Modelling Local Transition Functions in Cellular Automata Using Associative Memories

Aaron Turner

Submitted for the degree of Msc by Research

University of York

Department of Computer Science

8 May 97
Abstract

Cellular automata are based on local interactions, formed on a grid, each grid position having a state, the new state being based on that of its surroundings, its neighbourhood. The creation of a new state for a cell is governed by the local transition function.

This thesis investigates the modelling of local transition functions of cellular automata by using associative memories, specifically binary correlation matrix memories and the Advanced Distributed Associative Memory (ADAM), by considering the local transition function to be a rule-based system. Symmetries are sought in the local transition function to enable the efficiency of the implementation to be improved by coding the neighbourhood in a way that succinctly codifies the symmetry. Furthermore a scheme is presented that allows efficient coding of a set of transitions that are represented by a set of transitions with symmetry, with the addition of some transitions that break this symmetry.

The performance of the system is compared to that of implementations without the use of associative memories for two well known automata, Conway’s “Life” and C. G. Langton’s self-reproducing automaton. Also the functioning of a cellular automaton with a larger number of states and a very large number of transitions is examined.

Finally there is an overview of fault-tolerance, adaptive behaviour, and rule-extraction, although time did not permit a sufficiently in-depth investigation of these areas.

The conclusion of the thesis is primarily that for smaller systems the performance in terms of speed of the associative memory implementation is disappointing, if accuracy as high and non-associative memory implementation is required. Due to time constraints insufficient data was gathered on larger systems to allow a conclusion regarding their behaviour to be drawn, but the systems examined were insufficiently large. It is felt that associative memory implementations may have benefits for very large systems, as implied by results from papers by Song Yan.

The investigations are primarily based on empirical data.
Table of Contents

1 Overview 17

2 Local Transition Functions in Cellular Automata 19
   2.1 Cellular Automata 19
   2.2 Neighbourhoods and kernels 20
   2.3 The local transition function 21
   2.4 The global transition function 22
   2.5 Non local interactions 23
   2.6 Determining global transition functions from local transition functions 23
   2.7 Determining local transition functions from global transition functions 23

3 Degeneracy in Local Transition Functions 25
   3.1 Why Consider Degeneracy? 25
   3.2 Degeneracy categorisation 26

4 Associative Memories 27
   4.1 Introduction 27
   4.2 Correlation Matrix Memories 27
      4.2.1 Binary Correlation Matrix Memories 27
      4.2.2 Collisions between inputs, outputs, and already set bits 28
      4.2.3 Binary vectors and orthogonality 29
   4.3 ADAM 31
      4.3.1 Introduction 31
      4.3.2 Class Codes 31
      4.3.3 Generating Class Codes 32
      4.3.4 Thresholding Class Codes 32
      4.3.5 Thresholding and partial matching 33
      4.3.6 Class Codes and Input Ordering 34
      4.3.7 Recall time 35
5 Basic Architecture

5.1 Non-associative memory implementations of local transition functions 39
5.2 Local transition functions as rule-based systems 39
5.3 State machines and connections 40
5.4 Representation of states 40
5.5 Representation of neighbourhood state 41
5.5.1 ADAM 43
5.6 The spreader 44
5.6.1 Description 44
5.6.2 Heterogeneity and time variations 44
5.6.3 Why use a spreader? 45

6 Anisotropic matching system 47
6.1 Representation of the neighbourhood state 47
6.2 Ill-conditioned inputs 48
6.3 Conclusion 51

7 Isotropic symmetry solution: tensors 53
7.1 Aim 53
7.2 Bound Strings to Represent Neighbourhood States 53
7.3 States and neighbourhood positions 54
7.4 Comparison of Using Bound Strings and Neighbourhood Positions 56
7.5 Conclusion 57

8 Isomorphic System Using Occurrence Checking 59
8.1 Introduction 59
8.2 Method of Filer and Austin 59
8.3 Ill conditioned inputs 60
8.3.1 Mapping between code sets 62
## List of Figures

<table>
<thead>
<tr>
<th>Fig. 1</th>
<th>The action of a neighbourhood template function</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fig. 2</td>
<td>Neighbourhoods and kernels</td>
<td>21</td>
</tr>
<tr>
<td>Fig. 3</td>
<td>Rotations of a neighbourhood</td>
<td>25</td>
</tr>
<tr>
<td>Fig. 4</td>
<td>Isomorphs of a neighbourhood</td>
<td>26</td>
</tr>
<tr>
<td>Fig. 5</td>
<td>Teaching first association</td>
<td>28</td>
</tr>
<tr>
<td>Fig. 6</td>
<td>Teaching second association</td>
<td>28</td>
</tr>
<tr>
<td>Fig. 7</td>
<td>Recovering Raw Class</td>
<td>28</td>
</tr>
<tr>
<td>Fig. 8</td>
<td>Collisions in a correlation matrix</td>
<td>29</td>
</tr>
<tr>
<td>Fig. 9</td>
<td>ADAM teach and recall</td>
<td>31</td>
</tr>
<tr>
<td>Fig. 10</td>
<td>Various thresholding algorithms for ADAM class codes</td>
<td>33</td>
</tr>
<tr>
<td>Fig. 11</td>
<td>Partial matching for ADAM</td>
<td>34</td>
</tr>
<tr>
<td>Fig. 12</td>
<td>Partial matching for ADAM</td>
<td>34</td>
</tr>
<tr>
<td>Fig. 13</td>
<td>Linear tupling of size 4</td>
<td>37</td>
</tr>
<tr>
<td>Fig. 14</td>
<td>Simple architecture (here a simple 1D automaton is shown for clarity)</td>
<td>40</td>
</tr>
<tr>
<td>Fig. 15</td>
<td>State machines and associative memories</td>
<td>41</td>
</tr>
<tr>
<td>Fig. 16</td>
<td>Determining new cell state in a 1D automaton</td>
<td>42</td>
</tr>
<tr>
<td>Fig. 17</td>
<td>Advantage of using ADAM when there are few neighbourhood codes</td>
<td>43</td>
</tr>
<tr>
<td>Fig. 18</td>
<td>Architecture with spreader</td>
<td>44</td>
</tr>
<tr>
<td>Fig. 19</td>
<td>Inputs to a cell</td>
<td>47</td>
</tr>
<tr>
<td>Fig. 20</td>
<td>Ill-conditioning in training of a 1D binary automaton</td>
<td>49</td>
</tr>
<tr>
<td>Fig. 21</td>
<td>Random tupling</td>
<td>50</td>
</tr>
<tr>
<td>Fig. 22</td>
<td>States and neighbourhood positions</td>
<td>55</td>
</tr>
<tr>
<td>Fig. 23</td>
<td>Rotationally equivalent neighbourhoods</td>
<td>55</td>
</tr>
<tr>
<td>Fig. 24</td>
<td>Steps of the occurrence checking process</td>
<td>59</td>
</tr>
<tr>
<td>Fig. 25</td>
<td>Idealised graph of Hamming distances of set of ill-conditioned inputs</td>
<td>61</td>
</tr>
<tr>
<td>Fig. 26</td>
<td>Idealised of graph of desired Hamming distances for well-conditioned inputs</td>
<td>61</td>
</tr>
<tr>
<td>Fig. 27</td>
<td>Variability after linear tupling</td>
<td>62</td>
</tr>
<tr>
<td>Fig. 28</td>
<td>Exception to standard Conway’s Life</td>
<td>73</td>
</tr>
<tr>
<td>Fig. 29</td>
<td>Partitioning of classes of transition</td>
<td>74</td>
</tr>
<tr>
<td>Fig. 30</td>
<td>Composite System</td>
<td>76</td>
</tr>
<tr>
<td>Fig. 31</td>
<td>Degenerate neighbourhoods for isomorphic system</td>
<td>77</td>
</tr>
<tr>
<td>Fig. 32</td>
<td>Anisotropic exception</td>
<td>77</td>
</tr>
<tr>
<td>Fig. 33</td>
<td>Partial matching</td>
<td>78</td>
</tr>
<tr>
<td>Fig. 34</td>
<td>Configurations of automaton for <strong>Graph 32</strong></td>
<td>88</td>
</tr>
<tr>
<td>Fig. 35</td>
<td>Neighbourhood state ADAM, saturated, on recall</td>
<td>107</td>
</tr>
<tr>
<td>Fig. 36</td>
<td>Saturation in kernel state ADAM</td>
<td>107</td>
</tr>
<tr>
<td>Fig. 37</td>
<td>Superposition of state codes in the neighbourhood</td>
<td>108</td>
</tr>
<tr>
<td>Fig. 38</td>
<td>Incomplete or missing state codes in the neighbourhood</td>
<td>109</td>
</tr>
<tr>
<td>Fig. 39</td>
<td>The transitions for elements of the ring are of the form</td>
<td>112</td>
</tr>
</tbody>
</table>
Fig. 40  Key for Fig. 43 to Fig. 48 113
Fig. 41  Target ring to be supported in Fig. 43 to Fig. 48 113
Fig. 42  Transitions required for completion of ring 113
Fig. 43  Orthogonal classes- here no multiple states 113
Fig. 44  run 1 (390 rules learned correctly) Class1: 2 set in 256 113
Fig. 45  run 2 - 330 rules learned correctly, Class 1: 2 set in 256 114
Fig. 46  run 3 - 242 rules learned correctly: Class 1: 2 set in 256 114
Fig. 47  run 4 - 32 rules learned correctly: Class 1: 2 set in 256 114
Fig. 48  run 5 - 6 rules learned correctly: Class 1: 2 set in 256 114
Fig. 49  Same automaton with error state 115
Fig. 50  Same automaton with error state and element of subtractive noise 115
Fig. 51  7 elements missing 115
Fig. 52  6 elements missing 115
Fig. 53  5 elements missing 116
Fig. 54  4 elements missing 116
Fig. 55  3 elements missing 116
Fig. 56  2 elements missing 116
Fig. 57  Example of a system learning 120
Fig. 58  Configurations to learn 120
List of Graphs

Graph 1  Recall time as a function of class size and number of bits set for 100 patterns 35
Graph 2  Recall time as a function on input pattern parameters with a class size of 100, with 2 bits set 36
Graph 3  Percentage of transitions learned successfully 56
Graph 4  Number of samples per code position for a number of tuple maps with 16 tuples of size 4 64
Graph 5  Number of samples per bit position for 16 tuples of size 4 65
Graph 6  Number of samples per bit position for 24 tuples of size 4 65
Graph 7  Number of samples per bit position for 32 tuples of size 4 65
Graph 8  Number of samples per bit position for tuple size 5, 32 tuples 66
Graph 9  Number of samples per bit position for tuple size 6, 32 tuples 66
Graph 10 Deviation between theory and experiment for varying numbering of tuples, with tuple size 4 66
Graph 11 Deviation between theory and experiment for varying tuple size, with 32 tuples 67
Graph 12 Saturation with random tupling of 10 tuples of size 4 with and without replacement 67
Graph 13 Saturation with original patterns and probabilistic random tupling of 16 tuples of size 4 68
Graph 14 Saturation with probabilistic random tupling and random tupling with 32 tuples of size 4 68
Graph 15 Saturation with tupling with 8 tuples of size 5 69
Graph 16 Saturation with 16 tuples of size 5 69
Graph 17 Saturation with 16 tuples of size 6 69
Graph 18 Saturation with probabilistic random tupling of various sizes 70
Graph 19 16 tuples of various sizes 71
Graph 20 32 tuples of various sizes 71
Graph 21 Teaching times as anisotropic transitions are added 79
Graph 22 Iteration times for 25x25 automaton as anisotropic transitions are added 80
Graph 23 Time to teach automaton of size 25x25, with addition of isotropic transitions 80
Graph 24 Time for one iteration of 25x25 automaton, with addition of isotropic transitions 81
Graph 25 Time required to create and recall patterns for an anisotropic system as a function of class size 82
Graph 26 Time required to create and recall patterns for an isotropic system as a function of class size 82
Graph 27 Time required to create and recall patterns for an isomorphic system as a function of class size 82
Graph 28 Transitions learned correctly with anisotropic transitions overlaying isomorphic 83
Graph 29 Transitions learned correctly with isotropic transitions overlaying isomorphic. 84
Graph 30 Anisotropic system. 512 randomly selected codes of size 256 with 2 bits set to 1 87
Graph 31 Anisotropic system. 512 randomly selected codes of size 1024 with 2 bits set to 1 87
Graph 32 Hamming distance between reference system and associative memory system with 512 codes 88
Graph 33 Configuration densities over time for reference system with various initial configurations 89
Graph 34 Configuration densities over time for anisotropic system with various initial configurations 89
Graph 35 Configuration densities over time for isotropic system with various initial configurations 89
Graph 36 Configuration densities over time for isomorphic system with various initial configurations 90
Graph 37 Iteration times for reference system given various initial configuration densities 90
Graph 38 Iteration times for anisotropic system given various initial configuration densities 91
Graph 39 Iteration times for isotropic system given various initial configuration densities 91
Graph 40 Iteration times for isomorphic system given various initial configuration densities 91
| Graph 41 | Teaching times for anisotropic system given various class code parameters | 92 |
| Graph 42 | Teaching times for isotropic system given various class code parameters | 92 |
| Graph 43 | Teaching times for isomorphic system given various class code parameters | 92 |
| Graph 44 | Recall accuracy for anisotropic system given various class code parameters | 93 |
| Graph 45 | Recall accuracy for isotropic system given various class code parameters | 93 |
| Graph 46 | Recall accuracy for isomorphic system given various class code parameters | 93 |
| Graph 47 | Matrix 1 statuation for anisotropic system with various class code parameters | 94 |
| Graph 48 | Matrix 1 statuation for isotropic system with various class code parameters | 94 |
| Graph 49 | Matrix 1 statuation for isomorphic system with various class code parameters | 94 |
| Graph 50 | Matrix 2 statuation for anisotropic system with various class code parameters | 95 |
| Graph 51 | Matrix 2 statuation for isotropic system with various class code parameters | 95 |
| Graph 52 | Matrix 2 statuation for isomorphic system with various class code parameters | 95 |
| Graph 53 | Time to teach transitions for Langton’s Automaton | 98 |
| Graph 54 | Proportion of transitions correctly learned for Langton’s Automaton | 99 |
| Graph 55 | Time per cell per iteration for Langton’s Automaton | 100 |
| Graph 56 | Expected | 100 |
| Graph 57 | Simulated (for an automaton with a first ADAM class size of 448, neighbourhood code size of 96) | 101 |
List of Tables

Table 1  Codes for a simple 4 state automaton  47
Table 2  Linear Tupling of neighbourhood state codes  50
Table 3  Random tupling of neighbourhood state codes  51
Table 4  Times for recall with orthogonal class code vectors  81
Table 5  Proportion of transitions learned correctly for massive random system  104
Table 6  Time required for teaching and iterating a massive random system  104
Table 7  Transitions  106
1 Overview

Cellular automata are networks of interconnected and interacting elements, that rely on local connections. This thesis examines the implementation of these cells using associative memories, specifically the Advanced Distributed Associative Memory (ADAM).

A cellular automaton cell relies on interactions with its near neighbours. The thesis shows that this interaction may be modelled using an associative memory or system of associative memories, provided that the problem is converted into appropriate terms via a process of tokenisation.

However, the process of modelling can be enhanced if degeneracy is taken into account, i.e. some information about the relative disposition of the nearest neighbours may be ignored for some cellular automata, for at least some of the possible transitions. If these degeneracies can be exploited via appropriate preprocessing before information is presented to the associative memories then the associative memories may be made more efficient.

The thesis then goes on to show how a general set of transitions may be represented as a combination of transitions with various levels of degeneracy, and how this may be implemented using a system of several associative memories.

However, for some levels of degeneracy there are problems with conditioning the inputs such that the associative memories may function at their maximum efficiency.

After examining the performance of the various systems for various levels of degeneracy the thesis then goes on to examine two examples of widely known automata, Conway’s “Life”, and Langton’s self-reproducing automata. These were chosen because of their prominence in cellular automaton circles, and they are automata widely implemented by others in other ways. In addition an arbitrarily large automaton was created to assess the performance of the system for large numbers of transitions.

Examination was made of the issue of computing with unreliable elements, at a fairly cursory level.

A brief examination was also made of the possibility of evolving automata to perform specific calculations or sequences of actions.
2 Local Transition Functions in Cellular Automata

This chapter presents a brief overview of the theory of cellular automata, neighbourhoods, local and global transition functions.

2.1 Cellular Automata

Cellular automata were first developed in the 1930s by Ulam and Schrandt [1][2], but John von Neumann [4][5][6][7][8] and E. F. Codd [9] did much important work on them in the 1950s and 60s.

A cellular automaton consists of a grid of cells, of arbitrary shape in an arbitrary number of dimensions. Each cell is a small finite automaton or state machine, which determines its new state depending on its current state and the states of cells around it. The state machines have distinct, discrete steps, the mathematics of determining new states being also discrete. The shape of the grid, neighbourhood, and the way that the new states are determined are the fundamentals of the study of cellular automata. Usually the updating of the cells from one iteration to the next is synchronous, i.e. the new cell states are based on the values of cells in the current configuration. Asynchronous updating allows the state of the cell to be replaced with a new value before values for each and every cell have been calculated.

A cellular automaton comprises the grid of cells. Each cell has a neighbourhood of cells around it, and a local transition function which determines the calculation of the new state from old state and the state of the cells in the neighbourhood of that cell.

The way the set of states progress is a global behaviour that emerges from the interaction of the cells, whose behaviour is determined locally. Cellular automata are characterised as systems in which complex global behaviour emerges from local transitions (which may be quite simple).

Cellular automata are of interest for two major reasons:

1. The matrix of local interactions is similar in form to many physical processes - e.g. diffusion of gases, growth of bacteria colonies, and so on, and so offers a way to model these complex and non-linear systems using a series of small cells, which often may be individually modelled linearly.

2. A subset of all possible cellular automata can be shown to have the properties of a Turing machine, i.e be capable of supporting universal computation [10][11]. This means that a cellular automaton could be used as a general computation device (subject to one’s being able to program it effectively [12][14][15]). Cellular automata can be created to use unreliable components and so offer a route to fault-tolerant computation [8][17][18][19].
2.2 Neighbourhoods and kernels

In a cellular automaton connections are generally defined to be local, and are characterised by the neighbourhood. The neighbourhood is a set of cells of the automaton which a cell is connected to. The kernel is the union of the neighbourhood and the cell.

The neighbourhood function, $g$, maps an individual cell to a number of cells, defining the kernel. The kernel is the result of this mapping. The neighbourhood is generally taken to exclude the cell itself, but the kernel is defined to be inclusive of the cell itself.

An example of a mapping defined by a neighbourhood function might be:

$$
\alpha \xrightarrow{g} \{\alpha, \alpha + \delta_1, \alpha + \delta_2, \ldots, \alpha + \delta_n\}
$$

The neighbourhood function defines the neighbourhood template, which is the set of vectors $\{\delta_1, \delta_2, \ldots, \delta_n\}$ that define the offsets from $\alpha$ to the elements of the neighbourhood.

If the neighbourhood is of size $n$, or has $n$ elements, $\delta_i$ indicates some vector indicating the spatial offset from $\alpha$ to the $r$th element of the neighbourhood.

**FIGURE 1. The action of a neighbourhood template function**

The use of neighbourhood functions was studied by Ulam and Schrandt for the growth of figures [1][2].

