Abstract

Cellular automata (CA) are an interesting computation medium to study because of their simplicity and inherently parallel operation. These characteristics make them a useful and efficient computation tool for applications such as cryptography and physical systems modelling, particularly when implemented on specialized parallel hardware. In this dissertation, we study a number of applications of CA and develop new theoretical results used for them. We begin by presenting conditions which guarantee that a composition of marker cellular automata has the same neighbourhood as each of the individual components. We show that, under certain technical assumptions, a marker cellular automaton has a unique inverse with a given neighbourhood. We use these results to develop a working key generation algorithm for a public-key cryptosystem based on reversible cellular automata originally conceived by Kari. We also give an improvement to a CA algorithm which solves a version of the convex hull problem, ensuring that the algorithm does not require a global rule change and correcting the operation in a special case. Finally, we study a modified version of an established CA-based car traffic flow model for the single-lane highway case, and use CA as a modelling tool to investigate the coverage problem in wireless sensor network design. We developed functional software implementations for all of these experiments.
Acknowledgements

First and foremost, I would like to thank my supervisor, Dr. Kai Salomaa, for all of his help, intuition, guidance, and support. The quality of this work would not have been nearly as high without all of his thoughtful and well-appreciated advice. Kiitos!

I would also like to thank Jarkko Kari and Sami Torbey for their advice and answers to my questions, which made the extension of their work clearer and easier for me.

Thanks to my family and friends for all of your support.
# Table of Contents

Abstract .............................................. i

Acknowledgements .................................. ii

Table of Contents ................................... iii

List of Figures ...................................... v

Glossary .............................................. ix

Chapter 1:

Introduction ........................................... 1
1.1 Background ......................................... 1
1.2 Formal Definition .................................. 3
1.3 Literature Survey ................................... 6
  1.3.1 CA and Cryptography ........................... 6
  1.3.2 CA Algorithms ................................ 8
  1.3.3 Car Traffic Modelling ......................... 9
  1.3.4 Modelling Wireless Sensor Networks ......... 11
1.4 Chapter Summary .................................. 14

Chapter 2:

A CA-based Public-Key Cryptosystem ........... 15
2.1 Background and Motivation ....................... 15
2.2 Preliminaries ..................................... 18
2.3 Theoretical Results ................................ 20
  2.3.1 Neighbourhood Size of Compositions ........ 20
  2.3.2 Reversibility ................................ 28
2.4 A Public-Key Cryptosystem ....................... 32
  2.4.1 The Key Generation Algorithm ............... 34
  2.4.2 Security Concerns and Practical Considerations 37
2.5 Summary and Future Work ....................... 40
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Radius-1, two-dimensional Von Neumann neighbourhood (left) and Moore neighbourhood (right)</td>
</tr>
<tr>
<td>1.2</td>
<td>The evolution of Conway’s Game of Life for a 5x5 ‘alive’ cell initial configuration. The black cells are ‘alive’ states and the white cells are ‘dead’ states. The first 12 evolutions are shown, and the last one is a steady state.</td>
</tr>
<tr>
<td>2.1</td>
<td>An example of an ((N, N'))-neighbourhood mapping to an (N') neighbourhood.</td>
</tr>
<tr>
<td>2.2</td>
<td>A graph of the highest mapping frequency for the 50 states of an example public key.</td>
</tr>
<tr>
<td>3.1</td>
<td>Rules for transitioning from the grey state (G) to the white state (W). All reflections and 90 degree rotations of these rules apply. GB means the cell can be either grey or black. Blank cells mean that the state of that cell is irrelevant to the rule.</td>
</tr>
<tr>
<td>3.2</td>
<td>An example of the first stage of the algorithm in action - the grey area is contracted around the black cells. Generations 0, 10, 40, and 90 are shown.</td>
</tr>
<tr>
<td>3.3</td>
<td>A problem with using only the rules in Figure 3.1. Two of the grey cells will both turn white in the next generation, disconnecting the grey area. Note that there is a white cell boundary around the CA which is not shown.</td>
</tr>
<tr>
<td>3.4</td>
<td>Rule for going from the grey state to the yellow state. This rule applies in the situation where we are not sure whether or not a grey cell should turn white or stay grey. Again, all reflections and 90 degree rotations of this rule apply, and blank cells mean that the state of that cell is irrelevant to the rule.</td>
</tr>
<tr>
<td>3.5</td>
<td>Rules for transitioning from the yellow state to the white or grey state. Y means the yellow state. Blank cells mean that the state of that cell is irrelevant to the rule.</td>
</tr>
</tbody>
</table>
3.6 The yellow states in action, allowing the grey area to stay connected in the same situation as was shown in Figure 3.3. ........................ 48
3.7 Rules for transitioning from the white state to the Finder state. All 90 degree rotations of these rules apply, but their reflections do not, since we only want the Finder state to move in a counterclockwise direction. F means the cell must be in the Finder state. Blank cells mean that the state of that cell is irrelevant to the rule. ................................. 49
3.8 Rules for transitioning from the black state to the Anchor state. '!B' means that the cell can be in any state except for the black state, and F denotes the Finder state. Blank cells mean that the state of that cell is irrelevant to the rule. ................................. 51
3.9 The first few generations of a small test case showing the Finder state identifying a particular black cell as the Anchor state. Note that the outer boundary of white cells (part of the initial configuration) is not shown. F1 and F2 refer to the two "dummy" finder states. .................. 51
3.10 T1-T8 are the possible states that the Anchor cell may transition to. T and R refer to transmitter and receptor, which are not states, simply cells which must be in the white state in order for the transition to occur, and whose behaviour after the transition changes. The anchor state will transition to state T1 if the transmitter and receptor are in the white state; if not, it will try to transition to T2, and so on. .... 53
3.11 An example of a white cell starting the counterclockwise communication. It notices that the T3 state is its upper-right neighbour, and so it transitions to the Communication state. ................................. 54
3.12 Rule for transitioning from the white state to a state corresponding to the total number of grey or black states in a cell’s neighbourhood. All 90 degree rotations of this rule apply. C refers to the communication state. Blank cells mean that the state of that cell is irrelevant to deciding whether this rule should be applied, but if the blank cells in this figure are in the grey or black state, then clearly they will influence which 'GB sum' state that the white state will transition to. ..... 55
3.13 An example of the communication states in action. Note that the T3 state here counts as a black cell in the computation of the 'GB sum' state transition. ......................................................... 56
3.14 A rule for changing from the white state to the Detect state. All rotations/reflections apply. ................................. 56
3.15 An example of the third stage of the algorithm working. The communication states are not shown here. Detect states are the same colour as the black state for simplicity. In this example the detection occurred at the bottom left black cell. Generations 344, 364, 384, and 406 are shown.

