriented behaviour through dynamics with rich temporal structure. Two complementary coordinated foraging problems were used to demonstrate the proposed method.

Acknowledgements. Support for this work was provided by the National Science Foundation Grant No. IIS-0413321. We thank G. Sibley for proofreading.

References

1. A. Adamatzky, B. De Lacy Costello, C. Melhuish, N. Ratcliffe, Experimental reaction–diffusion chemical processors for robot path planning, *Journal of Intelligent Robotic Systems*, 37 (2003), 233–249.
2. R.C. Arkin. T. Balch. E. Nitz, Communication of behavioral state in multi-agent retrieval tasks, *Proceedings IEEE Conference on Robotics and Automation*, Atlanta, GA, 1993, 588–594.
3. R. Beckers, O.E. Holland, J.-L. Deneubourg, From Local Actions to Global Tasks: Stigmergy and Collective Robotics, *Artificial Life IV*, 181–189, Cambridge, MA, July 1994.
4. G.A. Bekey, *Autonomous Robots: From Biological Inspiration to Implementation and Control*, MIT Press, Cambridge, MA, 2005.
5. E.R. Berlekamp, J.H. Conway, R.K. Guy, *Winning Ways for Your Mathematical Plays*, Academic Press, New York, 1982.
6. E. Bonabeau, M. Dorigo, G. Thraulaz, *Swarm Intelligence: From Natural to Artificial Systems*, Oxford University Press, New York, NY, 1999.
7. R.A. Brooks, Intelligence Without Reason, *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI-91)*, Sydney, Australia, 1991, 569–595.
8. R.A. Brooks, *Cambrian Intelligence: The Early History of the New AI*, MIT Press, Cambridge, MA, 2001.
9. J.-L. Deneubourg, S. Goss, N.R. Franks, A.B. Sendova-Franks, C. Detrain, L. Chrétien, The Dynamics of Collective Sorting Robot-like Ants and Ant-like Robots, *Proc. First International Conference on Simulation of Adaptive Behavior*, Paris, France, 1990, 356–363.
10. B. Donald, J. Jennings, D. Rus, Minimalism + Distribution = Supermodularity, *Journal of Experimental and Theoretical AI*, 9, 20 (1997), 293–321.
11. M.E. Fisher, The theory of equilibrium critical phenomena, *Reports on Progress in Physics*, 30 (1967), 615–730.
12. D. Goldberg, *Evaluating the Dynamics of Agent-Environment Interaction*, Department of Computer Science, University of Southern California, May 2001.
13. T. Hogg, B.A. Huberman, Controlling chaos in distributed systems, *IEEE Transactions on Systems, Man, and Cybernetics (Special Section on DAI)*, 21, 6 (1991), 1325–1332.
14. O.E. Holland, C. Melhuish, Stigmergy, Self-Organization, and Sorting in Collective Robotics, *Artificial Life*, 5, 2 (1999), 173–202.
15. C.V. Jones, M.J. Matarić, Adaptive Division of Labor in Large-Scale Minimalist Multi-Robot Systems, *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 1969–1974, Las Vegas, NV, Oct. 2003.
16. C.V. Jones, D.A. Shell, M.J. Matarić, B.P. Gerkey, Principled Approaches to the Design of Multi-Robot Systems, Invited contribution to *Workshop on Networked Robotics, International Conference on Intelligent Robots and Systems*, Sendai, Japan, 2004, 71–80.
17. C.R. Kube, H. Zhang, Collective robotics: From social insects to robots, *Adaptive Behavior*, 2, 2 (1993), 189–219.
18. J.C. Latombe, *Robot Motion Planning*, Kluwer Academic Publishers, Norwell, MA, 1991.

19. E. Østergaard. G.S. Sukhatme, M. J. Matarić, Emergent Bucket Brigading — A Simple Mechanism for Improving Performance in Multi-Robot Constrained-Space Foraging Tasks, *International Conference on Autonomous Agents*, Montreal, Canada, May 2001, 29–30.
20. L.E. Parker, ALLIANCE: An architecture for fault-tolerant multi-robot cooperation, *IEEE Transactions on Robotics and Automation*, 14, 2 (1998), 220–240.
21. D. Payton, M. Daily, R. Estkowski, M. Howard, C. Lee, Pheromone robotics, *Autonomous Robots*, 11, 3 (2001), 319–324.
22. D. Pomerleau, *Neural Network Perception for Mobile Robot Guidance*, Kluwer Academic Publishers, Norwell, MA, 1993.
23. H.A. Simon, *The Sciences of the Artificial*, MIT Press, Cambridge, Massachusetts, 1996, third edition.
24. J.S. Small, *The Analogue Alternative: The Electric Analogue Computer in Britain and the USA, 1930-1975*, Routledge, London and New York, 2001.
25. S. Thrun, W. Burgard, D. Fox, *Probabilistic Robotics*, MIT Press, Cambridge, MA, 2005.
26. R.T. Vaughan, K. Støy, G.S. Sukhatme, M.J. Matarić, Blazing a trail: insect-inspired resource transportation by a robot team, *Proceedings of the International Symposium on Distributed Autonomous Robotic Systems*, Knoxville, TN, 2000, 111–120.
27. A. Vergis, K. Steiglitz, B. Dickinson, The complexity of analog computation, *Mathematics and Computers in Simulation*, 28 (1986), 91–113.
28. M. Weiser, Some Computer Science issues in Ubiquitous Computing, *Communications of the ACM*, 36, 7 (1993), 75–84.
29. N. Wiener, *Cybernetics, or Control and Communication in the Animal and the Machine*, MIT Press, Cambridge MA, 1962, second edition.

Author Index

Abbott, Russ 41

Balan, M. Sakthi 57

Cardona, Mónica 72
Coelho, Francisco 195
Colomer, M. Angels 72
Costa, José Félix 195

Dreyfus, Gérard 1
Duprat, Arthur 1

Flarup, Uffe 86
Fujio, Mitsuhiko 181

Gorecka, Joanna Natalia 130
Gorecki, Jerzy 130
Goulon, Aurélie 1

Hines, Peter 101

Ibarra, Oscar H. 113
Igarashi, Yasuhiro 130
Inokuchi, Shuichi 181

Jonoska, Nataša 139
Jürgensen, Helmut 57

Krishna, Shankara Narayanan 152, 243

Langdon, W.B. 166

Matarić, Maja J. 254
McColm, Gregory L. 139
Meer, Klaus 86
Mikoda, Akihiro 181
Mizoguchi, Yoshihiro 181
Mozer, Michael C. 20
Mycka, Jerzy 195

Ninagawa, Shigeru 207

Oltean, Mihai 217

Păun, Andrei 113
Pérez-Jiménez, Mario J. 72
Polani, Daniel 228
Prokopenko, Mikhail 228

Rama, Raghavan 243
Ramesh, H. 243

Shell, Dylan A. 254

Wang, Peter 228
Winfree, Erik 26
Woods, Damien 27
Woodworth, Sara 113

Yu, Fang 113

Zaragoza, Alba 72