Neighbourhoods are often characterised by the number of cells in the neighbourhood or kernel. There is some inconsistency in the literature and in this thesis a neighbourhood consisting of $n$ nearest neighbours will be referred to by the shorthand $n$-neighbourhood. The kernel size is $n+1$. The neighbourhood may also be characterised by its radius, which indicates the number of cells it spreads out to, this having most meaning if the neighbourhood is symmetrical and continuous. In figure 1 above the neighbourhood has a radius of 1. Some specific neighbourhoods may have particular names, such as the Moore neighbourhood for an 8-neighbourhood of radius 1 defined on a square two-dimensional grid.
Generally neighbourhoods are represented on square grids, are non-disjoint, and symmetrical. These are simplifying instructions but they need not apply. Non symmetrical neighbourhood templates can imply a flow of information from one end of an automaton to another. Disjoint neighbourhood templates can imply longer range interactions.

The neighbourhood state function, $h$, is a function operating on the cell which gives the states of the cells in the neighbourhood of the cell at that time instance. If the state of a cell, $\alpha$, at time $t$ is $\nu^t(\alpha)$ then $h$ is defined as

$$h^t(\alpha) = (\nu^t(\alpha), \nu^t(\alpha + \delta_1), \ldots, \nu^t(\alpha + \delta_n)) \quad (2)$$

**FIGURE 2.** Neighbourhoods and kernels

<table>
<thead>
<tr>
<th>Neighbourhood Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>radius 1</strong></td>
<td></td>
</tr>
<tr>
<td>4-neighbourhood defined on a square grid</td>
<td></td>
</tr>
<tr>
<td>8-neighbourhood defined on a square grid (Moore neighbourhood)</td>
<td></td>
</tr>
<tr>
<td>12-neighbourhood defined on a square grid</td>
<td></td>
</tr>
<tr>
<td>16-neighbourhood defined on a hexagonal grid</td>
<td></td>
</tr>
<tr>
<td>6-neighbourhood defined on a 3d rectangular grid</td>
<td></td>
</tr>
<tr>
<td><strong>radius 2</strong></td>
<td></td>
</tr>
<tr>
<td>8-neighbourhood with disjoint element defined on a square grid</td>
<td></td>
</tr>
<tr>
<td>18-neighbourhood defined on a square grid</td>
<td></td>
</tr>
<tr>
<td>22-neighbourhood defined on a square grid</td>
<td></td>
</tr>
</tbody>
</table>

### 2.3 The local transition function

A cell in a cellular automaton is a finite automaton and the new state of the cell is defined via the local transition function, $f$, acting on the current state of the cell and the states of the cells in its neighbourhood, i.e. on the kernel.

If we define $\nu^t(\alpha)$ to be the state of cell at time $t$ then the local transition function allows the new state of the cell to be calculated thus

$$\nu^{t+1}(\alpha) = f(h^t(\alpha)) \quad (3)$$
The set of all states of the cells across all cells is termed the alphabet of the automaton.

The research in this thesis shows a novel way of modelling the local transition function, $f$, using the techniques of associative memories.

### 2.4 The global transition function

The global transition function $F$ is a consequence of the action of the local transition function on all cells $\alpha \in I \times I$, where $I \times I$ indicates the automaton. Ordinarily the new states of any $\alpha_{ij} \in I \times I$ do not replace the old state until all the new states have been calculated, thus the updating is synchronous. Effectively we have a corresponding automaton, $I \times I'$ and for each $\alpha_{ij} \in I \times I$ we have a corresponding $\alpha'_{ij} \in I \times I'$, and we calculate

$$u(\alpha'_{ij}) = f(h(\alpha_{ij})) \text{ for all } i \text{ and } j \text{ such that } \alpha_{ij} \in I \times I$$

and then

$$u(\alpha_{ij}) = u(\alpha'_{ij}) \text{ for all } i \text{ and } j \text{ such that } \alpha'_{ij} \in I \times I$$

This restriction simplifies the possible analytical derivation of $F$ from $f$, although in general it is not possible to determine $F$ given $f$. This restriction may be relaxed in various ways [16] such that new states of cells are calculated asynchronously with the calculations of the states of other cells.

Work by Reynolds on modelling flocking behaviour in birds suggested that behaviours which appear to be generated by a global process $G$ (which is a global transition function) operating over a time scale $\tau$ to $\tau + \Delta \tau$ may actually be produced by some global transition function $G$ composed of local transition functions $f_1f_2...f_i...f_I$ operating on a time scale with granularity $\Delta \tau$.

Typically cellular automata are homogenous, i.e. $f$ is the same for every cell in the automaton. This is another restriction that may be relaxed [20][21][22], such that a particular local transition function, $f_r$, acts over some subset of the cells of $I \times I$. For this to be possible the automaton as a whole must have an alphabet and thus there must be closure for each function $f_r$ that operates on the automaton, i.e. if the function $g_r$ operates over some set of states $w_r$ and the alphabet of the automaton as a whole is $W$ and there are $n$ distinct local transition functions:

$$W \subseteq \bigcup_{r=0}^{n} w_r$$

must be true.
Also any $g$, operating on any group of elements of $W$ must only produce element of $W$.

This naturally assumes that $g$ is the same for all cells. In theory we may allow a heterogeneous system such that rather than $g$, for a particular cell $\alpha_{ij} \in I \times I$ we have $g_{ij}$ (in a two dimensional case).

### 2.5 Non local interactions

It is possible for neighbourhoods to be comparatively large or even disjoint (see Fig. 2). This begins to contravene the basic tenets of cellular automata in that cellular systems are normally based on purely local interactions. Such non-local interactions may be powerful and enable various computations and be conceptually easier. The resultant behaviour may be very complex.

### 2.6 Determining global transition functions from local transition functions

For some very simple systems it is clear what the global behaviour of the whole automaton will be, given an initial configuration. In general, however, this is not the case [26] and the only way to be sure of the exact behaviour is to set the automaton running and see what happens. It seems possible to characterise the general behaviour of automata into a number of broad classes (automata that enter steady states, automata that cycle through a number of states, automata that appear completely chaotic) [51][56] although it is not possible to make this classification, in general, a priori based on the local transitions.

### 2.7 Determining local transition functions from global transition functions

Often the global behaviour that a system should simulate is known, but the local interactions underpinning the global behaviour are not known. There are techniques available to extract the required local behaviour from examples of the global behaviour. In practice this is extremely difficult (see 2.6) above. There are two techniques for this.

1. **Rule extraction.** A formal system to extract local rules from the global behaviour.

2. **Genetic Algorithms.** A system which allows rules to vary in a way analogous to evolution in biological systems, which seeks to evolve a solution (set of local transitions) which are ‘fittest’, i.e. model the global behaviour.

However, typically automata to perform particular functions have been coded by hand, with the behaviour examined and the local transitions modified gradually until the desired effect is achieved. It should be noted that finding systems which exhibit self-reproduction and fault tolerance is more than doubly difficult.
3 Degeneracy in Local Transition Functions

Degeneracy may be present in the local transition functions of cellular automata. This chapter explains how exploiting this degeneracy offers a way of making the implementation of cellular automata simpler and more compact.

3.1 Why Consider Degeneracy?

Degeneracy occurs when more than one set of states in the kernel of a cell results in the same final state. This is analogous to degeneracy in the field of statistical mechanics when one or more configurations of particles may have the same level of energy. Here the analogy is between the energy of the configurations of particles, and the new state of the cell in the automaton.

Typically many cellular automata have degeneracy in the local transition functions. Consider Conway’s life [52]. This is a comparatively simple system, being binary and with eight nearest neighbours. The rules are comparatively simple totalistic rules, e.g. if a cell has one neighbour that is ‘alive’ (in state 1) then in the next iteration it will be in state 0 (‘dead’). If we were forced to distinguish between the elements of the neighbourhood then the simple statement “one neighbour that is ‘alive’” would actually have to be translated into “neighbour 0 alive, others dead; or neighbour 1 alive, others dead; etc.”. This way of describing the transition function presumes no knowledge of the degeneracy that underlies the local transition function.

If we enumerated all the possible neighbourhood states for Conway’s Life in detail, ignoring any degeneracy we would have 256 possible combinations.

If we consider rotational symmetry, then the 256 possible combinations could be reduced to 36.

However, Conway’s life is essentially interested in the number of cells in the neighbourhood in a particular state,
Many possible equivalencies of neighbourhood are possible, based on various levels of degeneracy, e.g. rotational symmetry, partial rotational symmetry, reflection, equivalencies based on the number of cells in particular states, relative and absolute position being irrelevant.

### 3.2 Degeneracy categorisation

In the thesis three major categories of degeneracy in the neighbourhoods will be considered. These are categories common over many shapes of neighbourhood, and are typical of degeneracy found in cellular systems. The terminology is drawn from physics.

**anisotropic**
The exact position and state of each cell in the neighbourhood are both important for each element. The behaviour is different in different directions. No degeneracy.

**isotropic**
The relative position of cells to other cells in the neighbourhood is important, but the rotation of this configuration of cells in the neighbourhood around the central cell is not important, provided the interrelations within the neighbourhood are preserved. Rotation is not important. This presumes that the shape of the neighbourhood is such that rotation is possible.

**isomorphic**
The position and relative positions of cells in the neighbourhood is not important, simply the number present in each state. This is equivalent to a totalistic automaton.
4 Associative Memories

This chapter discusses how the local transition function of a cellular automaton may be considered to be a rule-based system. Associative memories are then outlined as a way to implement these rule-based systems.

In this chapter ⊗ is used to denote bitwise logical ORing of two matrices of equivalent size. ⊗ is used to denote the outer product of two matrices

4.1 Introduction

Associative memories make an association between pairs of patterns or vectors, an input $I$, and an output $O$. This association may be perfect, or recall of a pattern given the input $I$ may be imperfect, depending on the exact form of the associative memories, the number of associations already learned, etc. The patterns or vectors may be a coded representation for more concrete data items. For discussion of associative memories see [46][47][48]

4.2 Correlation Matrix Memories

Correlation matrix memories are matrices that hold the correlations between sets of input and output vectors.

If there is a set of $p$ input vectors $I, \{i_1, i_2, \ldots, i_p\}$ and $q$ output vectors $O, \{o_1, o_2, \ldots, o_q\}$, then the correlation matrix memory $M$ will hold $r$ associations, each between an input vector and an output vector.

For further detailed discussion of correlation matrix memories see Willshaw [46].

4.2.1 Binary Correlation Matrix Memories

A subclass of correlation matrix memories is the binary correlation matrix memory. A binary correlation matrix memory comprises a matrix, the elements of which may only take binary values (0 or 1). The matrix makes associations between binary vectors (the elements of which may only take values 0 or 1). When a binary vector $I$ is associated with a binary vector $O$ in a memory $M$, the value of $M_{ij}$ is set to $I_i$ anded with $O_j$ ored with the existing value of $M_{ij}$, i.e. $M_{ij} = (I_i \cap O_j) \cup M_{ij}$, or more generally $M' = M \oplus (I \otimes J)$.
On recall a pattern $J$ is applied to the input. For each element of $J$ set to 1, sums of the columns of $M$ are performed. These values are stored in a raw result vector $R$, that contains integer values from 0 to $n$, where $n$ is the number of bits set in the input pattern $J$.

$$R_n = \sum_{i=0}^{n} J_i M_{ni}$$ \hspace{1cm} (7)

To obtain a binary vector, $R$ must then be thresholded. There are a number of possible ways (detailed later) to threshold the raw vector to get a binary result, depending on the parameters of the input and output vector sets, the expected loss of information during the teaching cycle and so on.

4.2.2 Collisions between inputs, outputs, and already set bits.

Such losses of information may occur when patterns in the input or output sets are not orthogonal compared to other elements in their set. These events are collisions.
Collisions are an inevitable consequence of using non-orthogonal patterns, but non-orthogonal patterns must be used if attempts are made to squash the correlation matrix memory. A squashed memory using non-orthogonal pattern can be considerably smaller. Given that a certain number of collisions are inevitable if there is storage-saving squashing, for optimal recall collisions should be avoided as much as possible given the constraints.

It is possible for patterns to be non-orthogonal and still result in collision-free matrices after teaching, for example the input patterns in teaching phases 1, 2 and 3 in the example above do not form a mutually orthogonal set, but if associated with the corresponding output patterns there will be no collisions. The only way to guarantee that there will be no collisions is for either or both sets of patterns to have the property of mutual orthogonality.

Saturation of the matrix memory occurs as more locations in the correlation are set to 1. The likelihood of collisions increases as the saturation of the matrix memory increases.

There are a number of parameters which affect how quickly the memory becomes saturated as teaching progresses, one of which is the mutual orthogonality of the sets of inputs and outputs which are associated.

4.2.3 Binary vectors and orthogonality

A binary vector is a vector which may have elements only 0 or 1. A vector in $n$ dimensions is a point on the unit $n$-sphere, but has quantised positions on the surface of that sphere. A binary code of $n$ bits is presumed to be equivalent to a binary vector in $n$ dimensions.

Two vectors are orthogonal if their dot product is zero.

If there are two vectors, $\mathbf{a} = (a_1, a_2, a_3)$ and $\mathbf{b} = (b_1, b_2, b_3)$ then

$$\mathbf{a} \cdot \mathbf{b} = a_1 b_1 + a_2 b_2 + a_3 b_3$$  \hspace{1cm} (8)
Making the equivalence between binary vectors and binary codes from §4.2.3, a set of vectors, $V$, in which there are $r$ vectors, is mutually orthogonal if

$$\sum_{i=1, j=i+1, j \neq i}^{r} v_i \cdot v_j = O$$  \hspace{1cm} (9)$$

If the absolute value of $O$ is low compared to the maximum value then it is said the set of vectors are nearly orthogonal. For orthogonality, $O = 0$.

When dealing with associative memories we typically deal with sets of codes that are all of the same size, and have a fixed number of bits set to 1 (i.e. all have a fixed Hamming weight). There exists a set of codes that have a fixed number of bits, $N$, and fixed Hamming weight, $k$, which we will term $C_{N,k}$. The value of $O$ for such a set can then be calculated using equation 9 above. The near orthogonality for a set of vectors, $V_{N,k}$ with $r$ members that are chosen arbitrarily from $C_{N,k}$ will be $O_{V_r}$, and $O_V$ can vary depending on how the vectors are chosen. Over all samples of $V$ from $C_{N,k}$ we can calculate an average value $\bar{O}_{V_r}$, to which a specific value of $O_{V_r}$ for a particular set of codes $V_r$ should be compared.

We might also be interested not only in the mutual orthogonality, but also the evenness of the distribution of bits set across the width of the codes. In practice this distribution is related to the orthogonality (if it is low, then the distribution will tend to be fairly even), but the evenness can be an easier measure to calculate, and can refer more directly to the way the binary correlation matrix memories may be saturated.

The bit distribution vector, $B$, calculates the total number of bits set to one for each position, divided by the maximum number of bits that could be set, i.e. the number of codes. For a set of $r$ codes with $k$ bits set in $N$ bits, the bit distribution is a vector containing the values for the sums for each element of each vector in the set of vectors. I.e. for a set of $r \{v_1, v_2, \ldots v_r\}$ vectors of width $N$

$$B = \left( \frac{\sum_{i=0}^{r} v_{i1}}{r}, \frac{\sum_{i=0}^{r} v_{i2}}{r}, \ldots, \frac{\sum_{i=0}^{r} v_{ir}}{r} \right)$$  \hspace{1cm} (10)$$

For example if we have 3 codes of size 3, $(1, 1, 0)$, $(1, 0, 1)$, and $(0, 1, 1)$, we could calculate $B$ to be $(\frac{1 + 1 + 0}{3}, \frac{1 + 0 + 1}{3}, \frac{0 + 1 + 1}{3})$, i.e. $(2/3, 2/3, 2/3)$. 
4.3 ADAM

4.3.1 Introduction

In a traditional correlation matrix memory for associations between input vectors of size $m$ and output vectors of size $n$ the correlation matrix memory must be of size $mn$. If there are a large number of patterns to be associated then $mn$ can be large, and $m$ and $n$ excessively so. In ADAM the associations are mediated by a third binary vector, the class pattern, $C$, which may be set to be smaller than $m$ and $n$. The class code is taken from a set of such class codes, all of which have the same number of bits set to 1 (i.e. the same Hamming weight).

If the size of $C$ is $c$ then the total storage required is $mp + pn$. i.e we are saving storage space if

$$c < \frac{mn}{m+n}$$

(11)

FIGURE 9. ADAM teach and recall

For further papers on ADAM see [31][32][35][36][37].

4.3.2 Class Codes

The associations stored in the correlation matrix are between pairs of inputs and outputs, such a pair being $I$ and $O$, mediated via a class code $C$ (one for each association). If the size of the class code, $c$, is set such that storage is being reduced (equa-
tion 11) then to store a large number of associations then the Hamming weight \((k)\) of the code will need to be greater than 1.

For example, if there are six inputs with one bit set to \(k\) in each, then if the size of \(C\) is required to be 5 then at least two bits must be set to one to allow the required number of binary codes to be created. Normally patterns with higher bit densities (the ratio of Hamming weight to size of code, i.e. \(k/c\)) lead to faster saturation (see §4.2.2) of the correlation matrix memories and consequent loss of information. Using thresholding and picking the class codes carefully performance can still be acceptable. Having a fixed number of bits set in the class code makes thresholding simpler as the expected number of bits set to \(k\) in the code after thresholding is known and can be accounted for. This allows more reliable association of sets of inputs which have different number of bits set to \(k\) with output patterns which have different numbers of bit set to \(k\).

If storage is not an issue then the class codes may be mutually orthogonal. This guarantees perfect recall irrespective of the parameters of the input or output vectors.

### 4.3.3 Generating Class Codes

Class codes can be picked at random from the set of possible codes with a specified bit width and number of bits set to 1. Such a set of codes will tend not to be the set of codes (except by pure chance) that is the most nearly orthogonal, and so will not produce optimal recall behaviour. Other methods can be used which create a set of codes with improved orthogonality properties, namely the method of Baum, Wilczek and Moody [68].

### 4.3.4 Thresholding Class Codes

The class codes used during the teaching phase will all have a set Hamming weight, \(k\). Ideally during the testing phase, a class code with Hamming weight \(k\) will be recovered from the response of the first correlation matrix memory, \(M_1\).

There are three main ways to threshold a raw class vector to derive an output vector:

1. **Threshold at Maximum**

For example, if the raw class is \((5, 3, 2, 1, 0)\), then the thresholded class is \((1, 0, 0, 0, 0)\). If the raw class is \((5, 5, 2, 1, 0)\) then the thresholded class is \((1, 1, 0, 0, 0)\). The elements with the maximal response have the corresponding elements in the thresholded class set to 1. There is no attempt to pick a code with a specified number of bits set to 1.

2. **Hard threshold**

In this case all elements above a certain level are set to 1. Thresholding \((5, 3, 2, 1, 0)\) from above at a level of 5 gives a result of \((1, 0, 0, 0, 0)\), also as above. Thresholding at a level of 2 would, however, give \((1, 1, 1, 0, 0)\). Again there is no attempt to pick a code with a specified number of bits set to 1.
33

(3) L-Max Threshold

In this case the threshold is set such that the number of bits set to 1 in the thresholded code at least \( k \), but as close to \( k \) as possible. It is not possible to guarantee that exactly \( k \) elements in the thresholded code will be set to 1.