4.1 A small example of cars following the BJH model. The dots refer to empty cells, and the numbers represent the velocities of cars. Here the density $\rho = 0.2$, $p_{\text{fault}} = 0.1$, and $p_{\text{slow}} = 0.5$. Cars drive from left to right.

4.2 A ‘zoomed-out’ view of a larger simulation of the BJH model. Black dots refer to cars, while white space is empty road. Here the density $\rho = 0.15$, $v_{\text{max}} = 5$, $p_{\text{fault}} = 0.1$, and $p_{\text{slow}} = 0.5$. The road is 1000 cells wide and the last 1000 evolutions out of 2000 are shown (to reach a steady state). Cars drive from left to right, and time 0 is at the top.

4.3 A ‘zoomed-out’ view of a larger simulation of our ‘slow-to-stop’ model. Black dots refer to cars, while white space is empty road. The simulation parameters used to produce this output are the same as those used for Figure 4.2. Cars drive from left to right, and time 0 is at the top.

4.4 The ‘fundamental diagram’ for our model. Each point represents the result from the latter 1000 iterations out of 2000 iterations (to reach steady state) on a road of length 1000 starting from a random configuration. Car density was set from 0 to 0.8, in intervals of 0.02, and ten simulations were performed for each density. $p_{\text{fault}}$ was set to 0.1, and $p_{\text{slow}} = 0.5$. 

4.5 A fuel economy diagram comparing our model with the BJH model. Each point represents the result from the latter 1000 iterations out of 2000 iterations (to reach steady state) on a road of length 1000 starting from a random configuration. The same simulation parameters as in Figure 4.4 were used, but average car speed and average number of acceleration cycles per car were recorded instead.

4.6 The figures from the paper by Cunha et al. [22]. The legends for the top two graphs refer to the size of the grid (1000x1000 down to 250x250). The legend for the bottom graph refers to the CA rule used: the 1 means 01/01, the 4 is 01234/01234, and the 8 is 012345678/012345678.

4.7 Number of nodes alive over time with several different local rules. Each of the curves shows this statistic for a different CA rule.

4.8 Number of active nodes over time with several different local rules. Each of the curves shows this statistic for a different CA rule.
4.9 Percent coverage over time with several different local rules. Each of
the curves shows this statistic for a different CA rule.
Glossary

**45-Convex Hull**  The convex hull of a set of cells in a cellular grid, where the boundaries of the convex set must be composed of straight lines or perfect diagonals.

**CA**  Cellular Automaton: A CA acts on a lattice of individual cells, each cell having a set of neighbouring cells which are usually in close proximity to it. The CA starts at time \( t = 0 \) with each of the cells in one of their \( n \) possible states, where \( n \) is finite. This global state is known as the *initial configuration*. All cells have their own state transition function whose input is the states of their neighbours, and output is one of the possible states. At time step \( t \), each cell uses the states of its neighbours as input to the state transition function, and the output of the function is the cell’s new state at time \( t + 1 \). The entire CA updates synchronously in this fashion.

**Convex Hull**  Given a set of points in a real vector space, the convex hull is the minimal convex set containing all of the points. A set is convex if the line joining any two points in the set is completely contained within the set.

**FDM CA**  Fixed-Domain Marker Cellular Automaton. The local rule of an FDM CA acts on a cell \( c \) (in state \( s \)) in the following simple way: if the neighbours of \( c \) are in a state configuration corresponding to an element of \( A \), then the state of \( c \) on the next generation is \( f(s) \). Otherwise, the state of \( c \) does not change. An FDM CA is just a special type of marker CA where all of the patterns are mappings from the set of neighbourhoods to \( S \), hence the term ‘fixed-domain’.

**Marker Cellular Automaton**  A marker CA is defined by a permutation \( \phi \) of the state set, and a finite collection of patterns \( P_1, P_2, \ldots, P_k \) around the origin. For each cell \( c \), the local rule of the marker CA checks if any of the patterns \( P_1, P_2, \ldots, P_k \) is present as the neighbourhood of \( c \) at time \( t \). If so, the permutation \( \phi \) is applied to \( c \)'s state at time \( t + 1 \), and if not, then \( c \)'s state does not change.
Public-key Cryptosystem  In a public-key (or asymmetric-key) cryptosystem a user generates a key pair consisting of a public key and a corresponding private key. They may publish the public key wherever they like. Other users can then encrypt messages using the public key and send the ciphertext. Since the original user is the only person with the private key, they are the only party able to decrypt ciphertext that was encrypted using their public key.

RCA  Reversible Cellular Automaton: A CA $C$ for which there exists an inverse CA $C^{-1}$ such that $C^{-1}(C(I)) = I$ for any initial configuration $I$.

Secret-key Cryptosystem  In a secret-key (or symmetric-key) cryptosystem a single key is generated and is used in some way to scramble or encrypt the input message into ciphertext. The same key is used to obtain the message back from the ciphertext. Clearly the sender and the receiver must both know the key, and eavesdroppers must not know the key.

WSN  Wireless Sensor Network. A network of several autonomous nodes working together to accomplish some task, which is usually monitoring something that the nodes can sense and detecting or reporting the results to a local gateway or sink. Each node has a relatively small communication and sensing radius, and may only be able to communicate with its immediate neighbours (not directly with the sink). The nodes also have limited computational power and therefore cannot make any complex decisions.
Chapter 1

Introduction

In this chapter we provide some background information concerning cellular automata, followed by a formal definition. We also give a survey of some of the literature pertaining to the use of cellular automata for algorithm design as well as physical systems modelling, and finally provide a chapter summary for this dissertation.

1.1 Background

A cellular automaton (CA) acts on a lattice of individual cells, each cell having a set of neighbouring cells which are usually in close proximity to it. The two standard radius-1 \((r = 1)\) neighbourhoods in a two-dimensional cellular grid are shown in Figure 1.1. The lattice may be finite or infinite.