If the class code should have 2 bits set and the raw class elements are (0, 1, 2, 1, 2, 3) then we can see that it is not possible to threshold such that only two bits are set in the result. We can threshold at a level of three, and obtain the binary \( 000001 \), or at two and obtain \( 001011 \). In practice if L-max returns too many bits then the confidence we can attach to elements of the class code can be determined (in this case 000001 is confident, so we expect that the correct result will be 001001 or 000101, but not 001010). L-max always returns at least enough points set, iff the raw result vector contains at least \( L \) non-zero elements.

Continuing the example, 000001 can be applied to the second correlation matrix and its contribution to the final result determined, and so the likely outputs determined more easily as the contribution of 000001 alone can be seen in the final output.

4.3.5 Thresholding and partial matching

Partial matching occurs when an input pattern not matching one of the associated patterns is applied. There may be more or fewer bits than expected. If there are fewer bits then the response will be less than expected, but thresholding may be applied in such a way that the class code for the nearest response is still recovered, if a partial match is desired.

Partial matching behaviour makes ADAM tolerant of faults in its inputs. This can be important if the computing elements (the cells in the automaton) are unreliable components as the behaviour of ADAM, and the local behaviour of the cell, may be to recover from these errors.

Consider the example:

**FIGURE 10. Various thresholding algorithms for ADAM class codes**

- Teaching
- Recall on same pattern
- L-max
- Threshold at max
- Threshold at number set in input
- Recall on partial pattern with various thresholds
Thresholding at the number of bits set in the input excludes anything other than an exact match.

Thresholding at the maximum response seen in the raw class response picks class code that matches most of the bits set, but will not give all the possible partial matches. It is most useful when there are additional bits present in the input, to indicate the main class code element(s) which is (are) being matched.

**FIGURE 11. Partial matching for ADAM**

L-max thresholding will always attempt to produce a class code separator with \( l \) or more bits set, and so will always produce a partial match, which contains at least some valid class codes that are the class codes for similar patterns. It may produce a series of superimposed codes for a number of near matches.

**FIGURE 12. Partial matching for ADAM**

When using L-max thresholding the superimposed codes must be unravelled using knowledge of the codes used during the teaching phase. If the partial match is against inputs with too many bits set the result of a thresholding at the maximum response can be useful. For example in Figure 12 above the L-max threshold tells us that the possible codes superimpose to give 1010110000, but the results of a threshold at maximum indicate that the codes for partial match are most likely to contain 1010110000, so we can discount a code 0010110000 being in the set of superimposed codes, even if it contains the correct number of bits set (3).

**4.3.6 Class Codes and Input Ordering**

When the class codes used during the association process are generated this is done without reference to which class code will be used for which pair of associations. The emphasis is normally on producing a set of codes that have good general properties, i.e. overall the memory does not saturate too rapidly with pairs of vectors to be associated and a
class code chosen at random from the set of possible codes. This approach has the advantage that the computational cost during the teaching phase is not too great.

However, the computational time of the teaching phase may be negligible compared to the computational time of running a large, complex automaton. It may be worth obtaining even a small speed up in the execution of the automaton if, over a large number of cells and iterations, it leads to a net saving, even if the teaching costs are considerably increased.[70][68][59]

If the class codes can be matched to pairs to be associated then it may be possible to reduce the saturation of the memory. Reducing saturation in itself can lead to a reduction in the requirements for summing and thresholding, but even more so if the same level of saturation can be tolerated, and by careful choice of triples of input, class code, and output, the size of the class code can be reduced. Alternatively, accuracy of recall could be improved.

### 4.3.7 Recall time

As a first approximation recall time is proportional to the sizes of the input, output and class codes, and the number of bits set to one in each. Thus if class codes are fixed, the number of bits set to one is fixed, then the recall time is roughly proportional to the class code size.

If we have two patterns which are associated via a simple correlation matrix memory, excluding any thresholding on the final output, with the input pattern being of size $N_I$ bits, with Hamming weight $k_I$, and the output being of size $N_O$ bits, with $k_O$ bits set to 1, with $A$ previous associations already present in the matrix, then recall time is proportional to

$$
\frac{A k_I^2 k_O}{N_I}
$$

(12)

**Graph 1.** Recall time as a function of class size and number of bits set for 100 patterns of size 10 with 10% bit density
Graph 2. Recall time as a function on input pattern parameters with a class size of 100, with 2 bits set

4.3.8 Tupling

Tupling is a technique which allows the a set of inputs to a correlation matrix memory to be transformed into a second set of inputs to the memory with a known number of bits set, based on a mapping function. The various forms of tupling indicate different ways of determining the mapping function. The mapping function may be linear, random (with or without replacement), or a number of other functions.

A tuple has a number of inputs, which are placed on the input space according to the mapping function. The values of these inputs then determine the state of the tuple, which is determined on the value of the bits determined as a binary number. This state is then represented by a single binary bit in the position corresponding to that state. The tupled output is the concatenation of all these binary patterns. The number of bits set in the output is fixed since even tuple state 0 is represented by a 1 in the 0th position of the binary output of that tuple.

For example, the pattern 100100100100 is to undergo linear tupling, with tuples of size 4:
The beneficial effects of tupling are twofold:

1. If the input patterns are dense, or of variable density, then the output pattern from the tupling stage will be of guaranteed density, and if the tuples are few and large, less dense.

2. Due to the decrease in density, and the positioning of bits in the patterns, the mutual orthogonality of a set of patterns after tupling will tend to be better than prior to tupling, unless the original set of patterns includes many very sparse patterns. The orthogonality parameters increase as the size of the tuples is increased.

Tupling represents a simple transformation which does not require detailed knowledge of the whole set of input patterns to be effective. However, tupling increases the size of the input patterns.

### 4.3.9 ADAM Properties Summary

1. ADAM allows us to compress the size of the required correlation matrix memories as compared to a comparable single stage correlation matrix memory, if the class codes are compressed. This allows larger sets of associations to be learned.

2. Compressing class codes can lead to errors.

3. The class code stage allows us to determine if there are errors appearing the first correlation matrix memory and for the confidence of recall in the second correlation matrix memory to be determined, allowing potentially more accurate recall than a correlation matrix with a lossy stage.

4. The partial matching capability of ADAM may allow fault-tolerant behaviour if an input to ADAM fails and there are fewer bits on the input than expected, depending on the way that the raw response is thresholded.
5 Basic Architecture

This chapter outlines an implementation of a cellular automaton comprising a number of state machines, each constructed from binary associative memories.

5.1 Non-associative memory implementations of local transition functions

There are various ways to implement cellular automata using computers. Typically implementations tend to be rather automaton specific, the code being applicable only to that automaton. This allows the code to be very efficient for that automaton. This is important if it is to be used as a tool for simulation, e.g. for lattice gases and so on, but development time is comparatively high. Recently more general ways to implement cellular automata have been developed, either as interfaces to parallel hardware, or as a higher level language producing efficient run-time code.

CAM 6 and 8 were developed in the early 1980s by Toffoli and Margolus [26][62] to allow easier use of a parallel hardware system. CAM 6 and 8 have been highly successful, partly due to the low cost of the supporting hardware. The syntax is, however, relatively obscure and based, to a certain extent, on binary arithmetic. Whilst there are simulators to allow software simulations of the hardware to be used, it is tied into the CAM board. It has been very successful in terms of speed.

Cellang [67] and CELIP [27] have taken the approach of offering a high level programming language that also allows options for parallelisation at compile time. These programming languages allow cellular automata to be coded flexibly in a compiled language.

5.2 Local transition functions as rule-based systems

Often cellular automata are viewed strictly in terms of functions at the local level, e.g. totalistic automata, but an alternative is to consider them in terms of rule-based systems. Thus each cell has a set of rules with the nearest neighbourhood and the current state as inputs that lead to an answer that is the new state of the system. Rule-based systems can be modelled using associative memories: the preconditions for a rule are associated in some way to the result of the rule.

Using associative memories allows us to consider all the techniques for associative memories that allow efficient use of storage, speed, etc.

One problem with using associative memories is that accurate recall may not be guaranteed. Von Neumann [8] (also Gacs [17][18]) suggested that accurate computation might be carried out with a system, i.e. a cellular automaton, comprising unreliable components. (The human brain is made of unreliable components yet is capable of computation.)
Only some local transitions have behaviour that allows an automaton to recover from errors produced by unreliable components.

Langton and Wolfram [51][64] suggest that the most fruitful areas for universal computation may be those areas close to chaotic behaviour. The following chapters will concentrate on finding an associative memory implementation that attempts to produce components as reliable as possible.

### 5.3 State machines and connections

The basic requirement is for some state machine, the input data to which is the current state and the states of cells in the neighbourhood of the cell, and the output of which will be the new state of the cell. In theory any simple associative machine which has the required number of inputs should be sufficient.

**FIGURE 14. Simple architecture (here a simple 1D automaton is shown for clarity)**

5.4 Representation of states

The states of the cells of the system are represented by binary patterns. Since the binary associative memories will be used, and the final association will be with a binary pattern the states of the cells are represented by binary patterns with a fixed number of bits set to 1. Using a fixed Hamming weight simplifies the application of L-max thresholding to the raw output to recover the final output.
5.5 Representation of neighbourhood state

The neighbourhood state is the collection of the states of the cells in the neighbourhood of a cell. The neighbourhood state will also be represented by a binary code, derived in some way from the binary patterns representing the cells in the neighbourhood, and their positions. This binary code is the neighbourhood code. The derivation of the neighbourhood code from the neighbourhood states should embody the inherent degeneracy of the local transition function of the cell (anisotropic, isotropic, or isomorphic). This can be achieved by a many-to-one or one-to-one mapping between the neighbourhood state and the neighbourhood code that is used to represent it.

**FIGURE 15. State machines and associative memories**

For example, in Conway’s Life [52] all neighbourhoods which lead to a particular change of state of a cell would be grouped together under one neighbourhood code. The local transition function for Conway’s Life treats all neighbourhoods with 0, 1, 4, 5, 6, 7, or 8 elements set to 1 as identical, so the first associative memory will perform a mapping from the preprocessed neighbourhood states for each of these to a single neighbourhood state code. For Conway’s Life there need be as few as three neighbourhood state codes.

If there are \( n \) elements in the neighbourhood, and the alphabet is of size \( w \), then the number of possible neighbourhood states is \( w^n \), but if there is degeneracy then the preprocessing will reduce this number. These will then be mapped to produce a number of neighbourhood state codes. At most there are \( w^n \) neighbourhood state codes. In the example of Conway’s Life, nine associations would be needed.
The neighbourhood code and current state are then used together to derive the new state for the cell. This is achieved via an associative memory which has been trained on the mapping of pairs of neighbourhood codes and cell states, to new states. The pair of current cell state and neighbourhood code is represented by computing a order two tensor from the multiplication of the neighbourhood code and current cell state, this forming the kernel state.

Example:

In the following example the automaton has two states, A and B, represented by the patterns 01 and 10.

**FIGURE 16. Determining new cell state in a 1D automaton**

Neighbourhood states are A and B. The binary representation of this is 0110.

This is associated in the first associative memory with a neighbourhood state code, chosen from some set of neighbourhood state codes. In this case let us assume the neighbourhood state code is 111000. 0110 is associated with 111000 in the first memory.

The second memory associates two inputs: neighbourhood state code and current state, with final state. The two inputs are dealt with by forming a order 2 tensor

\[
\begin{bmatrix}
1 & 0 \\
1 & 1 \\
1 & 0 \\
0 & 0 \\
0 & 0 \\
0 & 0
\end{bmatrix} \otimes
\begin{bmatrix}
1 & 0 \\
1 & 0 \\
1 & 0 \\
0 & 0 \\
0 & 0 \\
0 & 0
\end{bmatrix} =
\begin{bmatrix}
1 & 0 \\
1 & 0 \\
1 & 0 \\
0 & 0 \\
0 & 0 \\
0 & 0
\end{bmatrix}
\]

A vector is formed by concatenating the rows of the order 2 tensor, and this is associated with the required new state of the cell.

If the current state is \( \psi(t) \), represented by a binary pattern \( A \), and the neighbourhood state is \( h(\alpha) \), represented by neighbourhood code \( B \), and the new state \( \psi^{t+1}(\alpha) \) is represented by binary code \( C \) then the association is

\[ A \otimes B \rightarrow C \]  (13)

If there are \( N \) neighbourhood state codes then the associative memory requires \( Nw \) associations. For the example of Conway’s Life above, six associations would be needed.
The use of an intermediate kernel state code that is formed through this tensor multiplication can be justified by considering the following:

Let the size of a token representing a state be $s$. Let there be $n$ elements in the neighbourhood.

If only a single association stage were used it would be required that it associate the whole kernel state with the new state that the cell should take. This would require $ss^n$, i.e. $s^{n+1}$ associations. This is potentially $s^n$ associations fewer than the system using two associative memories. Using one associative memory necessarily requires that the input size of the associative memory be larger than the first associative memory of the two associative memory system.

When using only one associative memory the neighbourhood state part of the kernel state is presented to the memory $s$ times during teaching. This is unnecessary repetition of this pattern, which can lead to problems with the set of inputs representing all kernel states being not mutually orthogonal, which can lead to premature saturation of the matrix memory.

5.5.1 ADAM

ADAM is the preference for the first associative memory. The inputs to the first associative memory are the preprocessed neighbourhood states, and the mutual orthogonality of this set of inputs is determined by the form of preprocessing. If a simple correlation matrix memory is used then if there are few neighbourhood codes (i.e. the preprocessed neighbourhood states are divided up into a few classifications that determine the behaviour given the current state of the cell), the associative memory is more prone to saturation should the inputs not be mutually orthogonal than would be the case if an ADAM is used. ADAM allows the inputs to be associated to a set of intermediate codes, the class codes, the mutually orthogonality of which can be more explicitly controlled, and then these are associated with the neighbourhood states. In the case of a few neighbourhood codes the two correlation matrices will be larger than a single correlation matrix memory, but this larger size reduces the saturation along with the control of mutual orthogonality of the intermediate class codes and allows improved recall over a single correlation matrix memory.

FIGURE 17. Advantage of using ADAM when there are few neighbourhood codes

![FIGURE 17. Advantage of using ADAM when there are few neighbourhood codes](single cmm ADAM (less saturated cmms))
As was indicated in §4.3.7, the time for recall for any of the subsystems is roughly proportional to the class code size, all other factors being equal, so reducing the class size by exploiting degeneracy results in improved recall times.

5.6 The spreader

5.6.1 Description

It has been suggested that this basic architecture can be modified by introducing a spreader system. The spreader is essentially another correlation matrix memory working in the reverse sense to the state machine in that it takes a single input, the state of the cell, and produces a number of outputs equal to the number of elements in the neighbourhood of that cell. Essentially it performs a filtering operation on the state of the cell such that cells in the neighbourhood do not see the state of that cell directly but only after the action of the spreader. The output of the spreader in each direction may or may not be different.

The spreader can also act as a lossy channel: many input states to the spreader may become a single state from the output of the spreader, and the many-to-one mapping allows some information to be disregarded.

FIGURE 18. Architecture with spreader

Note that the current state of the cell passes unchanged through to its own state machine

5.6.2 Heterogeneity and time variations

If the system does not vary in time, even if there is a mix of cells with different local transition functions there is, in theory, a transform that takes the transition function that is split across state machine and spreader and forms a new automaton that has simple state machines, with no need for spreader functions. For the transformed automaton the local transition functions will be different. This transform is paramaterised by all the local transition functions and the spatial distribution of these local transition functions within the automaton. The form of this transform may not be totally obvious, in general, since it requires knowledge of all states and transitions, and will require a lot of power to compute. In fact it would be simplest in most cases to design the local transition function without a spreader stage.
However, if the heterogeneity is time varying (e.g. in rule competition for systems attempting to solve the prisoners dilemma, and other competing systems [61]) such a transform is only possible at each individual time step. For discussions of heterogeneous cellular automata see [20][21][22].

5.6.3 Why use a spreader?

The spreader can help conceptually in that it may allow the problem to be partitioned in a more intuitive way. If a system is to have a directional bias then the spreader can be used to pass on the state of the cell unchanged in all but one direction. The actual state of the cell does not change, but it appears different as viewed from neighbouring cells. A spreader which has a directional behaviour is equivalent to the cells that have the cell in their neighbourhoods having directional local transition functions. In a sense the spreader of a cell is an input buffer for the inputs of the cells in its neighbourhood. In a complex system the time taken to teach the main associative memory that forms the state machine may be large, and this ability to introduce directionality in the local transition functions via the spreader, which may be a small memory that has a small number of one-to-many associations trained. The number of associations that the spreader must learn is at most the number of states in the alphabet of the automaton. One association in the spreader associative memory affects potentially many associations in the state memory of a cell which has it in its neighbourhood. (If there are $w$ states, and the neighbourhood size is $n$, then there can be up to $w^n$ states in the main associative memory that forms the state machine).

However, it was decided that the additional complexity of the spreader was unnecessary for the analysis pursued in this thesis.
6 Anisotropic matching system

This chapter describes how to match an exact configuration of states in the neighbourhood of a cell.

6.1 Representation of the neighbourhood state

In the anisotropic case we want to match to the exact states in exactly specified positions in the neighbourhood. The simplest way to achieve this is to take the tokens from the alphabet and simply concatenate them (given some fixed and consistent order) and present this longer pattern as the input to the correlation matrix memory for both teaching and testing.

FIGURE 19. Inputs to a cell

In Fig. 19 we have the state of the cell $\alpha$ represented by $e$, $\alpha+\delta_1$ by $a$, and so on, $a$, $b$, $c$, $d$, and $e$ are tokens represented by binary patterns.

The input to the associative memory state machine for teach and test is $abcde$.

If we represent $a$, $b$, $c$, $d$, and $e$ with binary patterns thus:

<table>
<thead>
<tr>
<th>Code</th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>$d$</th>
<th>$e$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>10100</td>
<td>01010</td>
<td>11000</td>
<td>00011</td>
<td>10001</td>
</tr>
</tbody>
</table>

TABLE 1. Codes for a simple 4 state automaton

The pattern 10100010111000000111 is the input to the first associative memory. This will be associated with a neighbourhood state code, say 1000101
The kernel state code is then formed by the tensor multiplication of the neighbourhood state code, and the current state. In the case above a order 2 tensor, $K$, is formed from $1000101$ and $10001$

The $K$ is then associated with the resultant state for the local transition.

### 6.2 Ill-conditioned inputs

To form the neighbourhood state representation comparatively short tokens are concatenated. Each of these tokens is constrained to have a fixed number of bits. This means that the variation in the neighbourhood state vectors is very limited due to this restriction. There will be a fairly small number of output states. The combination may have the effect of overloading the memories of the ADAM as successive associations are taught as there are many similar inputs and outputs. The associative memories saturate sooner than would otherwise be the case, given inputs and output of the same size and number of bits set to one, and the same class parameters, but with patterns with better mutual orthogonality properties.

If class code parameters are such that class codes are not orthogonal then the average accuracy of recall of the ADAM is below that which would otherwise be the case were the inputs and outputs simply random patterns chosen with the required number of bits set to one.