The CA starts at time \(t = 0\) with each of the cells in one of their \(n\) possible states, where \(n\) is finite. Only a finite number of cells may be in the non-blank state (see Section 1.2). This global state is known as the initial configuration. All cells have their own state transition function whose input is the states of their neighbours, and
output is one of the possible states. At time step $t$, each cell uses the states of its neighbours as input to the state transition function, and the output of the function is the cell’s new state at time $t + 1$. The entire CA updates synchronously in this fashion.

There are a few general terms that can be used to classify different types of cellular automata. A CA is said to be:

- **Non-uniform** if not all of its cells evolve using the same transition rule, or the transition rules change with time

- **Asynchronous** if it updates cells independently, i.e. it does not have the restriction that all cells must be updated together at each time step

- **Reversible** if the configuration at time $t + 1$ has only one possible past configuration at time $t$, for all $t$

- **Totalistic** if states can be identified with a finite set of integers, and the transition rules only depend on a total or sum of the neighbourhood states

A CA known as Conway’s Game of Life [31], devised by J.H. Conway, is a well
known example of a uniform and synchronous CA with a totalistic local state trans-
sition function. There are two states, 'alive' and 'dead', and the cellular lattice is
an infinite two-dimensional grid with each cell’s neighbourhood being the eight cells
which are immediately horizontally, vertically, and diagonally adjacent to it (refer to
Figure 1.1, Moore neighbourhood). The transition rules are fairly straightforward. If
a cell in the 'alive' state has exactly 2 or 3 neighbours which are also alive, it continues
to be alive. Otherwise it dies (changes to the 'dead' state on the next generation). A
cell in the 'dead' state changes to the 'alive' state only when it has exactly 3 neigh-
bours in the 'alive' state. See Figure 1.2 for an example of the evolution of the Game
of Life CA for a particular initial configuration.

The phenomenon known as emergence is attributed to CA which are composed
of very simple transition rules and initial configuration, yet their evolution exhibits
extremely complicated or intricate behaviour. More information can be found in [11,
23,27,69].

Cellular automata are sometimes studied from a purely theoretical perspective,
but more often they are used either in algorithms research (to develop fast parallel
algorithms) [1,57,68] or as a modelling tool for physical systems [7,18,22,32].

1.2 Formal Definition

The formal definition of cellular automata we give here is a rewording of the original
by von Neumann [64] and a later summarization by Benenson and Torrens [8].

A cellular space is defined as an infinite $k$-dimensional space of automata cells to-
gether with a neighbourhood relation defined on this space and a finite set of possible
states for each cell $S$. The set of states $S$ must include a distinguished 'blank' state.
Figure 1.2: The evolution of Conway’s Game of Life for a 5x5 ‘alive’ cell initial configuration. The black cells are ‘alive’ states and the white cells are ‘dead’ states. The first 12 evolutions are shown, and the last one is a steady state.
The neighbourhood of each cell is a finite list of neighbouring cells. The time basis of the system is synchronous \((t = 0, 1, 2, \ldots)\).

Each automaton in the cellular space is specified by a transition rule \(T\) which gives the state of a cell at time \(t + 1\) \((S_{t+1})\) as a function of its own state and some other input \((S_t\) and \(I_t\) respectively) at time \(t\):

\[
T : (S_t, I_t) \rightarrow S_{t+1}
\]  

(1.1)

The cellular space of automata is said to be a \textit{cellular automaton system} if the input to each automaton’s transition rule is defined by the states of the automaton’s neighbouring cells. Also, it is required of \(T\) that if a cell and all of its neighbours are in the 'blank' state at time \(t\), then the cell will be in the 'blank' state at time \(t + 1\).

Finally, a \textit{cellular automaton} is made up of a cellular automaton system together with an assigned initial state for each cell in the cellular space at time \(t = 0\). The initial state of the system may only contain a finite number of cells which are in states other than the 'blank' state. In this way it is ensured that only a finite number of state transitions are computed at any particular time step.

It is worthwhile to point out some generalities. For example, the neighbourhood relation and the transition function of each automata cell need not be the same over the whole space, though they usually are in practice. Also, the preceding definition is only the deterministic case; a nondeterministic cellular automaton could have multiple initial states, and a transition function which outputs a set of possible next states for each cell. A probabilistic version would have a probability distribution for each possible initial state as well as a probability distribution for each possible next state of each cell.
In practice, an infinite cellular space cannot be implemented, but in many applications the region of cells which are not in the ‘blank’ state does not grow larger than a certain finite size. Alternatively, boundary conditions may be employed: the cellular space must be of finite size, so at the boundaries, cells are allowed to have different transition rules or neighbourhoods. For example, the case where the neighbourhood ‘wraps around’, so that cells at the right border who have neighbours to their right look for their neighbours starting from the left boundary - a toroidal boundary condition.

1.3 Literature Survey

In this section, we discuss some of the previous work which involves using CA for designing fast parallel algorithms or for modelling physical systems.

1.3.1 CA and Cryptography

Some of the first work connecting cellular automata with cryptography was by Wolfram [67, 68]. He showed that a particular cellular automaton could be used as a fast pseudo-random bit generator. This random bit stream can be used as a symmetric key; the input message is XORed bitwise with the random stream, forming the ciphertext. The ciphertext can be XORed again bitwise with the same bitstream in order to obtain the original message. This system was analyzed by Meier and Staffelbach [46], who found an attack which reduced the size of the key space enough to render the system practically insecure.

Gutowitz presented a more sophisticated approach [35, 36], combining the results
from an irreversible CA with those from a reversible CA in several computation rounds for encryption/decryption. He claims that the system is resistant to an advanced method of attack known as differential cryptanalysis [10]. We could not find subsequent papers challenging this claim.

Several symmetric-key systems have been proposed more recently. The system by Tomassini and Perrenoud [62] used the same idea as Wolfram’s by XORing a random key bitwise with the message to obtain the ciphertext, but they generated the pseudorandom bit stream using more complex one- and two-dimensional non-uniform cellular automata. Bouvry et al. [13] continued this line of research, using an evolutionary computation technique called ‘cellular programming’ to choose suitable random bit generating cellular automata. Seredyński and Bouvry [57] outline a cryptosystem which essentially adds to Wolfram’s scheme by evolving the ciphertext using a one-dimensional reversible cellular automaton for several generations, further obscuring it. The plaintext is then recovered by reversing the cellular automaton until the original ciphertext is obtained, then XORing it again with the key. Angelescu et al. [3] use a combination of CA as a random number generator, together with another combination of CA which is periodic, so that one evolves the periodic CA for half a period to encrypt, and for the other half to decrypt. They term this scheme Hybrid Additive Cellular Automata (HACA) and it appears to be quite involved. However, somewhat similar algorithms by Nandi et al. [51] and Sen et al. [56] have been disputed [5,12].

Little research has been done toward the development of CA-based public-key cryptosystems. Only a few are available to be found: the Tao-Chen cryptosystem [61] and another system by Guan [34] both use non-uniform CA. A system was also
proposed by Kari [39], which is discussed in detail in Chapter 2.

1.3.2 CA Algorithms

Cellular automata have been used to design algorithms for classical parallel computation problems. A well-known example is the firing squad synchronization problem (FSSP), where at $t = 0$ some fixed number of cells in a line (the soldiers) are in a particular state, one cell on one far end of them is in a different state (the general), and the cells have a radius-1 neighbourhood in a one-dimensional cellular space. The question is how to design a uniform cellular automaton that acts on this cellular space in such a way that at some time $t$, all of the soldiers switch to a 'firing' state simultaneously. The FSSP was first published with a solution by Moore [47]. Later, a solution requiring only six states was found by Mazoyer [45], who also proved that a four state solution was not possible. Since then, several variations and generalizations of the problem have been discussed in the literature. Researchers have considered the 'soldiers' standing in various configurations in higher dimensions, and have investigated time and state constraints for these situations. Other constraints such as the use of reversible CA, the amount of information exchanged between adjacent cells, and in a practical setting a constraint on the number of faulty cells have also been studied. A recent, short summary can be found in the paper by Gruska et al. [33].

Another example of a classical algorithmic problem solved using CA is the majority problem: given a cellular space with the two states '1' and '0', an initial configuration with $i$ cells in the '1' state and $j$ cells in the '0' state, and a proportion $\rho$, design a cellular automaton which eventually changes all states to '1' if the initial configuration has $\frac{i}{i+j} > \rho$ or to '0' if the initial configuration has $\frac{i}{i+j} < \rho$. This definition is
fairly restrictive, and the solution by Gacs et al. [30] which solves a random initial configuration correctly about 80% of the time has not been improved upon very much. Land and Belew [44] were able to prove that no perfect algorithm exists to solve this problem with only two states. However, Capcarrere et al. [15] showed that if one relaxes the exit condition so that different output configurations are recognized, then a perfect algorithm does exist, which they give and prove its correctness.

A well-known problem in computational geometry is the convex hull problem. A discrete version called the 45-convex hull problem was investigated by Adamatzky [1] and later by Torbey and Akl [63]. We will discuss our contribution to this area in Chapter 3.

1.3.3 Car Traffic Modelling

The study of car traffic flow is an interesting topic which has been around for some time, and cellular automata are a viable medium for conducting simulations to help understand it better. The first such study was conducted by Nagel and Schreckenberg [49], who develop a simple stochastic CA model to simulate single-lane highway traffic. Essentially the model says that all cars follow the same basic transition rules, and then move $v$ sites at each time interval. They increase their velocity $v$ by 1 up to some limit as long as there are no cars $v$ spaces ahead of them, slow down to speed $i - 1$ if they see a car $i$ spaces ahead of them, and randomly slow down by one speed unit with some probability $p$. The authors observe nontrivial, realistic simulation, particularly the transition from laminar traffic flow to start-stop waves as density increases. The Nagel-Schreckenberg (NaSch) model has been studied extensively in [25, 48, 53–55, 65].
Another model developed by Benjamin, Johnson, and Hui (BJH model) \cite{9} is quite similar to the NaSch model, but with the addition of a 'slow-to-start' rule. That is, a vehicle which has come to a complete stop moves forward at its first available opportunity with probability $1 - p_{\text{slow}}$, and on the time step after that with probability $p_{\text{slow}}$. The authors used this model to study the effect of junctions on highways, finding that setting a speed limit near junctions on single lane roads can greatly decrease the queue length to enter the road.

Since almost all major highways have two lanes or more, several researchers have constructed multi-lane models for highway traffic. The first work in this area was done by Rickert et al. \cite{52}, who designed a working model based on the NaSch model. They noticed that checking for extra space when switching lanes ('look-back') is an important feature of their model in order to get the realistic behaviour of laminar to start-stop traffic flow. Wagner et al. \cite{66} design a two-lane simulation which accounts for a faster left lane which is to be used for passing. Using simple rules, they are able to obtain the realistic behaviour that at higher overall densities, the left lane has a higher density than the right one. They remark that this correct macroscopic behaviour is fairly easy to obtain using a CA model, and cite some failed attempts to simulate multi-lane traffic using other types of models. Knospe et al. \cite{42} study heterogeneous two-lane traffic and find that even at low densities, a very small amount of slower cars effectively cause both lanes to slow significantly. Also, they note that a system with mostly slow cars and a small percentage of fast cars is almost identical to a system with all slow cars. Finally, Nagel et al. \cite{50} summarize the existing lane-changing CA models and propose a general scheme according to which realistic lane-changing rules can be developed.
Esser and Schreckenberg designed a complete simulation tool for urban traffic in [26]. The model accounts for realistic traffic light intersections, priority rules, parking capacities, and public transport circulation. The simulation of large traffic networks can be performed in multiple real-time (i.e. a simulation of a day of traffic can be completed in less than half a day). Several other researchers have devised related schemes [6,14,19,24,58].

In Section 4.1 we study the addition of a ‘slow-to-stop’ rule to the BJH model. In the BJH and NaSch models, cars may drive up at maximum speed only to slow down in a very short distance behind a car that was stopped the entire time. In practice, most drivers notice stopped cars ahead of them, and do not follow this type of behaviour (although some do). We will investigate the differences and similarities between these two models.