For example if we had a 4-neighbourhood with one bit set to one in the tokens representing the states, then there would be four bits set in the neighbourhood state patterns, but only a limited number of relatively similar patterns are allowed due to the neighbourhood state pattern being the concatenation of a number of small blocks, each with a fixed number of bits set to one. If the tokens were of size $s$ then a set of patterns with simply four bits in size $4s$ could have better mutual orthogonality properties, if chosen carefully. This freedom is not available when the tokens are used and concatenated. However, we can choose the class codes of the ADAM, and the neighbourhood codes themselves to offset this to a certain extent.

The following example of the problem considers a binary system, the patterns being represented by $01$ and $10$. $01$ and $10$ are used rather than $0$ and $1$, since $01$ and $10$ have a constant number of bits set which aids thresholding to determine the final state. (A threshold function will attempt to create a code with at least some bits set, thus $0$ cannot be used as a state code).
Consider transitions for the automaton in Fig. 20. The unique binary patterns on the input of the neighbourhood state associative memory are:

- 1010
- 0110
- 1001
- 0101

These patterns are not mutually orthogonal, which may lead to premature saturation of the first correlation matrix memory.

A solution to this problem is to instigate input tupling after concatenation of the tokens, but before presentation to the association memory.

Tupling generally produces an output that is more nearly orthogonal than the input and has a known number of bits set. Typically tupling is applied to inputs which may have a variable number of bits set, but in this case the number of bits set in the input is already known and it is the greater orthogonality that is desired.

Take an example from above using linear tupling with, say, a size of two:

If the input pattern is 1010, then the first tuple takes the first two binary digits, i.e. 10, and the tuple state is then $1 \times 2 + 0 \times 1 = 2$, which gives a tuple state of 2. The second tuple has a state also of $1 \times 2 + 0 \times 1 = 2$. This gives a tupled result of {0100, 0100}, or 01000100. Thus 1010 → 01000100
In this case linear tupling produces an orthogonal set of codes after tupling. However, linear tupling does not produce as much orthogonality as is theoretically possible since the tokens that make up the input pattern have a fixed number of bits set. An input pattern, prior to tupling may be 1010, but cannot be 1100, since the pattern is made up of the concatenation of combinations of 01 and 10. Thus the number of possible tuple states is limited. Ideally we would wish to have the maximum possible number of output states since this maximises the improvement in orthogonality without increasing the size of the output pattern from the tupling stage. To achieve this a tuple map is created that maps elements of the input to random elements in random tuples, with this tuple map being preserved for all subsequent input processing. This is random tupling.

For example, consider a particular instance of random tupling without replacement with a tuple size of 2 on a 4 bit code. Note that the inputs to tuple 1 and tuple 2 are now not taken in order from the 4 bit code. I.e. tuple 1, element 1 is not constrained to be the first bit of the code.

**FIGURE 21. Random tupling**

```
untupled pattern

1  0  1  0

0  1  0  1

Tuple 1  Tuple 2

output states     1     1

tuple outputs     0010   0010

output after concatenation  00100010
```

Non linear tupling allows more possible tuple states.
Given the tuple map above the example inputs would be transformed to these (which are more orthogonal than those for linear tupling)

<table>
<thead>
<tr>
<th>untupled codes</th>
<th>tuple state</th>
<th>tupled output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>binary</td>
<td>decimal</td>
</tr>
<tr>
<td></td>
<td>tuple 1</td>
<td>tuple 2</td>
</tr>
<tr>
<td>1010</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>0110</td>
<td>11</td>
<td>00</td>
</tr>
<tr>
<td>1001</td>
<td>00</td>
<td>11</td>
</tr>
<tr>
<td>0101</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>column totals</td>
<td>2222</td>
<td></td>
</tr>
</tbody>
</table>

TABLE 3. Random tupling of neighbourhood state codes

In fact given the scheme above a completely orthogonal set of tupled patterns is achieved.

The most suitable tuple map will be a function of the input tokens and tuple size, and is not generally simple, although it may be in certain cases.

### 6.3 Conclusion

This is a simple system. Neighbourhood state patterns are created by simply concatenating the patterns representing the states of the cells on the neighbourhood. It was shown how forming neighbourhood state patterns in this way can lead to ill conditioning in the set of neighbourhood state patterns due to the repetition of a small set of cell states in the same positions in the neighbourhood over the set of all possible neighbourhoods that must be recognised by the system. When these ill-conditioned neighbourhood state patterns must be used as inputs to the associative memories recall becomes difficult due to premature saturation of the memory, unless a large memory and/or space class codes are used, unless the inputs are preprocessed to make them better conditioned. Tupling is a form of preprocessing that achieves this aim. The time cost of additional processing of the inputs needs to be set against the time and storage costs of using a larger associative memory or larger, more nearly orthogonal class codes.
7 Isotropic symmetry solution: tensors

This chapter first describes a system for rotation-invariant recognition of patterns on the neighbourhood of a cell of an automaton, but then goes on to indicate that this has problems due to potentially large storage requirements. It then goes on to describe various attempts to develop a way of recognising rotated patterns, that overcomes these problems.

7.1 Aim

We are looking for a way to improve recall. In general as teaching may be done off-line we will not be concerned here with the computational requirements for teaching the memories that will later be used in recall.

Recall can be improved both in terms of accuracy of recall and speed of recall, and there is a trade-off between the two.

Recall time was discussed in §4, and equation 12.

7.2 Bound Strings to Represent Neighbourhood States

\( \oplus \) is used to represent a binary superimposition equivalent to a logical ORing operation.

\( \otimes \) is used to represent the Kronecker product of two tensors.

Note that \( \otimes \) distributes over \( \oplus \).

The superposition of binary vectors or tensors is a form of addition. If two order 2 tensors, \( A \) and \( B \), are superimposed to form a third tensor \( C \), then the elements are assigned such that \( C_{ij} = A_{ij} \oplus B_{ij} \) for every \( i \) and \( j \).

The papers by Smolensky [29][30] and from Queensland [28] discuss the use of the Kronecker product to model binding between variables. i.e. if \( a \) is to be linked with \( b \) then we create binary vectors for \( a \) and \( b \), \( v_a \) and \( v_b \), and represent the binding by the product \( v_a \otimes v_b \). This binding can be for an arbitrary number of elements. The binding process is non-commutative, i.e. \( v_a \otimes v_b \neq v_b \otimes v_a \). This allows us to bind variables together in a way that preserves information about their ordering.

In the case of a cellular automaton cell we can represent the relationship of a series of neighbourhood elements to another by binding the tokens representing the states together. I.e. if the neighbourhood contains state \( s_1 \) and neighbourhood element position \( p_1 \), and neighbourhood elements \( s_2 \) at neighbourhood element position \( p_2 \), and so on, and there are \( n \) elements in the neighbourhood then
\[ s_1 \otimes s_2 \otimes \ldots \otimes s_n = S_1 \]  

(14)

Uniquely represents the neighbourhood including its relationships as the order of association is important (i.e. it is a non-commutative operation).

If the neighbourhood were rotated then \( s_1 \) might now be at position \( p_2 \), \( s_2 \) at position \( p_3 \), \( s_n \) at position \( p_1 \), and so on. Thus the tensor representing this would be

\[ s_n \otimes s_1 \otimes \ldots \otimes s_{n-1} = S_2 \]  

(15)

This creates large input vectors.

If we have \( n_s \) states, and the token size is \( N_t \) with \( k_t \) bits set, and there are \( r \) neighbours then the basic anisotropic system requires a vector of size \( r N_t \) bits with \( r k_t \) bits set to 1.

For the system using vectors bound together in a tensor the tensor is of size \( (N_t)^r \) bits with \( (k_t)^r \) bits set to 1.

Although storage is larger by a factor of \( \frac{N_t^{r-1}}{r} \) for a fixed number of associations, and a fixed size of token with the same size of output pattern and same token parameters the system of bound tokens requires approximately \( \frac{k_t^{2(r-1)}}{N_t^{r-1}} \) times as much computational time for a recall, prior to thresholding, but if \( k_t \) is kept small this can be advantageous. However, this analysis neglects the time required actually to form the product of all the tokens in the first place.

We can then associate both \( S_1 \) and \( S_2 \) with the same output, i.e. the same neighbourhood code, which is a many-to-one mapping.

I.e. the associations are \( S_1 \rightarrow O \) and then \( S_2 \rightarrow O \), which is equivalent to \( (S_1 \oplus S_2) \rightarrow O \)

On recall simply \( S_n \) is applied, and the partial match capabilities used to extract \( O \). When the tokens representing the states are orthogonal this process is error free. Errors increase as the tokens become less nearly orthogonal, or if there are errors in the tokens, with some elements of the neighbourhood being in multiple states (see §13).

### 7.3 States and neighbourhood positions

The following approach uses a partial matching method, but is an attempt to reduce the size of the vectors required by reducing the dimensionality (order) of each array to 2 rather than \( r \).
FIGURE 22. States and neighbourhood positions

Here a tensor is formed between the token at each neighbourhood position, and a token representing the neighbourhood position \(v_1, v_2, v_3, v_4\) above. (Note: parentheses used to aid reading, since \(\otimes\) distributes over \(\oplus\).)

\[
(s_a \otimes v_1) \oplus (s_b \otimes v_2) \oplus (s_c \otimes v_3) \oplus (s_d \otimes v_4)
\]  

(16)

then represents the relationship of each of the elements with their position in their neighbourhood, rather than the position of elements relative to each other.

For example:

FIGURE 23. Rotationally equivalent neighbourhoods

by superimposing the tokens representing the position in the neighbourhood teaching, again recall being via a partial match mechanism, i.e.

\[
(s_a \otimes (v_1 \oplus v_2)) \oplus (s_b \otimes (v_2 \oplus v_3)) \oplus (s_c \otimes (v_3 \oplus v_4)) \oplus (s_d \otimes (v_4 \oplus v_1))
\]  

(17)

Would be the teaching vector.

However it can be seen that this is equivalent to associating both

\[
(s_a \otimes v_1) \oplus (s_b \otimes v_2) \oplus (s_c \otimes v_3) \oplus (s_d \otimes v_4)
\]  

(18)

and
with the same resulting pattern or output. If this is done for all rotations then the input vector can become saturated. I.e. \( s_a \) in any position would always vote for this output, even if none of \( s_b, s_c \) and \( s_d \) are simultaneously present. Therefore this attempt can be rejected.

7.4 Comparison of Using Bound Strings and Neighbourhood Positions

Empirical results are shown for the isotropic system using bound strings (as outlined in §7.2) and the system using the relationship of neighbourhood elements to their position in the neighbourhood (as outlined in §7.3). The automaton used for testing is Langton’s Self-Reproducing automaton [53], which is inherently isotropic.

The results show the percentage of transitions learned successfully for various sizes of the class size in the first ADAM, with the fixed size of neighbourhood code.

Graph 3. Percentage of transitions learned successfully

As can be seen from Graph 3, the method using the chaining the cell states together to form a vector (with which the relationship of one cell to the next is preserved) has far better performance in terms of accuracy. For this reason this is the system selected for isotropic matching. Since the accuracy is so much better the class size can be reduced compared to the method in §7.3. Speed of recall increases as the class size is reduced, and so the method from §7.2 has the potential for speed increases compared to the other method.
7.5 Conclusion

Using a system of tensored tokens allows us the most accurate recall statistics. This system will be used as the basis of recognising neighbourhoods in a rotation-invariant way.
8 Isomorphic System Using Occurrence Checking

This chapter describes how isomorphic input patterns on the neighbourhood of a cell might easily be recognised using occurrence checking. It then also describes a problem with the method due to ill-conditioned inputs, and a possible way around this.

8.1 Introduction

In a system which exhibits isomorphism the position and relative positions of cells in particular states is not important, and so the question becomes one of the number of cells in particular states. The problem becomes simply one of counting the number of cells in a particular state, and then representing that information to the associative memory. Ideally we seek a representation that allows some partial matching capability, as this allows a level of fault-tolerance, which is a desirable feature.

Here a particular method of representing counted states is presented, and the problems stemming from the use of this method with an associative memory explored.

8.2 Method of Filer and Austin.

In the method of Filer and Austin [60] the input to the associative memory is coded in such a way that the input space to an associative memory of slots equal in number to the number of neighbourhood elements. Tokens are then drawn from the neighbourhood, identified, and placed in the slot appropriate to the number of patterns of that type already seen, the pattern being ORed with any pattern already there.

Repeating patterns distributes the evidence of the number of occurrences of a pattern in a natural way. If there are \( q \) occurrences of pattern \( A \), then the evidence of there being 1, 2, ..., \( q \) occurrences are naturally coded into the input. In this way partial matching is enhanced as should an occurrence of pattern \( A \) is missed during counting then there will still be a match on all lower numbers present. This may trigger a spurious association should there be one trained for \( q-1 \) occurrences of \( A \), but in the absence of this lower order association for \( A \), then partial matching may still allow a match on the expected association for \( a \) occurrences of \( A \).

FIGURE 24. Steps of the occurrence checking process
For example if we had tokens A, B and C represented by patterns 100, 010, and 001, then the final outputs in each bin for this example would be

<table>
<thead>
<tr>
<th>Bin</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100 100 000 000</td>
</tr>
<tr>
<td>2</td>
<td>010 000 000 000</td>
</tr>
<tr>
<td>4</td>
<td>000 000 000 000</td>
</tr>
<tr>
<td>3</td>
<td>001 000 000 000</td>
</tr>
</tbody>
</table>

The bins must then be brought together so that the input to the associative memory can simultaneously process information pertaining to all of the symbol types, but this must be done in such a way as to preserve as much information without creating excessively large codes. To achieve this each bin is tagged with a vector. Tagging involves forming a order 2 tensor with a bin, \( b_r \), and a vector \( v_r \).

If the bins are \( b_1, b_2, b_3, b_4 \) and the vectors by which they are multiplied are \( v_1, v_2, v_3 \) and \( v_4 \) in this example then input \( I \) is given by

\[
I = (b_1 \otimes v_1) \oplus (b_2 \otimes v_2) \oplus \cdots \oplus (b_n \otimes v_n)
\]  

\((n \text{ is the neighbourhood size, 4 in the example of above}).\)

If \( v_1, v_2, v_3, \ldots, v_n \) are unique and orthogonal, then this is the equivalent of forming a matrix

\[
\begin{bmatrix}
  b_1 \\
  \vdots \\
  b_n
\end{bmatrix}
\]

which in the example above would be

\[
\begin{bmatrix}
  1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

But if \( v_1 = v_2 = \ldots = v_n \) then we have the equivalent of the form \([b_1 \oplus b_2 \oplus \cdots \oplus b_n]\) which in the example would be

\[
\begin{bmatrix}
  1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

The latter uses an \( n \)th of the storage. However if token codes are not orthogonal the number of bits set in the input will vary as bits set to one overlap in the columns. In this case matching occurs due to the partial match capability of ADAM.

### 8.3 Ill conditioned inputs

As can be seen in the example above, the left hand columns of the input vector or matrix tend to be used most, since given a random selection of \( n \) from a set of patterns, subsets with only a few of each type of pattern are likely (there is only one selection AAAA, for example). The fact that the bits set to one in the inputs tend to cluster around a few areas means that the inputs become ill conditioned - they are all fairly similar, with little variation. This leads to faster satu-
ration of the correlation matrix memories than would otherwise be expected since areas of the matrix will be almost unused, except for certain parts of the coding of rare events such as a group of patterns, in the above example, of AAAA. The set of inputs for a typical selection of transitions that we might want the automaton to deal with will tend to be ill-conditioned, i.e. not nearly mutually orthogonal.

We can look at the nearness of the orthogonality by looking at the Hamming distance between codes. The Hamming distance is the number of bits that must be flipped to turn one code into another. For example the Hamming distance between \texttt{0101} and \texttt{0110} is 2. The Hamming distance characteristics of a set of typical inputs for an automaton using occurrence checking would be roughly of the form plotted in Fig. 25.

**FIGURE 25.** Idealised graph of Hamming distances of set of ill-conditioned inputs

However, to avoid saturation in the memory we would want a graph that looks more like Fig. 26.

**FIGURE 26.** Idealised of graph of desired Hamming distances for well-conditioned inputs.

i.e. the distance between codes is enhanced as much as possible, to allow efficient use of the correlation matrix memory, without recourse to matching class code separators to inputs to obtain maximum efficiency.

To achieve this we need a method to recode the input set in such a way that the new set is more mutually orthogonal, and has the bits set to 1 distributed fairly evenly across the code when all codes are considered. If we have \( M \) codes of size \( N \) with \( k \) bits set, then if we average the number of bits set in the \( i \)th element of a code, then the average should tend to \( Mk/N \).
Recoding may affect the partial match capability. The recoding should ideally maximise Hamming distance whilst retaining the relationship between codes (topological ordering). Codes which were similar should remain more similar than codes that were dissimilar, but the dissimilarity of similar codes should be enhanced within that constraint. This is equivalent to CMAC coding.

Possible candidates for recoding are:

1. Direct mapping between set and another, more orthogonal set.
2. CMAC coding.
3. Tupling.

Overlap is used as a measure of the orthogonality properties of the codes. For an explanation of this see §4.2.3.

8.3.1 Mapping between code sets

Mapping between code sets can be rejected as the number of possible codes for a given value of \( N \) and \( k \) may be large, and a hashing function between the two would be difficult to find. If such a hashing could be found then it could be used to replace the associative memory in its entirety, and simply map from input to final new state required.

8.3.2 CMAC coding

CMAC (Cerebellar Model of Articulation Control) coding is a form of coding that retains information about the topological ordering of the codes. See Kolcz and Allinson [71]. However, this was not investigated due to lack of time.

8.3.3 Tupling

Tupling should allow the orthogonality of the set of patterns to be increased. Since the data is clustered in certain areas then linear tupling will also tend to result in a limited variation in the codes as a large number of tuples will have all zero inputs, and so there will still be a lack of variation and low mutual orthogonality.

FIGURE 27. Variability after linear tupling

4 patterns, with size 4 linear tuples

no variation in tuple state  little variation in tuple state
Random tupling need not take its inputs in sequence, and can avoid the case above where some tuples always have the same state. For tupling efficiency to be maximised the probability of a tuple being in any one of the possible states should be approximately uniform, as this maximises the probability of maximising the variability in the tupled output. To maximise variation for all tuples the bit density of the inputs would need to be around 50%. To achieve this the density of the tokens representing the states of the system may need to be increased, which in turn may require the set of tokens to be non-orthogonal. As has been seen already, the probability of combinations including low orders of a particular pattern is higher, so most set bits are at one end of the pattern representing occurrences in the neighbourhood. With non-orthogonal tokens, bits will be overlapped and information lost. Given the number of comparatively similar neighbourhoods, discrimination may be low.

What is needed is tupling that adapts to the variability in the source codes. The method examined below allocates a larger number of tuple inputs than the size of the codes to be tupled. i.e. if the size of the codes is \( N \), and there are \( t \) tuples of size \( s \), then \( ts > N \). The inputs to the tuples are allocated randomly, but weighted towards those positions in the codes where variability in the set of codes is higher. The more tuples of a larger size that are used, the more closely the tupling (here referred to as probabilistic random tupling) can match the variability in the code set.

Variability for a position in the code is determined by computing a basic measure of variability, determined by summing the number of bits in that position across all the codes. If the number of bits set is 0, or the maximum (equal to the number of codes present) then there is no variability. A measure is calculated that indicates the distance from these extremes for each position in the code, creating a vector of weights. The weighting is proportional to the measure of variability. If \( c_i \) is a code in the set of \( n \) codes, \( C \), and \( j \) is a bit position in the code, then if all \( c_{ij} \) for all \( i \) are the same then there is no variability in that column, which occurs if

\[
\sum_{i=0}^{n} c_{ij} = 0 \text{ or } \sum_{i=0}^{n} c_{ij} = n
\]  

(21)

E.g. If the patterns are:

\begin{align*}
010010 \\
010001 \\
010011
\end{align*}

Then the column sums are \((0, 3, 0, 0, 2, 2)\).

If there are \( n \) codes then we calculate \( n/2 \) and then we calculate the probability that an input line will sample at a bit position \( j \) is \( p(j) \).

\[
p(j) = 1 - \frac{1}{n} \left( \frac{n}{2} - \sum_{i=0}^{n} c_{ij} \right)^2
\]  

(22)
In this example this gives us, after normalisation, a vector of weightings for tupling, \( p \), \( (0.0, 0.0, 0.0, 1.0, 1.0) \). Thus we can see that \( j = 5 \) and \( j = 6 \) have the greatest variability, and probabilistic random tupling should ignore all elements of \( c_j \) except \( c_{i_5} \) and \( c_{i_6} \), for any \( i \).

8.4 Results of simulations of probabilistic tupling

8.4.1 Probabilities of sampling

The following graphs (Graph 4 to Graph 11) show how closely for probabilistic tupling follows the theoretical weightings, \( p(j) \). The untupled set is derived from generating the isomorphic forms of all possible neighbourhood states for a four nearest neighbour automaton, with eight cell states being represented by orthogonal tokens of eight bits in size. Let these tokens be \( A, B, C, D, E, F, G, H \). Note that any connecting lines between points are shown only for clarity, the data being discrete.

Graph 4. Number of samples per code position for a number of tuple maps with 16 tuples of size 4.

As can be seen in Graph 4 the number of samples for a typical tuple map can vary widely, since the level of random bias is large, and with the comparatively number of tuples above the variation in the results due to sampling “noise” is high. Note that for the region of code positions 25 onwards there are no samples at all from any of the tuple maps shown, whereas the theoretical values are very small but non-zero. Let the size of a tuple be \( s \), and the number of tuples be \( t \). For smaller values of \( st \), the region from 25 onwards is unlikely to be sampled, thus codes identical except in the region from 25 onwards will be unlikely to be distinguished. The area from 25 onwards represents the last token bin, which is full only if the neighbourhood has exclusively one type of token. If there are fewer than four tokens then the code representing one of the other tokens will be present in one of the other token bins, 1-3. Thus despite the fact that no, or almost no, sampling takes place of the 4th token bin, there is data in token bins 1-3 that allows a neighbourhood AAAA to be distinguished from AAAX, where \( X \in \{ B, C, D, E, F, G, H \} \). Thus the lack of sampling in token bin 4 is compensated for.

For each graph, Graph 5 to Graph 9, below, the plots show the empirical results for a particular set of tuple parameters, but averaged over 50 experiments, and normalised by dividing each sampling frequency by the number of samples.
predicted by theory. (For example, if the weighting for sampling at position were 0.5, and there were 6 input lines for sampling, then the theoretical number of samples would be 3. The empirical data might show examples of this position being sampled 2, 2, and 3 times, for an average of 2.3333, which is then normalised on the graph to 2.3333/3.0, or 0.7777). The standard deviation is plotted by error bars. The theoretical values predicted by equation 22 are also shown.

Graph 5. Number of samples per bit position for 16 tuples of size 4

Graph 6. Number of samples per bit position for 24 tuples of size 4

Graph 7. Number of samples per bit position for 32 tuples of size 4
Graph 8. Number of samples per bit position for tuple size 5, 32 tuples

Graph 9. Number of samples per bit position for tuple size 6, 32 tuples

In Graph 10 and Graph 11 the deviation between empirical values (calculated over 50 experiments) and the theoretical values is calculated. This is calculated over all bit positions. For example, if the number of samples predicted by theory for bit positions 0, 1, 2, and 3 where 7, 6, 6, and 2, and the empirical data showed average sampling frequencies for these bit positions of 6.5, 5.9, 5.7, and 2.3, then the deviation calculated for the plot would be \[ \sqrt{(7 - 6.5)^2 + (6 - 5.9)^2 + (6 - 5.7)^2 + (2 - 2.3)^2}, \] or 0.1936. This is a form of standard deviation.

Graph 10. Deviation between theory and experiment for varying numbering of tuples, with tuple size 4
Graph 11. Deviation between theory and experiment for varying tuple size, with 32 tuples

Graph 10 summarises how close the empirical data from Graph 5 to Graph 7 matches the theoretical required sampling. As can be seen, as the deviation decreases as the number of tuples increases, indicating a better match.

Graph 11 summarises the level of match between empirical and theoretical sampling for Graph 7 to Graph 9. Note that when the tuple size is small the distribution matches the theory more closely. This is because each input for each tuple must be distinct. I.e. a tuple may sample code positions (1, 2, 3, 4), but may not sample (1, 2, 2, 2). Thus as the tuple size increases towards the size of the code the sampling density per bit position for each tuple, \( p(i) \), tends to a constant, \( N/s \), where \( N \) is the size of the code and \( s \) is the size of the tuple. Thus the overall density would tend to \( N^t/s^t \), where \( t \) is the number of tuples. Thus to match the distribution most closely a large number of small tuples is required. However, larger tuples lead to less saturation of the correlation matrix memory, as shown in the next subsection.

8.4.2 Saturation of matrix memory when using probabilistic random tupling

The following are graphs showing the saturation of the first correlation of an ADAM as associations are presented to the ADAM by looking at the number of collisions between the number of bits in the input patterns and number of bits already set in the matrix memory. The tokens were of size five with one bit set, and the neighbourhood size was 8.

Graph 12. Saturation with random tupling of 10 tuples of size 4 with and without replacement
In Graph 12 we see that for small tuple sizes and number of tuples, tupling, of any form, produces input patterns that create saturation in the first correlation matrix memory of the ADAM that are greater than that for untupled codes. Thus small tupling parameters lead to poorer performance.

Graph 13. Saturation with original patterns and probabilistic random tupling of 16 tuples of size 4

Graph 14. Saturation with probabilistic random tupling and random tupling with 32 tuples of size 4

Graph 12 to Graph 14 indicate that as the size of the tuples increases first correlation matrix memory is saturated less quickly than when using the untupled patterns, and that as the number of tuples increases the probabilistic method, matched to the input pattern set, produces lower saturation than the random tupling. This would suggest that the probabilistic tupling may lead to better recall statistics.
Graph 15. Saturation with tupling with 8 tuples of size 5

Graph 16. Saturation with 16 tuples of size 5

Graph 17. Saturation with 16 tuples of size 6

Graph 15 to Graph 17 indicate that as the size of tupling increases the probabilistic tupling method again creates inputs less likely to saturate the first correlation matrix memory than the original patterns, or random tupling with the same tuple parameters.
Graph 18. Saturation with probabilistic random tupling of various sizes

| patterns after probabilistic random tupling, 32 tuples of size 4 | + |
| patterns after probabilistic random tupling, 16 tuples of size 5 | x |
| patterns after probabilistic random tupling, 16 tuples of size 6 | o |

Graph 18 shows how saturation for the probabilistic tupling method varies for a number of tuple parameters. It can be seen that few tuples with a large size produces low saturation. This is expected as this produces input patterns which have low density. Low density itself does not translate to good discrimination, since as was seen in Graph 12, large tuple sizes do not necessarily produce sampling that matches as closely that desired as a larger number of small tuples. However, at all stages probabilistic random tupling produces lower saturation than random tupling for equivalent tuple and teach parameters, which would suggest that for the same parameters there is an improvement in discrimination. The recall behaviour must be explicitly investigated to determine if a large number of small tuples, or a small number of large tuples, produce the best recall behaviour.

8.4.3 Number of associations learned correctly

The following shows the proportion of associations learned correctly given some of the tupling and teaching regimes detailed in §8.4.2, both for randomly allocated class and neighbourhood codes, and ones allocated using the method of Baum, Wilczek and Moody[68]. The recall statistics are taken after teaching 400 associations.

---

1 The method of Baum, Wilczek and Moody (BM&W) describes a way of creating a set of codes that have good mutual orthogonality properties, which are analytically calculable.

If there are \(k\) bits to be set in the code, the width of the code is divided up into \(k\) mutually prime fields, \(f_1, f_2, \ldots, f_k\), which are of size \(s_1, s_2, \ldots, s_k\), such that there is no common denominator between \(s_1, s_2, \ldots, s_k\). The values required for \(s_1, s_2, \ldots, s_k\) also depend on the number of codes required. A counter, \(c\), is incremented for each new code calculated, and a bit set in each field, \(f_i\), at a value of \(c\) modulo \(s_i\).

For example, if three bits codes are required then the values of \(s_1, s_2, s_3\) might be 2, 3, 5, and the codes generated would be (with field divisions shown by spaces in the code)

- 10 100 10000
- 01 010 01000
- 01 001 00100
- 01 100 00010
- 10 010 00001
- 01 001 10000 etc.
As can be seen in Graph 19 and Graph 20 probabilistic random tupling always does better than random tupling in terms of accurate recall of associated patterns. Using BM&W coding improves the recall statistics, but has a greater effect in conjunction with probabilistic random tupling than it does with random tupling. Note that 32 tuples, of size 5, has a marginally better recall statistic than 16 tuples of size 6, despite the saturation of the first correlation matrix memory being higher for 32 tuples of size 5. This is because 32 tuples of size 5 matches the required variation of input sampling better, and this outweighs the disadvantage due to the increase in saturation.

8.5 Conclusions on Probabilistic Random Tupling

As can be seen above, probabilistic random tupling provides a method which recodes a set of input vectors to a second set that has better orthogonality characteristics than would be the case were random tupling used with the similar tuple sizes and numbers.
No detailed comparison was done of the time required for the processing of tuples for the various tupling methods. However each method simply addresses the input space, and apart from the details of the creation of the tuple map there is no essential difference in the mechanism of the tupling. It takes longer to calculate the tuple map for probabilistic random tupling, since an analysis must first be made of all the possible input codes, however this process only takes place once, in the initial set up phase, and does not affect the speed of processing later.

Probabilistic random tupling is an engineering solution to a more fundamental problem of coding theory. While it improves the properties of a set of vectors it does not analyse the deeper questions concerning coding and associative memories and it is presumed that there will be other methods which perform better orthogonalisations of sets of vectors.

The time required for the probabilistic random tupling is the same as random tupling, since they both rely on a tuple map that is generated prior to any iterations. The overhead for probabilistic random tupling is the initial creation of this map, which requires that all input patterns are known, i.e. that all transitions are known. Typically the transitions are known, except in the case of adaptive systems. For very large systems the overhead required to calculate the probabilities on which the mapping is based may be prohibitive.

### 8.6 Conclusions on Occurrence Checking

Due to the lack of orthogonality of the sets of input vectors provided by the preprocessing for occurrence checking, recoding is required to improve orthogonality, and probabilistic random tupling (or something better) is a requirement for the system to have a significant level of success for anything other than a very small number of transitions.
9 Composite System

This chapter describes how a system that deals with a mixture of transitions with various levels of degeneracy might be easily built by combining a number of smaller, simpler units together.

9.1 Partitioning of sets of local transitions

There may be cases when the local transitions of a cellular automaton do not quite fit with a single type of symmetry or degeneracy. E.g. we could envisage a modification of Conway’s Life which is standard, apart from when the neighbourhood of the cell is as shown in Fig. 28. In this case rather than a cell being ‘born’ in the central position, as is normal when there are three ‘living’ nearest neighbours, the cell remains ‘dead’. This exception to the normal set of isomorphic rules indicates a directional preference.

FIGURE 28. Exception to standard Conway’s Life

Local transition functions with these breaks in degeneracy will be suitable for modelling physical phenomena where there is not full degeneracy, e.g. modelling sediments in the presence of a gravitational field, and so on.

To model a system which is fully degenerate the isomorphic system described in chapter 8 can be used. If only one associative memory may be used then even if the degeneracy is broken in one case, then only the anisotropic system may be used. Since the anisotropic system is the slowest this represents an increase in recall time out of all proportion amount of divergence from a fully degenerate system. For this reason a system is sought that allows a more proportionate increase in recall times as degeneracy is broken, but ideally uses associative memories that inherently embody degeneracy when possible.

The local transition function may be partitioned into a number of overlapping parts. In general there will be an underlying degenerate set of transitions, on top of which there are a number of various less degenerate transitions, that partly obscure the transitions of higher degeneracy.
Consider Conway’s Life as an example.

Previously (§3) the transitions were broken up into three categories - anisotropic, isotropic, and isomorphic.

Consider an example of Conway’s Life with an exception:

We denote an ordered set of inputs states as $(p, q, r, s, t, u, v, w)$

The basic transitions of Conway’s Life state that:

- a cell with 0 neighbours in state 1 always ends in state 0 \hspace{1cm} (22.1)
- a cell with 1 neighbours in state 1 always ends in state 0 \hspace{1cm} (22.2)
- a cell with 2 neighbours in state 1 stays in the same state \hspace{1cm} (22.3)
- a cell with 3 neighbours in state 1 always ends in state 1 \hspace{1cm} (22.4)
- a cell with 4 neighbours in state 1 always ends in state 0 \hspace{1cm} (22.5)
- a cell with 5 neighbours in state 1 always ends in state 0 \hspace{1cm} (22.6)
- a cell with 6 neighbours in state 1 always ends in state 0 \hspace{1cm} (22.7)
- a cell with 7 neighbours in state 1 always ends in state 0 \hspace{1cm} (22.8)
- a cell with 8 neighbours in state 1 always ends in state 0 \hspace{1cm} (22.9)
But we add an exception - the above rules hold unless the neighbourhood is \((0, 0, 1, 1, 0, 0, 0)\) in which case the cell always ends in state 0. (22.10)

The underlying transitions exhibit degeneracy, apart from the exception noted in equation 22.10.

If only one associative memory may be used to hold the local transitions then due to the non-degenerate exception to the generally degenerate transitions, the anisotropic system (as described in §6) must be used. This requires mapping 256 separate neighbourhood states to 4 neighbourhood codes, by using 256 separate associations.

If multiple associative memories are used, each one holding a part of the local transition function as shown in Fig. 29, then a single exception to the general degenerate nature may be learned by one associative memory, and the rest of the transitions learned by an associative memory trained in the isomorphic form (chapter 8). This requires 1 association for the single exception, and 9 for the general behaviour, a total of only 10 associations. However, the exceptions must be checked first. As shown in Fig. 29, there may be a hierarchy of associative memories and classes of transition, from most specific to most general, and the neighbourhood state must be checked in this order for a match with one of the transitions stored in the corresponding associative memory.

### 9.2 The composite system of associative memories

The neighbourhood state matchers are identical to the ones outlined in chapters 5, 6, 7, and 8, except for partial matching ability. Each matcher contains the preprocessing stage, and an associative memory (ADAM) that contains the associations between preprocessed neighbourhood states and the appropriate neighbourhood code. The set of neighbourhood codes is shared among the matchers. A case in point is a neighbourhood \(c\) for the anisotropic matcher represents a neighbourhood that has the same transitions as a neighbourhood having neighbourhood code \(c\) for the isomorphic matcher, although the actual neighbourhoods will be distinct.

The combination of the anisotropic matcher and second associative memory is, apart from the partial matching ability and the number of transitions learned, the same as the anisotropic system from §6. The combination of the isotropic matcher and second associative memory is, apart from the partial matching ability and the number of transitions learned, the same as the isotropic system from §7, etc.
9.2.1 Teaching

Each associative memory does not necessarily learn the full set of associations required for closure. For example, if there is one transition that breaks the underlying degeneracy only this transition is learned by the associative memory for that break in degeneracy. The associative memory responsible for learning isomorphic transitions, the mostly highly degenerate transitions, may or may not have a set of transition that ensure closure, depending on the number of exceptions to degeneracy. If there are sufficient exceptions then an underlying isomorphic transition maybe be completely masked.

9.2.2 Recall

(1) Find match for neighbourhood:

(a) attempt match for neighbourhood with anisotropic transitions, if not

(b) attempt match for neighbourhood with isotropic transitions, if not

(c) attempt match for neighbourhood with isomorphic transitions

(2) The set of three neighbourhood state matchers output a neighbourhood class code, that forms the input to asso-
ciative memory 2, along with the current state.

The neighbourhood state code is a $k$-point code uniquely representing the neighbourhood state as seen by the neighbourhood state matchers.

e.g. the isomorphic matcher will output the same code for all these neighbourhoods

**FIGURE 31. Degenerate neighbourhoods for isomorphic system**

but the anisotropic matcher will output a different code for this neighbourhood, even though it is one that is present in the neighbourhoods that the isomorphic matcher will match with.

**FIGURE 32. Anisotropic exception**

Thus some neighbourhoods may have two or more codes associated with them, but the precedence relationship that selects from the most specific to the general should ensure that only one neighbourhood code is passed onto the second correlation matrix memory.

### 9.2.3 The unwanted partial match problem with l-max thresholding

Normally ADAM has a partial matching capability. The partial matches break down into two broad classes:

1. An input may match more than one transition. After l-max thresholding more than l bits may be set in the thresholded class code separator. This represents a match of multiple rules, the class code separator containing superimposed class code separators for all rules partially matched. This sort of partial match can be easily detected by counting the number of bits set in the class code separator.

2. An input may not match any association learned by the matrix memory, but an association learned contains an input close to the input presented. L-max thresholding produces a class code separator with 1 bit set, as expected, but the match is not exact, and the raw response from which the thresholded class code separator is produced is lower.
In the composite system the associative matchers dealing with breaks in degeneracy from more degenerate types of transitions may have few associations, potentially as few as one. Thus there is a high likelihood of spurious partial matches that partially match only one association, i.e. ones that are l-max thresholded to perfectly valid class code separators for the lower-degeneracy associative matchers. This would lead to spurious matches on neighbourhoods by the lower degeneracy matchers, unless partial matches are prevented for the lower degeneracy associative matchers in the composite system, i.e. for anisotropic matcher an exact match is required, and any partial matches are rejected by using a thresholding scheme in the associative memory that rejects partial matches. For example, if the thresholding, rather than l-max, is changed to a hard thresholding at the number of bits set in the input, then the match must be exact for a valid class code separator to be output by the associative memory. In the example above the thresholding value would be 4, and thresholding at this value for recall 2 would lead to a class code separator with 0 bits set, which would be invalid, indicating a failed match.

Preventing partial matches in this way has consequences for the fault-tolerant behaviour of the system. Should an element in the neighbourhood be in an unexpected state due to an error in its associative processes, such that a match with a lower degeneracy associative matcher would have been expected had the cell in the neighbourhood been in the expected state, the lack of partial match will mean that this neighbourhood state will not be recognised by the lower degeneracy associative matcher. The isomorphic system may still have partial matching enabled since there are no other matchers of lower precedence.