1.3.4 Modelling Wireless Sensor Networks

Several researchers have attempted to use CA to model wireless sensor networks (WSNs). A WSN is a network of several autonomous nodes working together to accomplish some task, which is usually monitoring something that the nodes can sense and detecting or reporting the results to a local gateway or sink. Each node has a relatively small communication and sensing radius, and may only be able to communicate with its immediate neighbours (not directly with the sink). The nodes also have limited computational power and therefore cannot make any complex decisions, such as those that may be required by an efficient distributed routing algorithm. In many applications it is required that the nodes operate in a completely unsupervised environment.
These characteristics are indeed some of the defining qualities of cellular automata, and this fact has convinced many researchers that modelling WSNs using cellular automata is a reasonable thing to do. A paper by Cunha et al. [22] makes reference to features of WSNs such as low computational power, low bandwidth capacity, limited energy supply, and high cost of real WSNs (hence the need for simulation), as reasons for why the applicability of CA to simulate some aspects of sensor networks should be verified. In order to perform this verification, the authors choose to evaluate a topology control algorithm (an algorithm addressing the coverage problem) which can be simulated in a straightforward way using CA. They concluded that CA can be a worthwhile simulation tool for large WSNs, because it allows for the fast and objective verification of network properties such as degree of coverage and degree of connectivity.

Kwak et al. [43] design a self-organizing and energy efficient intrusion detection sensor system based on concepts from CA. The system wakes up sensors in waves that sweep across the network which are self-organizing and robust. It is shown that this system still works in cases of sensor failure, gaps in the distribution of sensors, and intelligent intruders. They remark that this behaviour combined with the simple local-rule logic forms an ideal infrastructure for this application.

In a WSN, certain nodes usually have different roles from most of the others; some nodes may act as simple sensors who report their data, while others may act as gateways or clusterheads, whose job is to aggregate data from a portion of the network. Frank and Römer’s paper [28] proposes a robust and efficient generic role assignment scheme for WSNs. They mention that this scheme is probably similar to an asynchronous cellular automaton.
Other researchers have attempted to apply CA to problems in mobile networking. Subrata and Zomaya [60] address the location tracking/management problem. They use CA combined with a genetic algorithm to create an evolving parallel reporting cells planning algorithm. The genetic algorithm is used to discover the 'best' CA transition rules for the application. Their results showed that the discovered CA rules describe near-optimal solutions to the reporting cells problem. In a related application, Kirkpatrick and Van Scy [41] investigate the use of CA to model message broadcasting in highly mobile ad hoc networks. They point out some strengths of the CA approach (discrete time steps useful in defining rapidly changing systems, and the approach is conducive to the design of a GUI), and derive an upper bound on the time required by a broadcast algorithm to distribute a message across a network.

Hochberger and Hoffman [38] use a cellular processing model to simulate the Lee algorithm for the routing of connections on printed circuit boards. A modification was made to the Lee algorithm so that in a CA implementation it only requires 14 states. They postulate that mapping classical algorithmic problems onto the CA model may lead to new ideas in their theory and implementation.

One design challenge faced in most WSN applications is the problem of coverage. Usually, designers do not want all the nodes in the network to be on at the same time, since there is no reason to monitor the same area several times over. To eliminate this redundancy, each node must somehow know when (and for how long) to turn itself off or go into low power mode. The question of how to manage this sleep/wake cycle while maintaining a high percentage of total monitored area for as long as possible is known as the coverage problem. We study some basic coverage schemes using a cellular automaton model in Section 4.2.
1.4 Chapter Summary

In Chapter 2 we explore the use of cellular automata as a medium for data encryption. After providing background information, we develop some new theory which enables us to complete Kari’s idea [39] for a CA-based public-key cryptosystem. A code listing of our Python implementation for this working system is given in Appendix A. This chapter is a more detailed version of a paper [20] which has been accepted for publication in the proceedings of the 3rd International Conference on Language and Automata Theory and Applications.

Torbey and Akl proposed a CA algorithm to solve the two-dimensional 45-convex hull problem [63]. In Chapter 3 we give a modification of their algorithm which does not require a global rule change, is more correct in a few special cases, and has a slightly simpler representation of transition rules. This chapter is a more detailed version of a paper [21] which has been accepted for publication in the Journal of Cellular Automata.

The benefits of simulation to study physical problems are clear: simulators can be cheaper, faster, and easier to modify than real-world experiments. We use CA as a modelling tool for car traffic and wireless sensor network simulation in Chapter 4, and show that conducting experiments or simulations for these types of systems is relatively easy to do using CA.

Finally, we summarize our findings and provide some ideas for future work in Chapter 5.
Chapter 2

A CA-based Public-Key Cryptosystem

2.1 Background and Motivation

Cryptography has been a part of our everyday lives for some time now. Most widely-used public-key encryption algorithms rely on advanced number theoretic results to achieve a high level of security, such as RSA (Rivest-Shamir-Adleman) encryption, whose security is believed to rely on the perceived hardness of the integer factorization problem. These systems tend to have relatively slow implementations [61], and since we will always want more efficient and secure encryption algorithms, it makes sense to consider alternate techniques. Cellular automata (CA) as a medium for encryption is an attractive idea in theory because most CA can be implemented on very fast hardware [17, 29, 70], hence a CA-based scheme may have the potential to encrypt and decrypt messages faster than existing techniques.

Let us very briefly describe the two types of cryptosystems. A secret-key (or
symmetric-key) cryptosystem consists of a single key which is used in some way to
scramble or encrypt the input message into ciphertext. The same key is used to obtain
the message back from the ciphertext. Clearly the sender and the receiver must both
know the key, and eavesdroppers must not know the key. Public-key (or asymmetric-
key) cryptosystems work differently: a user generates a key pair consisting of a public
key and a corresponding private key. They may publish the public key wherever
they like. Other users can then encrypt messages using the public key and send the
ciphertext. Since the original user is the only person with the private key, they are
the only party able to decrypt the messages that were encrypted using the public key.

Most investigations into CA-based cryptosystems have been aimed at traditional
secret-key systems [3,35,36,57,59,68]. There appear to be very few CA-based public-
key cryptosystems in the literature; one is the Finite Automata Public-Key Cryptosys-
tem, or Tao-Chen cryptosystem [61], and there was another attempt by Guan [34]
although they both use non-uniform CA. Kari’s paper [39] outlines an idea for a
public-key cryptosystem based on reversible cellular automata, and poses the ques-
tion of how to implement the key generation algorithm. We now review this paper in
some detail, as it is the main reference for our work.

The general objective of a public-key cryptosystem based on reversible cellular
automata (RCA) is to design an RCA that is very hard to invert without some secret
knowledge. That way, the RCA can be published and its inverse can be kept as
the private key. Kari emphasizes the importance that the RCA be at least two-
dimensional, since there exist algorithms to invert any one-dimensional RCA [2], and
also because of the following theorem.

Theorem 2.1.1. [40] It is undecidable if a given two-dimensional CA is reversible.
This is true even when restricted to CA using the von Neumann neighbourhood.

This theorem implies that, in general, the neighbourhood size of the inverse of a two dimensional RCA has no upper bound. This is good justification for the security of Kari’s public-key cryptosystem [39] in a theoretical sense. In a more practical sense, it is true that the decryption automaton must have a bounded neighbourhood size if one wants to actually store it and use it, however this certainly does not mean that the algorithm is necessarily insecure in this practical sense. The basic idea outlined in Kari’s paper was to compose together several simple and reversible ‘marker’ CA (which we define in Section 2.2) in order to form a more complex cellular automaton

\[ C = C_n \circ C_{n-1} \circ \cdots \circ C_1, \]

with inverse

\[ C^{-1} = C_1^{-1} \circ C_2^{-1} \circ \cdots \circ C_n^{-1}. \]

Encryption occurs by encoding the message as the initial configuration of the CA, then evolving the composed CA for some \( k \) generations to obtain the ciphertext. The inverse automaton does not need to be computed explicitly; one need only apply each component of the composition in succession. The inverse is then applied for \( k \) iterations to decrypt the ciphertext. The composition \( C_n \circ C_{n-1} \circ \cdots \circ C_1 \) is the public key, and each of the inverse automata of the composition \( (C_1^{-1}, C_2^{-1}, \ldots, C_n^{-1}) \) are kept as the private key. A well-constructed public key should be very hard to invert without knowledge of the components \( C_1, C_2, \ldots, C_n \) because the neighbourhood size of the composed inverse automaton would be quite large.
Kari’s paper [39] includes an example of a marker RCA composition with a 2-dimensional neighbourhood of 4 cells, and whose inverse has a 2-dimensional neighbourhood of 9 cells. The composition is made up of 5 very simple reversible marker CA. This is of course just an illustrative example, and Kari points out that longer and more complex (more states and a less restricted form) compositions would be needed in order to ensure security against brute force attacks. However, a public key with $s$ states and neighbourhood size $n$ requires $s^n$ entries in its local rule table, so it is essential to try to keep $n$ small so that the public key can be stored in reasonably sized memory.

The main issue preventing the practical implementation of Kari’s cryptosystem is the question of how to choose (or randomly generate) reversible marker CA such that the neighbourhood size of the composition remains small. In this paper, we give one possible answer to this question and investigate the resulting working cryptosystem.

We will state some preliminary assumptions and definitions before discussing our results concerning the composition of a class of marker CA in Sections 2.2 and 2.3. We give an algorithm\(^1\) for generating public and private keys in Section 2.4, and discuss practical implementation issues, security considerations, and ideas for future research in Section 2.5.

### 2.2 Preliminaries

In this work we assume that in a cellular array containing $M_1 M_2 \cdots M_d$ cells, where $M_i$ is the number of cells of each dimension for $i = 1, \ldots, d$, the neighbours of cells near the edge of the cellular array are determined by adding the component indices

\(^1\)Email Adam Clarridge for a working software prototype.
cyclically (modulo $M_i$). This is simply the toroidal boundary condition. The term ‘neighbourhood’ in this paper refers to either the pattern of cells around the cell, or the states themselves in the pattern, depending on the context.

Given a state set $S$, a ‘marker’ CA is defined by a function $\phi : S \to S$, and a finite collection of patterns $P_1, P_2, \ldots, P_k$ around the origin. For each cell $c$, the local rule of the marker CA checks if any of the patterns $P_1, P_2, \ldots, P_k$ is present as the neighbourhood of $c$ at time $t$. If so, $\phi$ is applied to $c$’s state at time $t+1$, and if not, then $c$’s state does not change. In some texts, it seems to be a requirement that $\phi$ be one-to-one (a permutation of the state set), but we are not sure whether this restriction is completely standard. Marker CA are analogous to ‘marker automorphisms’ [4] in the dynamical systems literature.

For technical reasons (in order to make the results clearer in the next section), we choose to create a new definition of marker CA which is perhaps more manageable. We define a ‘fixed-domain’ marker cellular automaton (or FDM CA) to be a five-tuple $(d, S, N, A, f)$ with dimension $d$, state set $S$, neighbourhood vector $N = (\bar{n}_1, \bar{n}_2, \ldots, \bar{n}_k)$, $\bar{n}_i \in \mathbb{Z}^d$ for $i = 1, 2, \ldots, k$, acting set $A \subseteq S^k$ with elements whose entries correspond to the positions defined by $N$, and a function $f : S \to S$. The local rule of an FDM CA acts on a cell $c$ (in state $s$) in the following simple way: if the neighbours of $c$ are in a state configuration corresponding to an element of $A$, then the state of $c$ on the next generation is $f(s)$. Otherwise, the state of $c$ does not change. An FDM CA is just a special type of marker CA where all of the patterns use the same neighbourhood, hence the term ‘fixed-domain’. Note that, conversely, an arbitrary marker CA can be represented as
an FDM CA by choosing \( N \) to be sufficiently large.

In the next section we give necessary and sufficient conditions characterizing change in neighbourhood size of compositions of FDM CA. For this purpose we do not need to assume that \( f \) is one-to-one, however, when an FDM CA composition is required to be reversible, it is necessary (though not sufficient to guarantee invertibility) for \( f \) to be one-to-one.

We use the terms ‘invertible’ and ‘reversible’ interchangeably when referring to cellular automata. Also, we define compositions of cellular automata in the following way: for any two cellular automata \( C_1 \) and \( C_2 \) acting on the same cellular grid, one generation of the CA \( C_2 \circ C_1 \) refers to the application of one generation of \( C_1 \) followed by one generation of \( C_2 \).

### 2.3 Theoretical Results

#### 2.3.1 Neighbourhood Size of Compositions

As we have noted above, for implementing a public-key cryptosystem based on compositions of RCAs, a desirable property is that the composition should have a small neighbourhood size. Here we give necessary and sufficient conditions that characterize the effect on neighbourhood size of composing an FDM CA with an arbitrary CA. For readability, we give the result first for a very restricted type of CA with a single cell neighbourhood. The underlying idea used for the general case (in Proposition 2.3.2) is similar but the notation is more complicated.