9.2.4 Parallelisation

In the description above it has been assumed that the lower degeneracy matching is conducted first, then the higher degeneracy matching, in sequence. If a parallel machine is available then all matching for the neighbourhood can be conducted in parallel, and the final neighbourhood state code passed on depending on which associative memories found a match for the neighbourhood state, but as dictated by the precedence relation, low degeneracy matcher to high degeneracy matcher.
9.3 Composite System Performance

9.3.1 Introduction

The rationale behind the composite system is that it allows isomorphic, isotropic and anisotropic transitions to be mixed in a single system, using the high speed of the isomorphic system for most rules, with anisotropic and isotropic parts of the system being used to match a few transitions. It is hoped that since only a few transitions may be matched by the anisotropic and isotropic systems, the sizes of these subsystems can be kept small, and the overall time for recall kept down as far as is possible. As was seen in §4.3.7 the speed of recall of an associative memory is approximately proportional to the class code size.

9.3.2 Results of simulations

Results are first presented from simulations run using an isomorphic implementation of Conway’s Life as the basis, with an increasing number of anisotropic and isotropic exceptions. In fact the ‘exceptions’ are simply elements of the anisotropic and isotropic transitions for Conway’s Life, taught into those subsystems, but matching may occur on these first. A number of random initial configurations of various densities were used, and the iteration times averaged across them to provide the widest cross-section of data. The experiments all assume that data has been averaged over 50 simulations.

Graph 21. Teaching times as anisotropic transitions are added

In Graph 21 we can see that as anisotropic transitions are added to a system with an underlying isomorphic character that the total teaching time increases with a gentle curve, indicating a non-linear relationship. Note that there are two points at around 250 transitions which are far from the expected curve. The reason for these two elements in unknown.
Graph 22. Iteration times for 25x25 automaton as anisotropic transitions are added.

Note: in Graph 22 the data has been smoothed.

As can be see in Graph 22, the general trend is for the time taken for each iteration of the automaton to rise as the number of anisotropic transitions present is increased. The equivalent time for a fully anisotropic system is the time for the above with 256 anisotropic transitions. As can be seen, for systems with fewer than this number, considerable savings in execution time can be made by implementing the system in a composite fashion over an all-anisotropic fashion.

The recall time appears to increase linearly with the number of anisotropic transitions added. In fact there will be a small constant overhead time required for the creation of the anisotropic input vector to the anisotropic transition associative memory. However, the time required to create an anisotropic input is negligible, and so not visible on the scale of the graph above.

Graph 23. Time to teach automaton of size 25x25, with addition of isotropic transitions

Note: the stepping above is due to the grouping of similar sets of neighbourhoods in order.
The results are less clear for the addition of isotropic transitions, mostly due to the fact that there are fewer isotropic transitions to add, and so the trend is over a smaller interval, but the trend is indicated by the dotted line in Graph 24.

The trend is still apparent.

Note the periodic waves in training time, which are due to the way similar transitions were grouped in the data set, with similar transitions grouped together and transitions being added in numerical order.

Note also that for 0 isotropic transitions (i.e. a purely isomorphic case) the speed is far lower than the instances with a number of isotropic transitions. The jump (which is followed by an approximately linear increase in recall time) is due to the fixed overhead of creating the isotropic input vector.

### 9.4 Times required to create inputs, and times for recall

The following set of results look at the times required to create the inputs for each subsystem, as opposed to the actual time for recall of associations from the first ADAM stage. The results are the proportion of time spent on each of these tasks, and are taken from iterations of Conway’s Life, over 50 iterations with an initial configuration density of 0.5, and orthogonal neighbourhood codes.

<table>
<thead>
<tr>
<th>Type of matcher</th>
<th>Proportion of time to create input pattern to ADAM</th>
<th>Proportion of time to recall from ADAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>anisotropic</td>
<td>(46.1 ± 1.8)%</td>
<td>(42.1 ± 1.5)%</td>
</tr>
<tr>
<td>isotropic</td>
<td>(79.7 ± 0.5)%</td>
<td>(16.1 ± 0.5)%</td>
</tr>
<tr>
<td>isomorphic</td>
<td>(58.1 ± 1.4)%</td>
<td>(23.4 ± 2.1)%</td>
</tr>
</tbody>
</table>

**TABLE 4. Times for recall with orthogonal class code vectors**
The following graphs (Graph 25 to Graph 27) show the proportion of time spent on creating input patterns, and the time spent on recalling given those input patterns, for various matching systems. In each case there are two bits set to 1 in the first ADAM class codes.

**Graph 25. Time required to create and recall patterns for an anisotropic system as a function of class size**

![Graph 25](image)

**Graph 26. Time required to create and recall patterns for an isotropic system as a function of class size**

![Graph 26](image)

**Graph 27. Time required to create and recall patterns for an isomorphic system as a function of class size**

![Graph 27](image)

As can be seen from these results, Graph 25 to Graph 27 the time required to create the input pattern for the first ADAM stage is a major portion of the required time for recall of the neighbourhood code.
The creation of the input pattern for the first ADAM is a serious bottleneck for all of the subsystems described in the previous chapters, as data must be brought in from a number of cells, and this is a slow process. However, for some of these subsystems the time required for creation of these input patterns is greater than others. For a system with isomorphic degeneracy and isomorphic implementation is still faster than an anisotropic implementation due to the few associations required to encode the transitions, but the time required to create the input patterns is greater. Thus the systems do not run, even for the anisotropic system, at anything like the potential speed of the associative memories.

The following graphs (Graph 28 and Graph 29) show the accuracy of a composite system. The transitions are, however, the same for the non-isomorphic subsystems and the underlying isomorphic implementation. The graphs are for Conway’s life, with two bits set in the first ADAM class code for the non-isomorphic subsystem, orthogonal class codes for the isomorphic subsystem, and orthogonal neighbourhood codes.

**Graph 28. Transitions learned correctly with anisotropic transitions overlaying isomorphic.**

Note that Graph 28 does not start at the origin.

Note that in Graph 28 there is a trough of lower recall success as the number of anisotropic transitions reaches around \( \frac{2}{3} \) of the maximum. This is due to associations being correctly identified as incorrectly learned, on an attempt to recall, and the underlying isomorphic transitions being triggered. The underlying isomorphic transitions are always correctly recalled since orthogonal class codes are used during the teaching phase for these particular transitions. The dotted lines show the approximate behaviour of the failure to recall the anisotropic transitions alone, were the underlying isomorphic transition not there to cope with such failures. The presence of the isomorphic transitions can be considered to be a form of fault tolerance, as it means that some default behaviour is always available to be followed.
Graph 29. Transitions learned correctly with isotropic transitions overlaying isomorphic. Note that the graph does not start at the origin.

In Graph 29 a number of ripples are seen, which are smaller analogues of the trough seen in Graph 28.

9.5 Conclusion

The graphs indicate that implementing a system as a base isomorphic systems with exceptions provides a way to implement more complex systems in a way that scales with the additional non-isomorphic transitions learned. For a small number of non-isomorphic transitions the speed of recall is significantly lower than a fully non-isomorphic system. However, there is an overhead in using additional associative memories that is independent of the number of additional transitions learned, and this is the creation of the required input vectors. The vector creation overheads are actually largest for the isomorphic subsystem, which has the poorest recall time \textit{per transition}, but has learned far fewer transitions, thus performs better on the transitions as a whole.

Recall using the composite system is faster than the recall of the pure anisotropic or isotropic systems. Thus the composite approach is useful.
10 Implementation of Conway’s “Life”

This chapter investigates the implementation of Conway’s Life, a binary automaton, with anisotropic, isotropic, and isomorphic systems.

10.1 Introduction

Conway’s Life is an example of a totalistic binary cellular automaton, i.e. the order of the tokens in the neighbourhood is unimportant, just the number of neighbourhood elements in a particular state. However the associative memory architecture for implementing a cellular automaton described above does not exploit this degeneracy, and therefore takes a large number of rules to encode Conway’s “Life” as it must enumerate each combination of tokens in the neighbourhood. 512 rules are required to encode Conway’s “Life”.

10.2 Simulation Results

The reader should be aware that there is an error in the results since the many-to-one mapping in the first associative memory was not used. A case in point is that whereas all neighbourhoods with 0 elements set to one should have been mapped to the same neighbourhood code as all neighbourhoods with 1, 4, 5, 6, 7, and 8 elements set to 1, this was not done, and separate neighbourhood state codes allocated for each of these. Thus the neighbourhood state codes are larger than strictly required, meaning that all correlation matrix memories are larger than required. Had the many-to-one mapping been employed the speed of recall could have been increased. There was insufficient time to correct these results, but preliminary results indicate that there is an approximate five-fold speed up for the simulation of Conway’s Life over the results presented.

The following presents results for simulations using the composite system with three basic rule sets:

1. An anisotropic system with 512 transitions.
2. An isotropic system using 72 rules.
3. An isomorphic system using 18 rules.

Each system fully implements Conway’s Life.

Each system was run on a number of configurations. The configurations were created with a particular density of bits set to 1, the actual configurations being created by a random allocation of bits. The configurations were 25 by 25 elements, for a total of 625 cells.
Various statistics were extracted for each system for a variety of class code parameters

(1) how closely its performance matched that of a reference automaton (not using associative memories).

(2) the time required to perform the iterations.

(3) the number of rules learned correctly.

(4) the time required to teach these rules.

(5) the saturation of the first two correlation matrix memories.

10.3 Analysis of results of running of automata

As Life is a binary system a simple Hamming distance measure between two configurations can be used to indicate how much the two automata have diverged.

If the configuration of the reference system at a time $t$ is $R_t$, and an element at position $(i, j)$ is $R_{i,j}$, and the corresponding element for the configuration generated by the system of associative memories is $A_{i,j}$, the automaton being of size $(m, n)$ then the Hamming distance will be

$$
\sum_{i=1, j=1}^{m,n} |A_{i,j} - R_{i,j}| = H
$$

From several runs, an average hamming distance, $\bar{H}$, is calculated. This comparison allows the data to be viewed as a whole, but obscures the details.
Graph 30. Anisotropic system. 512 randomly selected codes of size 256 with 2 bits set to 1.

Graph 31. Anisotropic system. 512 randomly selected codes of size 1024 with 2 bits set to 1.

Graph 30 and Graph 31 show the average hamming distances, $H$, averaged over 50 simulations for each plot. This comparison allows the data to be viewed as a whole, but obscures the details. As can be seen from the graphs after a number of iterations $H$ tends to a constant value. This seems to correspond to the emergence of small stable configurations in one or other of the systems. The small ‘satellite’ sub configurations may be present in the system using associative memory while not in the reference system, due to an error in the rules learned by the system using associative memories spawning a chain of events that creates an island that can be sustained. More commonly it indicates an absence of such stable subconfigurations when they should be present. However, the trivial case of stable subconfigurations is when all cells are in the quiescent state in one or both of the systems.
Graph 32 shows the Hamming distance track for a particular instance of an automaton run. A fixed distance between the configurations and reference configurations for each iteration can be seen. Looking at the actual configurations shows that this is the case because there are subconfigurations missing in the system using associative memories during the latter iterations. In fact, there are no elements that are not zero at all. For this reason comparing the densities of the configurations at particular iterations provides the best comparison between a system using associative memories and the reference system.

Graph 32. Hamming distance between reference system and associative memory system with 512 randomly selected class codes of size 256 with 2 bits set to 1. Initial configuration bit density 0.1

Here we can see that after some initial variation the difference in Hamming distance between the reference system, which does not use associative memories, and the system using associative memories tends to a constant.

As can be seen in the configurations over time (Fig. 34) the automaton implemented with associative memories fails to sustain the automaton, and it becomes quiescent. Plotting hamming distances between two sets of parallel configurations may obscure this behaviour. For this reason the density of the configurations may be a more useful measure for looking at average behaviour.

FIGURE 34. Configurations of automaton for Graph 32
10.4 Densities of configuration (graphs exclude initial configuration)

This data was taken from an average of 50 simulations per point plotted, using a variety of initial configurations of the same given density.

**Graph 33. Configuration densities over time for reference system with various initial configurations**

**Graph 34. Configuration densities over time for anisotropic system with various initial configurations**

Class 1: 2 bits set in 256. Neighbourhood code: 2 bits set in 256. 4 random tuples

**Graph 35. Configuration densities over time for isotropic system with various initial configurations**

Class 1: 2 bits set in 288. Neighbourhood code: 1 bit set in 36. 4 random tuples
Graph 36. Configuration densities over time for isomorphic system with various initial configurations

Class 1 bits: 2 bits set in 18. Neighbourhood code: 1 bit set in 18. 4 random tuples

Graph 33 to Graph 36 show the ability of the various subsystems to support an automaton. This, along with the speed required to process the configurations, is the main criterion for the success of the system of associative memories used to implement the cellular automaton.

As can be seen from the above, the anisotropic system fails to sustain a configuration at any point. If the pure recall statistics are examined (§10.7), this may seem surprising. However, Conway’s Life is an automaton that is highly susceptible to transitions being learned incorrectly. If there were two automata, A and A’, with configurations at time t being $C_t$ and $C_t'$, which were different by a small amount, then at time $t + \Delta$, $C_{t+\Delta}$ and $C'_{t+\Delta}$ might be very different. I.e. Conway’s life is an chaotic system.

10.5 Times of iterations of automata

These timings are for an initial configuration with density 0.5, except where indicated. Parameters for each associative memory system are as for Graph 34 to Graph 36. Each point was averaged from 50 simulations.

Graph 37. Iteration times for reference system given various initial configuration densities
Graph 38. Iteration times for anisotropic system given various initial configuration densities

Graph 39. Iteration times for isotropic system given various initial configuration densities

Graph 40. Iteration times for isomorphic system given various initial configuration densities

As can be seen in the graphs (Graph 37 to Graph 40) above, the isomorphic system is the fastest. However, if the times for the recall time per transition are examined it will be discovered that it is the slowest, even despite having by far the smallest class codes for the first ADAM. This apparent slowness is due to the large overhead required to produce the input vector, which involves counting the number of occurrences of patterns in the input, and copying of data into the input vector. The isomorphic system is the fastest because employing degeneracy allows the data to be encoded in a way which uses so few transitions, and thus can be encoded with few class codes, which then means that a small class code size may be used, which is the main limiting factor on the recall time, as seen in §4.3.7.
10.6 Times for teaching

Times indicated includes token generation, and testing of recall. Each point was averaged from 50 simulations.

Graph 41. Teaching times for anisotropic system given various class code parameters

Graph 42. Teaching times for isotropic system given various class code parameters

Graph 43. Teaching times for isomorphic system given various class code parameters

The above graphs indicate that teaching time increases linearly with neighbourhood code size and first ADAM class code size. The graph for the isomorphic system is less smooth since it is over such a small set of class intervals.
10.7 Recall

Each point was averaged from 50 simulations.

Graph 44. Recall accuracy for anisotropic system given various class code parameters

Graph 45. Recall accuracy for isotropic system given various class code parameters

Graph 46. Recall accuracy for isomorphic system given various class code parameters

The above graphs (Graph 44 to Graph 47) show the trends for the recall, given various code parameters. Note that while the anisotropic system achieves nearly 50% accuracy for some combinations, this does not enable it to sustain an automaton (see Graph 34).
10.8 Saturation of ADAM matrix 1 (proportion of bits set to 1)

Each point was averaged from 50 simulations.

Graph 47. Matrix 1 saturation for anisotropic system with various class code parameters

Graph 48. Matrix 1 saturation for isotropic system with various class code parameters

Graph 49. Matrix 1 saturation for isomorphic system with various class code parameters
10.9 Saturation of correlation matrix memory 2

Each point was averaged from 50 simulations.

Graph 50. Matrix 2 saturation for anisotropic system with various class code parameters

Graph 51. Matrix 2 saturation for isotropic system with various class code parameters

Graph 52. Matrix 2 saturation for isomorphic system with various class code parameters
10.10 Conclusions

After accuracy of recall, which affects the accuracy with which the system models the chosen cellular automaton, the most important criterion for judging the system is speed. The raw results presented here indicate that even the isomorphic system, which given its compact coding of a totalistic automaton is the fastest of the three associative methods, is approximately 21 times slower than the reference system which is hand coded in ‘C’. However, as was pointed out in §10.2, there is an improvement in the way that neighbourhood codes can be allocated. Allocating the neighbourhood codes properly will reduce the number of neighbourhood codes to only 4, and this leads to the isomorphic system performing only about 4 times slower than the reference system, while retaining 100% accurate recall.

Since the anisotropic system has a large number of transitions, with the code parameters specified only a comparatively small section of code space is allowed for each transition. The isotropic system has fewer rules, and thus more volume in code space per code. In addition the coding of the input vectors for the isotropic system is more mutually orthogonal, even given tupling on the inputs for the anisotropic system. Thus, as can be seen in Graph 47 to Graph 52 the saturation of the first ADAM matrices are highest for the anisotropic system, to lowest for the isomorphic system. Hence the anisotropic system is most prone to errors in recall, as is seen in Graph 44 to Graph 46.

A paper by Song Yan [69] indicates that associative memories, specifically ADAM, only have benefits for speed of recall, given their relative complexity, if the number of associations is relatively large. Thus Conway’s Life, being a relatively small system, especially when the totalistic nature is exploited, is an unfair test of the associative system. If, however, it can perform at one-quarter of the speed of a hand-coded system this should be seen as a victory. Whilst the hand coded version is obviously faster, it indicates that a large system might run faster with the system of associative memories, provided class compression is used. It should also be noted that specialised hardware is being developed for ADAM[33][39][44][45], which offers the possibility of improvements in recall speed. However large systems require larger storage for the associative memories, unless accuracy is sacrificed for lower storage demands.
11 Implementation of Langton’s Self-Reproducing Automaton

This chapter introduces a comparison between Langton’s Self-Reproducing Automaton, implemented using associative memories and the Cellang cellular automaton programming language of J. Dana Eckhart. I would like to thank J. Dana Eckhart and Chris Langton personally for their help.

11.1 Aims

This chapter compares two implementations of Langton’s automaton, one using a system based on associative memories, and one using a compiled language, both of which run on general computers without a requirement for parallel hardware (although both systems have scope for implementation using parallel hardware). This investigation attempts to determine if there are any speed advantages to be had using an associative memory implementation without excessive hazards due to the unreliable nature of associative memory computing elements when class codes are compressed for improved speed and storage capacity.

11.2 Langton’s Automaton

Langton’s automaton is selected for investigation as it is a self-reproducing system, which has implications concerning the ability of the space to contain a general computing machine embedded within it. However the Langton system is relatively small.

Langton’s self-reproducing system [53] defines a loop, comprising a ‘sheath’ and data internal to this loop which can produce a copy of the loop and data after a number of iterations. Langton’s system is an important simplification of Codd’s self-reproducing system, itself an attempt of simplifying Von Neumann’s 29 state system. Langton’s system uses 8 states, four nearest neighbours, and the transitions are isotropic.

11.3 Cellang

Cellang is a system developed by J. Dana Eckhart [67] and is a compiled language for manipulating cellular automata, that offers a relatively simple programming interface, along with viewing tools for data produced.
11.4 Results of Simulations

The following are the results of running a 50 by 50 automaton using various implementations of Langton’s Self-Reproducing Automaton, with both associative memory based systems and Cellang. The initial configuration for the automaton was a loop capable of self-reproduction, assuming error-free transitions.

Graph 53 to Graph 55 are calculated from the average of 50 simulations per point.

11.4.1 Teaching

Graph 53 shows time required to teach the transitions for various class code and neighbourhood code parameters. For the ADAM class and neighbourhood class, two bits are set to 1.

Graph 53. Time to teach transitions for Langton’s Automaton

Here we see that the time required to teach the transitions is roughly linear with both first ADAM class size and neighbourhood class size.

Graph 54 shows the number of transitions successfully learned against various class code and neighbourhood code parameters for an cellular automaton based on associative memories.
We see that for all but comparatively small values of the first ADAM class size the performance of the system. The fall off is quite steep as the first ADAM class size approaches the minimum value, which is of a similar shape to that expected for recall from a single ADAM system. The major contribution to failure in this axis of the graph is due to the failure of the first stage to recognise the neighbourhood state, although any erroneous output may be compounded by errors in the second stage.

The fall off as neighbourhood class size reduces is more gradual, as the saturation of the second ADAM, which contains the mapping between neighbourhood states and the new states is lower than the first ADAM, despite the class compression being higher in this associative memory.

For most of the range plotted above the response from individual cells in the system is good.

11.5 Iteration

Graph 55 shows the time required per iteration of a cellular automaton based on associative memories. Also represented on an axis is the number of times slower the system is compared to Cellang. The equivalent required time for Cellang is $(6.472 \pm 0.016) \times 10^{-6}$ seconds.
For reasons not entirely clear the results for the time required per iteration were highly erratic. The fastest times possible correspond with the smallest ADAM class size, representing fast recall from the relatively dense first ADAM, and the largest neighbourhood class size, representing fast recall from the relatively sparse second ADAM. However, even the fastest response time above corresponds to approximately 1400 times slower than the Cellang implementation.

In **Graph 56** and **Graph 57** the states 0-7 are represented by increasing levels of grey from white (0). Black is allocated to represent cells in an unknown state. The first ten iterations are shown, left to right.
In **Graph 57** the class code parameters correspond to a recall rate of approximately 85%. However the behaviour is erratic, and uncontrolled. Langton’s automaton is not an automaton that automatically corrects itself when cells enter states that are not expected given their antecedent states and neighbourhood states. For this reason it is sensitive to unreliable computing elements, such as systems implemented using associative memories with class codes compressed such that errors are possible. For the system implemented using associative memories this is an example of *catastrophic error propagation*.

For a discussion of fault tolerance see §13.

### 11.6 Conclusions

Again the conclusion is that the system implemented using associative memories for a system of this number of transitions is far slower than other implementations. Even for the results plotted in **Graph 57**, which show poor behaviour of the automaton, the system is still running approximately 2000 times slower than the Cellang system. However it should be noted that the first generation of the Advanced Uncertain Reasoning Architecture hardware developed at York [40][42][43][44] should allow (with a full 70 processor system) a 200 times speed up over figures for the associative memory system shown here. Specialised hardware for the implementation of cellular automata is required for the associative memory system to compete with Cellang. However, if specialised hardware is used the valid comparison would be between a system of associative memories supported by hardware, and an implementation not using associative memories, but also using hardware, such as the CAM system.
12 Massive Random System.

This chapter describes an implementation of a system with a large number of transitions.

12.1 Introduction

The nature of associative memories would suggest that the benefit of associative memories over more conventional methods should only be apparent for much larger systems [69].

12.2 The System

An automaton with sixteen states and eight nearest neighbours was chosen, which gives a possible $4.25 \times 10^9$ possible neighbourhood states. A modification was made - if no neighbourhoods matched the transitions taught then the cell state is not updated. Since it is not feasible to generate the required number of neighbourhood codes to characterise every neighbourhood character that is present, a neighbourhood code is allocated for each transition taught. 80000 transitions, a subset of the possible $16 \times 4.25 \times 10^9 = 6.872 \times 10^{10}$ anisotropic transitions were chosen at arbitrarily.

As a reference system was created against which the performance of the system using associative memory could be compared. This was coded in ‘C’. Due to the arbitrary nature of the selection of transitions to be learned from the set of all possible transitions, the state machine had to be achieved by hand coding a series of ‘C’ language if statements detailing all possible neighbourhood states. This hand-coding presents some problems, as the set of transitions must be split into sufficiently small sections to allow current compiler technology to compile the source.

12.3 Results

The following results were generated on Challenge DM system with 4 150MHz IP19 processors.

The following results show the number of transitions learned successfully for a number of class code parameters. In all cases there are 32 tuples of size 4, with 2 points set in the class codes.
For the most accurate set of parameters, i.e. for class codes of 2 bits set in 3000, the following timings were determined (averaged from 50 simulations):

<table>
<thead>
<tr>
<th>class size</th>
<th>class code compression</th>
<th>class code density</th>
<th>n’hood code size</th>
<th>n’hood code compression</th>
<th>n’hood code density</th>
<th>proportion of transitions learned successfully</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>80</td>
<td>0.0020</td>
<td>1000</td>
<td>80</td>
<td>0.0020</td>
<td>1.25×10⁻⁵</td>
</tr>
<tr>
<td>2000</td>
<td>40</td>
<td>0.0010</td>
<td>2000</td>
<td>40</td>
<td>0.0010</td>
<td>1.00×10⁻⁴</td>
</tr>
<tr>
<td>3000</td>
<td>26.7</td>
<td>0.0007</td>
<td>3000</td>
<td>26.7</td>
<td>0.0007</td>
<td>2.00×10⁻⁴</td>
</tr>
</tbody>
</table>

TABLE 5. Proportion of transitions learned correctly for massive random system

Due to lack of time further massive systems could not be investigated, so it is not known if there is a point at which implementations using associative memory become viable over directly coded versions in ‘C’, in terms of speed, or accuracy, or both. The associative memory system is, however, simpler to program since the form of programming is invariant of size, which is not the case for a C implementation, due to the limits of C compilers in handling large program sizes. There will be a point for which C implementations, relying on if statements, cannot be compiled with current C compiler technology.

<table>
<thead>
<tr>
<th>System</th>
<th>Time to teach 80000 transitions (s)</th>
<th>Time per cell per iteration (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Associative memory</td>
<td>(234 ± 2) × 10³</td>
<td>17.0 ± 0.2</td>
</tr>
<tr>
<td>C program</td>
<td>68.63 ± 0.94 *</td>
<td>(1.17 ± 0.01) × 10⁻²</td>
</tr>
<tr>
<td>Speed up of C program</td>
<td>3411 ± 76</td>
<td>14532 ± 295</td>
</tr>
</tbody>
</table>

TABLE 6. Time required for teaching and iterating a massive random system

* This is the compile time, which is the analog of teach time.

12.4 Conclusions

Due to lack of time further massive systems could not be investigated, so it is not known if there is a point at which implementations using associative memory become viable over directly coded versions in ‘C’, in terms of speed, or accuracy, or both. The associative memory system is, however, simpler to program since the form of programming is invariant of size, which is not the case for a C implementation, due to the limits of C compilers in handling large program sizes. There will be a point for which C implementations, relying on if statements, cannot be compiled with current C compiler technology.
13 Fault Tolerance

This chapter describes how to overcome a problem with associative memories: the fact that associative memories, when the class codes are compressed to enhance storage, may become inaccurate.

13.1 Introduction

Von Neumann talked of computation using unreliable elements [8]. Associative memories, if the class codes are not orthogonal, are potentially unreliable elements as there is no guarantee that the recall of a new state, given the current kernel state, will be 100% accurate. This means the cells using associative memories are unreliable computing elements. If associative memories are to be used as the state machines for cells then the problem of the unreliable computing needs to be addressed.

13.2 Failure and fault tolerance at Cell Level

Here we consider the possible modes of error from cells comprising associative memories, and also the possibility that they may be tolerant of faulty information in the kernel, and recover correct new states for the cells despite incomplete information.

13.2.1 Failure with correct state codes in kernel

The following discussion presumes that the cells in the kernel of the cell in question have correct state codes, i.e. that they are valid state codes corresponding to the alphabet of the automaton, with no superposition of state codes. Failure may come about, even when the state codes in the kernel are correct, due to saturation in one or more of the correlation matrix memories.

If we threshold the raw class codes to pick out just maximum values\(^1\) there are four possible results:

1. Recall returns a state code with a Hamming weight of \(k_x\).

---

\(^1\) Assume that the class codes used during teaching are of Hamming weight 2. If the raw class code after summing the columns in the correlation matrix memory is \(5, 4, 4, 1, 0\), then a maximum threshold would produce a class code of 10000. In the same situation L-max threshold at 2 would produce a code with at least 2 bits set, which in this case would have to be 11100. Neither code is 'correct' in that neither has a Hamming weight of 2.
(2) A state code with a Hamming weight less than \( k_s \) may be returned, with no bits in the code corresponding with bits in the expected state code given the kernel state.

(3) A state code with a Hamming weight less than \( k_s \), one or more of which correspond with bits set in the expected state code, given the kernel state.

(4) The code returned is a superposition of state codes of the alphabet, i.e. more than \( k_s \) bits are set in the code.

In each case the failures (2) to (4) may be determined by comparing the number of bits set against \( k_s \), the number of bits that should be set in a state code.

If L-max thresholding were used to threshold the raw output also (and this would be possible, since \( k_s \) is known), then only (4) would be a valid failure mode, since L-max thresholding would automatically ensure the thresholded code had at least \( k_s \) bits set to one, and so what would have been cases of (2) and (3) under maximum thresholding become cases of (4) under L-max thresholding. (see \(^1\) on previous page).

Consider the case when a large number of similar neighbourhood states, \( N_1, N_2, \ldots, N_r \), are associated with the same neighbourhood state code, \( n_1 \). There may be one neighbourhood state, \( N_{r+1} \), otherwise similar to the others which is to be associated with a second neighbourhood state code \( n_2 \). If the class code, state code and neighbourhood state codes are of such density that the neighbourhood state ADAM becomes saturated then when presented with \( N_{r+1} \) the memory may generalise to \( n_1 \), rather than \( n_2 \).

In the following example the automaton is a simple 1D system (2 nearest neighbours).

The following transitions are learned:

<table>
<thead>
<tr>
<th>Transition</th>
<th>Neighbour 1</th>
<th>Neighbour 2</th>
<th>Current State</th>
<th>New State</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( P )</td>
<td>( P )</td>
<td>( P )</td>
<td>( P )</td>
</tr>
<tr>
<td>2</td>
<td>( P )</td>
<td>( Q )</td>
<td>( Q )</td>
<td>( P )</td>
</tr>
<tr>
<td>3</td>
<td>( P )</td>
<td>( P )</td>
<td>( P )</td>
<td>( P )</td>
</tr>
<tr>
<td>4</td>
<td>( P )</td>
<td>( Q )</td>
<td>( Q )</td>
<td>( P )</td>
</tr>
<tr>
<td>5</td>
<td>( Q )</td>
<td>( P )</td>
<td>( P )</td>
<td>( P )</td>
</tr>
<tr>
<td>6</td>
<td>( Q )</td>
<td>( P )</td>
<td>( Q )</td>
<td>( P )</td>
</tr>
<tr>
<td>7</td>
<td>( Q )</td>
<td>( Q )</td>
<td>( P )</td>
<td>( Q )</td>
</tr>
<tr>
<td>8</td>
<td>( Q )</td>
<td>( Q )</td>
<td>( Q )</td>
<td>( Q )</td>
</tr>
</tbody>
</table>

**TABLE 7. Transitions**

These transitions can be moderated using two neighbourhood codes, one for transitions 1-6, and a second for transitions 7 and 8.

Fig. 35 shows recall for a kernel state, showing the effect of saturation in the neighbourhood state ADAM.
Here $P$ and $Q$ are represented by binary codes $01$ and $01$ respectively. The neighbourhood state ADAM class codes are represented by $1010$, $0110$, $0101$, $0011$. The two neighbourhood state codes are represented by $110$ and $011$. The kernel state ADAM class codes are $1010$, $0110$, $0101$ and $0011$. The following represents are recall given the kernel state from transition 8 in Table 7. In this example it is assumed that the transitions are anisotropic, and for simplicity no tupling is considered.

**FIGURE 35. Neighbourhood state ADAM, saturated, on recall**

In Fig. 35 the neighbourhood state code recovered has fewer than $k_n$ bit set to 1. Also note none of these bits match the expected neighbourhood state code which was used during the teaching phase. If Fig. 35 was in fact the kernel state ADAM, then this would represent the recovery of a state code with fewer than $k_n$ bits set to one, of which none would match those of the expected state code for that kernel state. (Failure mode (2)).

**FIGURE 36. Saturation in kernel state ADAM.**
In Fig. 36 we see that with a different partial neighbourhood code (the same code as recovered in Fig. 35), a partial match results in a superposition of state codes, and thus an indeterminate state. (Failure mode (4)).

Signal to noise ratios and confidence values for recall can be computed during recall, which may indicate that the result has been produced with a lower confidence than the average for set of associations, however such a computation would be expensive if performed for every cell in every iteration of the automaton.

13.2.2 Failure with incorrect state codes in kernel

If the incorrect state codes that may be produced due to the failure modes detailed in §13.2.1 are propagated through to the next iteration of the automaton then cells may be presented with state codes in the kernel that are, in some way, incorrect. The behaviour in this instance is complex, and is dependent on both the modes of failure for those incorrect cells, and the form of transition.

Generally it is found that those neighbourhood state codes which are more strongly represented in the teaching set tend to dominate.

Consider the following examples, using the same parameters as Fig. 35 and Fig. 36.

FIGURE 37. Superposition of state codes in the neighbourhood

In Fig. 37 we have the first element of the neighbourhood as a code which is a superposition of the state codes for $P$ and $Q$. Yet the character of $P$ dominates since $P$ is present in 6 of the 8 transitions in Table 7.
FIGURE 38. Incomplete or missing state codes in the neighbourhood

Again, in Fig. 38 the character of $P$ dominates.

Thus we see that those elements dominant in teaching may dominate in recall. If errors creep into an automaton at time $t$, then at $t + 1$ some cell states may have been selected out due to lack of dominance during teaching, resulting in wildly unexpected behaviour by $t + 2$.

It is the reuse of the neighbourhood state codes that may lead to this dominance. If neighbourhood codes are not reused for every neighbourhood state that is degenerate for transitions from state to state in the kernel then this problem may be avoided. However, using more neighbourhood codes then requires that the neighbourhood code size increase if the density is to remain the same, requiring large correlation matrix memories and longer recall times. Thus there is a trade off between the speed of iteration of the automaton, and its ability to reject noise, some of which may have been generated internally due to unreliable computing elements.

13.2.3 Composite systems

As was seen in chapter 8, if we have a composite system, in which each ADAM has only partial information about the whole set of transitions, we have to threshold in such a way that the generalisation behaviour is inhibited to stop unwanted generalisation, leading to an ADAM recognising a neighbourhood state that should actually be recognised by another ADAM. With a hard or Willshaw threshold generalisation is inhibited, and also the partial matching ability.

13.3 Higher levels of fault tolerance

As was seen in §13.2, the system of associative memories itself cannot be expected to reject even internally generated “noise”\textsuperscript{2}. Ideally a higher order of fault tolerance will allow rejection of unexpected cell states by using transitions

\textsuperscript{2} This is not strictly noise, but rather a loss or distortion due to the chaotic nature of the system that it is pragmatic to treat as noise.
which have rejection of “noise” coded into the transitions, rather than the mechanics of the operation of the state machine.

(1) Consider the case when a state code contains a subset of those bits required for a valid state code. For example if the state codes have a Hamming weight of 2, a valid state code might be \( c_1 (1100) \), but we might encounter an erroneous state code in a cell with a Hamming weight of Hamming weight 1, e.g. \( 1000 \) (which shares a set bit with \( c_1, c_2 (1010) \) and \( c_3 (1001) \)).

(2) Alternatively we may have a state code which is the superposition of valid state codes. For example, two valid state codes might be \( c_1 \) and \( c_2 \), and we might encounter an erroneous state code of \( 1110 \) (which is \( c_1 \oplus c_2 \)).

These codes (ones of Hamming weight not equal to \( k_r \)) are passed on with others in the neighbourhood to form the neighbourhood state code. The associative memory then attempts to recognise this neighbourhood, including an incorrect code. The associative memory will inherently attempt to reject noise by partial matching. Assume the associative memory has been taught on strings including \( c_1, c_2, \) and \( c_3 \) with resulting new state codes for the cell being \( c_4, c_5, \) and \( c_6 \) respectively, will lead to a final state code for the cell which is itself of a superposition of two or more state codes being produced. For case (1) the actual resultant state may be \( c_4 \oplus c_5 \oplus c_6 \). For case (2) it will be \( c_4 \oplus c_5 \). These new cell state codes are not valid codes, and they will be propagated out to other cells via the network of neighbourhood corrections, leading to the propagation of errors, with the incorrect states potentially becoming denser and denser superpositions of codes, which will cause saturation problems on the inputs of the associative memories.

The erroneous states noted in (1) and (2) may be easily detected because they do not have the expected number of bits set to one. Such erroneous state codes may then be replaced by a state or states that explicitly represent an error. The error state \( \epsilon \) will be arranged to have \( k_s \) bits set to one, and is thus a valid state code, but one on which a special interpretation is placed.

Given the error state (or states) \( \epsilon \), the local transition function can be designed to take neighbourhood states including a number of cells in the error state, and attempt to rebuild the expected configuration at the next iteration given this deficiency. The local transition functions contain references to the error state in the kernel states that they operate on, but unless the error state and quiescent state are degenerate, it should not be possible for the error state to be the result of the operation of the local transition function on a kernel state, whether or not it contains an error state.

If the error state is \( \epsilon \) and the alphabet of the automaton is \( W \) and contains the error state, then we distinguish a second set of states, \( W' \) which is the alphabet of the automaton which we are actually trying to model with the system \( \epsilon \in W' \), but \( \epsilon \notin W' \). If the local transition function is \( F \), and the neighbourhood state is \( h(\alpha) \), and \( \upsilon(\alpha) \) is the current state of the cell, then if

\[
h(\alpha) = \{ \upsilon(\alpha), \upsilon(\alpha + \delta_1), \ldots, \epsilon, \ldots \upsilon(\alpha + \delta_n) \}
\] (24)

then
\[ F(S, \upsilon(\alpha)) \in W' \quad (25) \]

But also the cell which is in the error state, \( \varepsilon \), should be rebuilt, i.e. if

\[ h_\alpha(\alpha) = \{ \upsilon(\alpha), \upsilon(\alpha + \delta_1), \ldots, \upsilon(\alpha + \delta_n) \} \quad (26) \]

then

\[ F(S_c, \varepsilon) \in W' \quad (27) \]

assuming no additional errors in recall.

### 13.4 Rebuilding Subconfigurations

To rebuild a configuration from a subconfiguration we must recognise that part of the configuration is missing, or in an error state. However, the local transition function has only local communication and so the rebuilding of the configuration can only be done if elements are recognised at the scale of the neighbourhood radius. This severely limits the rebuilding that can be done since larger structures cannot be ‘seen’ by the individual cells unless the neighbourhood radius is increased, which then makes the local transition function less local. This inability to ‘see’ at long ranges becomes more intense if the number of elements of \( W \) is small, since then information is represented in a more diffuse form, in normal space rather than in token space, the same amount of information being represented across a number of cells, to which a cell \( \alpha \) may not have access, rather than being represented by a richness of cells.

Due to the need to recognise local elements of a configuration we cannot rebuild from a subconfiguration containing errors for an arbitrary automaton. The automaton must be designed to have this fault tolerance built into the local transition functions. This limits the general applicability of this method to only systems that are designed to be fault tolerant in this way, not ones that model physical behaviour (e.g. automata that model diffusion of gasses and so on).