Let \( B \) be a CA with dimension \( d = 1 \), state set \( S \), neighbourhood \( N = (-1,0) \) (the cell to the left and the cell itself), and arbitrary transition function.
The state changes of $B$ can be described by a function $h_B : S \times S \rightarrow S$. If $s$ is the state of a cell $c$ at time $t$, then at time $t + 1$ the state of $c$ will be $h_B(s', s)$, where $s'$ is the state of the left neighbour of $c$ at time $t$.

For $s \in S$ we denote

$$\text{next\_state}_B(s) = \{h_B(s', s) \mid s' \in S\}.$$  

The set $\text{next\_state}_B(s)$ consists of all possible states that a state $s$ may directly transition into (depending on the left neighbour of $s$).

We want conditions which guarantee that the composition of $B$ with an FDM CA that has the same dimension, state set, and neighbourhood as $B$ has the same neighbourhood as $B$.

**Proposition 2.3.1.** Let $B$ be an arbitrary CA with dimension $d = 1$, state set $S$, a neighbourhood $N = (-1, 0)$ (the cell to the left and the cell itself) and transition function $h_B$. Let $D$ be an FDM CA $(d, S, N, A_D, f_D)$.

The composition $D \circ B$ has neighbourhood $N$ if and only if for all $s \in S$,

$$\exists s' \in S : f_D(h_B(s, s')) \neq h_B(s, s') \Rightarrow \text{next\_state}_B(s) \subseteq A_D \text{ or } \text{next\_state}_B(s) \cap A_D = \emptyset \quad (2.1)$$

**Proof.** Suppose that condition (2.1) holds. Consider two consecutive cells $c_1$ and $c_2$ that are in states $s$ and $s'$ respectively. We have to show that when applying the composition $D \circ B$, the next state of $c_2$ depends only on $s$ and $s'$. If $f_D(h_B(s, s')) = h_B(s, s')$, $D \circ B$ maps the state of $c_2$ always to $h_B(s, s')$. Next assume
that \( f_D(h_B(s, s')) \neq h_B(s, s') \). Now according to (2.1), next\_state_B(s) is either a sub-
set of \( A_D \) or it is disjoint with \( A_D \). In the former case, independently of the state of 
the left neighbour of \( c_1 \), \( D \circ B \) maps the state of \( c_2 \) to \( f_D(h_B(s, s')) \). In the latter 

case, again independently of the left neighbour of \( c_1 \), \( D \circ B \) maps the state of \( c_2 \) to 
\( h_B(s, s') \). Thus, we can compute the transition of \( D \circ B \) at cell \( c_2 \) knowing just the 
current states of \( c_1 \) and \( c_2 \).

Conversely, assume that (2.1) does not hold. This means that there exist \( s, s_1, s_2 \in S \) such that 
\[
h_B(s_1, s) \in A_D \quad \text{and} \quad h_B(s_2, s) \notin A_D,
\]
and that there exists an \( s' \in S \) such that \( f_D(h_B(s, s')) \neq h_B(s, s') \).

Consider three consecutive cells \( c_1, c_2, c_3 \) that at time \( t \) are in states \( s_1, s, s' \). Now 
in \( D \circ B \), the \( B \) automaton changes the states of \( c_2 \) and \( c_3 \) to \( h_B(s_1, s) \) and \( h_B(s, s') \)
respectively. Now since \( h_B(s_1, s) \in A_D \), the \( D \) automaton changes the state of \( c_3 \) to 
\( f_D(h_B(s, s')) \). So given the configuration \( s_1, s, s' \) at time \( t \), \( D \circ B \) maps the state \( s' \) 
to \( f_D(h_B(s, s')) \) at time \( t + 1 \).

On the other hand, if the states of \( c_1, c_2, c_3 \) at time \( t \) are \( s_2, s, s' \), in \( D \circ B \), the 
\( B \) automaton changes the states of \( c_2 \) and \( c_3 \) to \( h_B(s_2, s) \) and \( h_B(s, s') \) respectively. Since \( h_B(s_2, s) \notin A_D \), we know that the \( D \) automaton will not change the new state 
of \( c_3 \). So given the configuration \( s_2, s, s' \) at time \( t \), \( D \circ B \) maps the state \( s' \) to \( h_B(s, s') \) 
at time \( t + 1 \).

Since \( h_B(s, s') \neq f_D(h_B(s, s')) \), this means that the CA \( D \circ B \) does not have 
neighbourhood \( N \), since there is a dependency on the neighbour two cells to the 
left.

We now address the more general case, where \( B \) is an arbitrary cellular automaton
with state set $S$, neighbourhood $N_B = (\bar{n}_1, \bar{n}_2, \ldots, \bar{n}_k), \bar{n}_i \in \mathbb{Z}^d, \ d \geq 1$, and local transition function $h_B : S^k \rightarrow S$ ($h_B$ maps the neighbourhood of a cell to its next state).

Denote the set of all possible states of the neighbourhood $N_B = (\bar{n}_1, \bar{n}_2, \ldots, \bar{n}_k)$ of a cell $c$ by

$$S_{N_B}(c) = \{(s_{\bar{n}_1}, s_{\bar{n}_2}, \ldots, s_{\bar{n}_k}) \mid s_{\bar{n}_i} \in S \text{ for } i = 1, \ldots, k\},$$

where each $s_{\bar{n}_i}$ refers to the state of the cell in position $\bar{n}_i$. A $k$-tuple of $S_{N_B}(c)$ determines, according to the local transition function $h_B$, the state of the cell $c$ at the next time step.

The neighbourhood of the neighbourhood of a cell $c$ contains any cell that is a neighbour to one of $c$’s neighbours. Let us refer to this set as the second order neighbourhood of $c$. We will assume without loss of generality that each cell is a neighbour to itself, so each cell in the neighbourhood of $c$ belongs to its second order neighbourhood as well.