The rebuilding rules will be designed to sustain a small subconfiguration (around an order of magnitude greater than the neighbourhood radius) despite errors. This is analogous to a small organism, rebuilding damage to it, and continuing to grow.

If the error state, \( \varepsilon \) is the same as the quiescent state there can be a side effects creating spurious subconfigurations, where there are simply elements of a configuration, rather than a subconfiguration with gaps in, due to the fact that the local transition rules that govern rebuilding occur over the range of the neighbourhood radius, which typically is 1. For this reason when seeding a cellular automaton space, definite subconfigurations that are designed to be self-sustaining and error-correcting should be used.
The ability to repair damage must be balanced against the probability of a large subconfiguration being seeded by a small subset of cells. There is a small, but finite, probability in a system relying on unreliable computing elements that any state will randomly appear. The smaller a subconfiguration that can produce a large configuration, the greater the probability that the small subconfiguration will appear spontaneously, and produce the large subconfiguration. This would be undesirable behaviour. Linear self-supporting subconfigurations have a greater probability of being created from a small subconfiguration since the number of links of a cell to others in its self-supporting subconfiguration will be few, given the comparatively short range of the neighbourhood radius. For this reason compact self-supporting subconfigurations (‘cellular automaton organisms’) are more able to support themselves without the danger of spurious subconfigurations (‘cancers’) appearing.

13.5 Empirical Data

A binary automaton with a Moore neighbourhood (a neighbourhood of size 8 with the same shape as that for Conway’s Life) was used. The transitions are set to support an 8 element ring, which supports itself, all other subconfigurations other than those that are a part of the ring are not supported.

In the simulations below there were 512 separate transitions, these transitions being anisotropic. The inputs were tupled with 8 random tuples of size 4. The neighbourhood codes were all orthogonal.

**FIGURE 39. The transitions for elements of the ring are of the form**

![Diagram showing transitions](image)

**Fig. 43 to Fig. 48** shows the automaton completing the ring when an element of the ring is ‘removed’ by setting its state to the quiescent state. This is subtractive noise. Also an element of additive noise is added at a position that is disjoint from the ring, and this is rejected.

13.6 Some Results - for continuation with multiple states allowed

13.6.1 Single subtractive and additive noise

Here we examine how the system behaves when cells are allowed to assume ‘multiple states’, i.e. states which are the superposition of more than one valid state from $W$, with more than $k_x$ bits set to 1.
The automaton has the following parameters: 8 nearest neighbours, 2 tokens (plus an error flagging token). 4 random tuples of size 8. Orthogonal neighbourhood codes. There are 512 transitions, based on Conway’s Life, only with modifications to support the formation of a ring (Fig. 42). The initial configuration contains an additional element of noise, that is disjoint from the supported ring, and an element missing from the ring. It requires 16 transitions to code for rebuilding of the ring.

**FIGURE 40.** Key for Fig. 43 to Fig. 48

- state 1
- quiescent
- multiple state

**FIGURE 41.** Target ring to be supported in Fig. 43 to Fig. 48

![Target ring](image)

**FIGURE 42.** Transitions required for completion of ring.

<table>
<thead>
<tr>
<th>from</th>
<th>to</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>13</td>
</tr>
<tr>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>12</td>
<td>16</td>
</tr>
</tbody>
</table>

NB: in Fig. 42 the cell in question for the numbered transition is highlighted in light grey.

**FIGURE 43.** Orthogonal classes- here no multiple states

![Orthogonal classes](image)

- element of additive noise rejected

**FIGURE 44.** run 1 (390 rules learned correctly) Class1: 2 set in 256

![Run 1](image)
In Fig. 43 the disjoint element of additive noise is easily rejected, since all transitions have been learned correctly as orthogonal class codes are used. Also the ring is rebuilt, bridging the single gap left by the subtractive noise.

In Fig. 43 to Fig. 48 examples with varying number of transitions correctly learned. Since most transitions are concerned with coding the majority of transitions required for Conway’s Life, the system can fail to learn many transitions without a large probability of those transitions concerned with rebuilding the ring are affected. The failure to learn the transitions will, however, affect the general behaviour of the system in other ways. In this sense the examples above are limited and possibly not conclusive. However, it can be seen that the ring is resilient to poor training of the set of transitions under subtractive noise.

13.6.2 Addition of a cell in an ambiguous state

In the following two examples an error is injected. Rather than an element of the ring being simply in the quiescent state, effectively ‘missing’, an element outside the ring is set to an ambiguous state, i.e. the code is the superposition of the code for the quiescent and non-quiescent states. Class codes are orthogonal.
From Fig. 49 and Fig. 50 we can see that it takes a number of iterations for a stable subconfiguration to recover from damage. For this reason another restriction on the maximum number of errors tolerable is the frequency in time. For noise to be tolerated then noise events should occur, on average, less often than the typical time it takes for a stable configuration to rebuild itself [18].

13.6.3 Subtractive noise of more than 1 element

The following examples (Fig. 51 to Fig. 56) show the ability of the automaton to recover when more elements of the ring are missing. In the examples the class codes are orthogonal.
As can be seen in Fig. 51 to Fig. 56 the simple transitions used for regenerating the ring fail when more than one contiguous element is missing. This is because the neighbourhood template has only a radius of 1, thus can only see order on a range of 1. If more than 1 contiguous element is missing the attempts to regenerate at the end are independent and unaware of the long range order, and so attempt to regenerate the whole from very limited information, yet the standard transitions for Conway’s Life are also present for all other transitions, hence the very messy growth. Such a naive set of transitions for regeneration can only hope to cope with single elements of noise.

To improve high order fault tolerance either requires a more in-depth set of transitions that do not suffer from competition with the underlying transitions (Conway’s Life in this instance), or the increase of the neighbourhood template radius to allow longer range order to be considered.
13.7 Conclusions

The automaton can recover from both elements missing from the ring, and elements that are in ambiguous multiple states. Both these errors may occur when using associative memories to model local transitions when the class codes used are not orthogonal. Either the wrong state may be recovered (here represented by elements of the ring set to the quiescent state) or in a unrecoverable multiple state. As can be seen from the examples above, even when the class codes of the system are not orthogonal, in simple cases a system designed to self-support subconfigurations can recover a full subconfiguration from a modified subconfiguration. The ability to recover from errors is highly dependent on the transitions necessary for recovery being learned correctly.

The ability to recover a full subconfiguration from a damaged subconfiguration allows a system to run with compressed class codes yet still recover from errors. However, large number of errors will still render the system inoperable since it will be impossible for a system to recover from massive errors affecting many elements of a stable subconfiguration without the transitions being such that damage to a subconfiguration may lead to uncontrollable growth of the subconfiguration, or the spawning of multiple copies, or more complex behaviour. The ability to recover from errors is, however, distributed and if the average distance between errors is greater than the typical size of the self-supporting subconfigurations then the self-supporting elements may remain stable. The distributed nature of the system and its fault tolerance means that a fault-tolerant system will be large, by an order of the size of the self-supporting elements.
14 Adaptive Systems and Inhomogeneity

This chapter describes how a system might learn required transitions to cope with errors, by allowing heterogeneity during a training phase.

14.1 Introduction

Brown [61] describes a system with competing rules. This idea can be adapted to form a basis for evolving additional transitions to cope with noise. A supervisor algorithm is required, and a copy of the cellular automaton in question that is not subject to error. At each iteration the states of each cell in the automaton must be compared with the reference system. In the event of an element of noise being caused by a cell in an ambiguous state, each cell with this faulty cell in its neighbourhood must learn a new transition between a neighbourhood containing the fault cell state and the correct new state for its cell, as predicted by the perfect reference system. Such additional teaching is easy for an associative memory system since additional transitions can be easily added by simply teaching additional transitions.

14.2 Learning

If a series of configurations $I \times I_t, I \times I_{t+1}, I \times I_{t+2}, \text{ etc.}$ are created with the desired behaviour, then it might be possible to use the learning properties of associative memories to learn the required local transitions.

If $c_{ij}$ is a cell in $I \times I_t$ with state $s_t$, and $h_{ij}$ is the neighbourhood state of $c_{ij}$, and the new state of $c_{ij}$ in $I \times I_{t+1}$ is $s_{t+1}$, then we let the associative memories in $c_{ij}$ learn the association

$$ (s_t, h_{ij}) \rightarrow s_{t+1} $$

Associations are learned for every $c_{ij} \in I \times I_t$, for every $t$.

During the teaching phase the automaton will be heterogeneous - each cell will tend to have learned a different set of transitions to other cells in the neighbourhood. Convergence to a single set of transitions will only occur given sufficient sets of example configurations. However, convergence is not guaranteed.

Convergence may happen to not occur if:

- A transition from one configuration to another may happen to violate the rules of cellular automata - e.g. the creation of a subconfiguration from a group of cells in the quiescent state.
• The local transitions required at a particular spatial point might be inconsistent from configuration to configuration (i.e. temporally), and local transition functions varying over time would be required.

• The local transition functions required might be inconsistent across the spatial dimensions of the configurations. A case in point is where an inhomogeneous automaton must be used, as described by a number of researchers, notably Vichniac [20][22], [14], [21], and Brown [61]. To ensure homogeneity every cell must learn an association for every possible set of neighbourhood states, i.e. every neighbourhood state.

In general, as noted by Toffoli, it is not possible to know how an automaton will progress given an initial configuration, and the converse is true, given a series of configurations it is not possible to know if an homogeneous automaton exists for them. This is related to the ‘Garden of Eden’ [57].

Whilst the configurations are presented in sequence, the sequential nature itself is not learned, but only the transitions required to move from one configuration to another. The cellular automaton as a whole is an associative memory, learning associations between configurations [50][14]. The ideal is to look for automata in which the associations learned by the individual computing elements, i.e. the cells, converge to homogeneity since this represents the highest generality of solution. If there is a heterogeneity the solution is only valid for an automaton of that particular size. For a solution with homogeneity it might be possible to assume that the local transition functions are valid for systems of other sizes.

Assuming a toroidal connectivity, and a Moore neighbourhood the individual transitions from one configuration to another are:

And so on, for configurations 1 and 2, etc.
Rule extraction itself by this method requires that the rules can be extracted from the system of associative memories that form the cells, and an abstraction placed on them to provide some symbolic representation of the transitions. Not all series of configurations will have homogeneous local transitions that can succinctly code the transformations between them.

14.3 Determining convergence

14.3.1 Recovering transitions

After training, the associative memories of a cell contain the data of all the transitions learned. If the class codes used are orthogonal then the transitions learned can be recovered by applying all possible neighbourhood states to the inputs, and recovering the outputs. It is possible for the transitions to be learned simply by examining, for each cell, for each pair of configurations, the transition required from the kernel state to the new state, not using associative memories. However, associative memories offer the advantage of partial matching, should the training data be insufficient to detail every possible transition. Thus associative memories may be able to determine an acceptable and convergent set of local transitions on partial data.

14.3.2 Convergence

Convergence occurs when all sets of transitions for all cells are identical.

14.4 Conclusions

Associative memory-based cellular automata offer the possibility of rule extraction in either a homogeneous or inhomogeneous system, provided sufficient training data is supplied. However, simply identifying the transitions using other methods may allow rule extraction. Associative memories do, however, offer the possibility of generalisation via partial matching and so a system of associative memories, through this generalisation, may allow correct interpretation of the data without the explicit transitions being recoverable.

Time was not sufficient for further investigation of this problem.
15 Conclusions

The associative memory system of implementing cellular automata was compared against other systems based on two main criteria - how quickly it can perform iterations of an automaton, and how accurately it can do so.

15.1 Size of data

The size of the possible data sets examining cellular automata is enormous. The data presented in the thesis is empirical data, but given the size of the data an analytic system for the prediction of the behaviour of the systems is required to allow parameters to be chosen easily. The thesis concentrated on the empirical and no such prediction of behaviour was attempted. As was noted by Toffoli, predicting the behaviour of cellular automata is impossible in general. More analysis of the behaviour of the associative memories would be an advantage.

15.2 Speed

On the first of these criteria, the associative memories do not appear to have performed well for the systems chosen as bench marks. It may be the case that if the systems are scaled up to ones comprising a very large number of transitions then a benefit of associative memories over more traditional implementations may be seen. However time did not permit this investigation: large systems are extremely slow. Because of this time limit on the research the break-even point was not discovered.

A benefit of systems of associative memories over compiled languages is that the teaching stage is simpler than the compilation of extremely large sequential programs, but a preprocessing stage for such large sequential programs could be designed to eliminate this problem.

The introduction of specialised hardware [49][45] such as the CNNAP [33][39] and SAT [40][43][44] boards may offer a way for systems of associative memories to compete with sequential, compiled implementations. However, it is the creation of the inputs for ADAM for isotropic and isomorphic systems that appears to be the fundamental bottleneck. Smolensky [29][30] offers a high-speed route for creating those inputs, at least for the isotropic system.

15.3 Accuracy

There are three broad classes of associative memory in terms of how they respond to errors:
(1) those that depend on exact implementation, such as Conway’s Life, Langton’s Self-Reproducing Automaton, for which an error will lead to the wrong result, or worse, complete failure.

(2) those that are embodying some principle of physics, such as diffusion, for which small errors in the transitions will lead to a slightly incorrect answer, but not a total failure.

(3) systems which are fault tolerant.

As was seen in the discussions on Conway’s Life and Langton’s Self-Reproducing Automata, the system of associative memories performs quite badly when class codes are compressed, leading to an improvement in speed, but inaccurate recall. Systems of associative memories would seem to be unsuited to such systems except when the system may be so excessively large that approximation is the only way that any sort of modelling might be done in a reasonable time.

For systems which are naturally fault-tolerant associative memories might be applicable, provided that the transitions governing the fault tolerance are learned. Probabilistic heterogeneity offers a way to lessen the impact of the errors in learning the transitions correctly.

15.4 Adaptive Systems

A great advantage of associative memories is their ability to learn, and this may be the great strength, as outlined in §14, conveying the ability to learn (in certain circumstances) the required transitions to progress between a series of configurations.

15.5 Further Work

- Further investigation of massive systems to see if there is a break-even point with non-associative memory systems in terms of speed of iteration.

- More investigation into naturally fault tolerant and adaptive systems.
• Statistical Parallelism [38][69] may offer a way to implement multiple cellular automata simultaneously using the same set of associative memories. It allows several computations to be carried out in parallel in a single associative memory, the required overhead for parallel implementation being such that the total time required divided by the number of strands of computation yields an effective time per computation strand that is less than if the computational strands were each implemented on separate associative memories. If this is extended to cellular automata it is possible to implement several on the same set of memories, which requires less processing power than implementing each automaton separately, on separate processors with separate associative memories.

• Investigation of the integration of the associative memories with dedicated ADAM hardware [40][43][44].

15.6 Final Word

Finally, the performance of the system of associative memories has mostly been disappointing, apart from the possibility of learning transitions. However, using ADAM presents the possibility of using specialised hardware, and statistical parallel techniques, which may offer significant speed ups not available from other techniques, and it is these areas that promise may be shown. There was insufficient time in this project to examine these areas, but this area has been outlined by the comparative lack of success in implementations that are presented in this thesis.
List of Technical Terms

**ADAM** Advanced Distributed Associative Memory
is a form of associative memory using two correlation matrix memories and a class code.

**Alphabet**
is the set of possible states the cells in the automaton may take.

**Anisotropic**
describes the case when the state and position of each cell in the neighbourhood is important.

**Associative Memory**
a memory that learns associations between input and output patterns.

**Automaton**
is a collection of cells in some sort of grid or other arrangement.

**Cell State**
the state of the cell in an automaton, this being updated on each iteration of the automaton.

**Cell**
the smallest element of a cellular automaton.

**Class Code**
a binary code belonging to a set of class codes that all have a fixed Hamming Weight.

**CMAC**
Cerebellar Model of Articulation Control. A form of coding in which the coding behaves as thermometer coding for values within a certain range \((r-d \text{ to } r+d)\) close to a central value, \(r\), but departs as the integer to be coded, \(n\), departs from \(r\). Thus \(m\) integers can be represented in a code space that has vectors that are less than \((m+1)\) bits wide, but only linearly close to \(r\).

**Configuration**
is the description of the states of all the cells in the automaton.

**Conway’s Life**
a well known cellular automaton.

**Correlation Matrix Memory (or CMM)**
is a form of associative memory.

**Degeneracy**
occurs when different initial conditions result in the same final result.

**Global Transition Function**
describes the transition from one configuration to another.

**Hamming Distance**

The number of bits that need to be changed in one code \(c_1\) to change it to another code \(c_2\).

**Hamming Weight**

the number of bits set to 1 in a binary pattern or vector.

**Isomorphic**

describes the case when just the number of cells in the neighbourhood in certain states is important.

**Isotropic**

describes the case when the relative positions of the states of cells in the neighbourhood to one another is important, but their rotation relative to the central cell is not.

**Kernel State Code**

is a binary vector that represents the kernel state.

**Kernel State**

is the set of states in which the cells of the neighbourhood are, plus the current state of the cell.

**Kernel**

is the neighbourhood, plus the current state, and the new cell state depends on the state of cells in the kernel.

**Kronecker product**

the outer product of two matrices

**L-Max Thresholding**

is a thresholding process which requires that at least \(l\) bits must be set to 1.

**Langton's Self Reproducing Automaton**

a well known cellular automaton.

**Local Transition Function**

describes the way that the new cell state is calculated from the kernel state.

**Neighbourhood State Code**

is a binary vector that represents the neighbourhood state.

**Neighbourhood State**

is the set of states in which the cells of the neighbourhood are.

**Neighbourhood Template**

is the set of offsets that define the positions of the elements of the neighbourhood relative to the cell.

**Neighbourhood**

is the set of cells in the neighbourhood of a cell on which forms part of what the new state of the cell depends.
**Raw Class Code**

is the response from a correlation matrix memory.

**State Code**

is a binary vector representing the cell state.

**Teaching**

The process by which inputs and output pairs are associated by using a correlation matrix memory (CMM) or Advanced Distributed Associative Memory (ADAM).

**Thermometer Code**

A form of coding which translates a value into a binary code in which the distance of set bits from the zeroth bit corresponds to the value. E.g a code to represent $n$ (from a set of integers 0 to $m$) would have the $(n+1)$th bit set in a code of $(m+1)$ bits in length.

**Thresholding**

is a process to create a class code from a raw class code.
List of Symbols Used

\(x \oplus y\) represents the bitwise logical ORing operation on vectors \(x\) and \(y\).

\(x \otimes y\) represents the formation of the outer product of two matrices (also known as the Kronecker product).

\(N\) represents the size of a class code.

\(k, k_s\) represents the Hamming weight of a binary code.

\(l\) Hamming weight which a code must have (or exceed) after l-max thresholding.
16 References


University of Illinois Press, 1970


[31] J Austin, “The design and application of associative memories for scene analysis”, YCST 87/06, University of York, 1986


[38] J Austin, “Parallel Distributed Computation”, Proceedings of International Conference on Artificial Neural Networks, Brighton, 1992


[67] JD Eckhart, Reference Manuals for Cellang Computation System. Available by WWW from Rutgers University


[69] Song Yan, J Austin, “Bitwise/Bit Vector Computation in Statistical Parallelism”, submitted to IEEE Transactions on Neural Networks