Denote the collection of all second order neighbourhoods of a cell $c$ with neighbourhood $s = (s_{\bar{n}_1}, s_{\bar{n}_2}, \ldots, s_{\bar{n}_k}) \in S_{N_B}(c)$ by

$$\bar{S}_{N_B}(s) = \left\{ \begin{bmatrix} t_{\bar{n}_1+\bar{n}_1} & t_{\bar{n}_1+\bar{n}_2} & \cdots & t_{\bar{n}_1+\bar{n}_k} \\ t_{\bar{n}_2+\bar{n}_1} & t_{\bar{n}_2+\bar{n}_2} & \cdots & t_{\bar{n}_2+\bar{n}_k} \\ \vdots & & & \vdots \\ t_{\bar{n}_k+\bar{n}_1} & t_{\bar{n}_k+\bar{n}_2} & \cdots & t_{\bar{n}_k+\bar{n}_k} \end{bmatrix} \in S^{k \times k} \mid \forall \bar{n} \in N_B, t_{\bar{n}} = s_{\bar{n}} \right\}.$$
s. The states in positions \( \bar{n}_1, \bar{n}_2, \ldots, \bar{n}_k \) are fixed (they are the states of \( s \)), while the rest of the second order neighbourhood is arbitrary.

The automaton \( B \)'s neighbourhood state changes for a cell \( c \) with neighbourhood \( s \) can be described by a function

\[
\bar{h}_B : S_{N_B}(s) \rightarrow S_{N_B}(c)
\]

which maps a second order neighbourhood to a first order neighbourhood of \( c \). The second order neighbourhood contains all the information needed in order to determine the first order neighbourhood of \( c \) on the next time step.

For the neighbourhood \( s \in S_{N_B}(c) \) we denote the set of all possible next neighbourhoods by

\[
\text{next_neighbourhood}_B(s) = \{ \bar{h}_B(\bar{r}) \mid \bar{r} \in S_{N_B}(s) \}.
\]

Now we can extend the result of Proposition 2.3.1.

**Proposition 2.3.2.** Let \( B \) be an arbitrary CA with dimension \( d \), state set \( S \), neighbourhood \( N_B \), and transition function \( h_B \). Let \( D \) be an FDM CA \( (d,S,N_B,A_D,f_D) \).

The composition \( D \circ B \) has neighbourhood equal to \( N_B \) if and only if for all \( s \in S_{N_B}(c) \),

\[
f_D(h_B(s)) \neq h_B(s) \Rightarrow \text{next_neighbourhood}_B(s) \subseteq A_D \quad \text{or} \quad \text{next_neighbourhood}_B(s) \cap A_D = \emptyset
\]

**(2.2)**

**Proof.** Suppose that condition (2.2) holds. We want to show that for all \( s \in S_{N_B}(c) \), where \( c \) is some cell, we do not need any more information than the neighbourhood \( s \) to compute the transition of \( D \circ B \). For neighbourhoods \( s \in S_{N_B}(c) \) such that the left
hand side of condition (2.2) is false, $D$ always maps $h_B(s)$ to itself. Clearly in this case $D$ does not create a dependence on a larger neighbourhood. Now consider neighbours $s \in S_{N_B}(c)$ such that both sides of the implication (2.2) are true. The right side of (2.2) means that all next possible neighbourhoods of $s$ must either be contained in the acting set of $D$, or completely separate from the acting set of $D$. So the particular neighbourhood that $s$ actually gets mapped to by $B$ is not important, since $D$ already knows from $s$ whether or not it will act. If next_neighbourhood$_B(s) \subseteq A_D$, then $D$ will apply $f_D$ to $h_B(s)$, and if next_neighbourhood$_B(s)$ is disjoint from $A_D$, then $D$ will apply the identity map to $h_B(s)$. Thus, we can compute the transition of $D \circ B$ at a cell $c$ knowing just the current states of its neighbours, $s$.

Conversely, assume that (2.2) does not hold. This means that for some $s \in S_{N_B}(c)$ there exist $\bar{r}_1, \bar{r}_2 \in \bar{S}_{N_B}(s)$ such that

$$\bar{h}_B(\bar{r}_1) \in A_D \quad \text{and} \quad \bar{h}_B(\bar{r}_2) \notin A_D,$$

and that $f_D(h_B(s)) \neq h_B(s)$. Recall that $\bar{r}_1$ and $\bar{r}_2$ agree on the states of $s$, and that $\bar{h}_B$ denotes that function that maps a second order neighbourhood to a neighbourhood of $B$.

Consider a collection of cells in the configuration of $\bar{r}_1$. When we apply $D \circ B$, the $B$ automaton changes the states of $s$ to a neighbourhood which is in $A_D$. The $D$ automaton is then applied. So the next state of the cell $c$ is $f_D(h_B(s))$.

On the other hand, consider a collection of cells in the configuration of $\bar{r}_2$. When we apply $D \circ B$, the $B$ automaton changes the states of $s$ to a neighbourhood which is not in $A_D$. The $D$ automaton applies the identity map. So the next state of the cell $c$ is $h_B(s)$. 
Since \( f_D(h_B(s)) \neq h_B(s) \), this means that the CA \( D \circ B \) cannot have the neighbourhood \( N_B \), since it depends on one or more of the states of \( \bar{r}_1 \) and \( \bar{r}_2 \) which differ and are outside of \( N_B \).

The condition of Proposition 2.3.2 can be used to inductively define a sequence of FDM CAs \( C_1, C_2, \ldots, C_n \) such that \( C_1 \circ C_2 \circ \cdots \circ C_n \) has the same neighbourhood as each of its components.

One interesting property of FDM CAs is that a carefully chosen composition can represent any cellular automaton.

**Proposition 2.3.3.** Every cellular automaton \( C \) with neighbourhood \( N_C \) of size \( k \) and state space \( S \) can be represented exactly by a composition of \( |S|^k + 1 \) FDM CAs with the same neighbourhood, if the FDM CAs are allowed \( |S| + |S|^k \) states.

**Proof.** We give a proof by direct construction. Consider a cellular automaton \( C \) with neighbourhood \( N_C \) of size \( k \). The automaton has a state transition function for updating the state of a cell \( c \),

\[
f_C : S^k \rightarrow S
\]

with \( |S|^k \) inputs. Enumerate all of these inputs by \( m_1, m_2, \ldots, m_n \), where \( n = |S|^k \), and let \( m_i[0] \) refer to the state of \( c \) in input \( m_i \). Define \( |S|^k \) new states by \( p_1, p_2, \ldots, p_n \). The FDM CAs with neighbourhood \( N_C \) of size \( k \) that will form the composition will use state space \( S \cup \{p_1, p_2, \ldots, p_n\} \), where the \( p \) states will be used to temporarily refer to states in \( S \).

We now describe the composition of the \( n + 1 \) FDM CAs \( D_{n+1} \circ D_n \circ \cdots \circ D_1 \) that will emulate \( C \). The first automaton in the composition, \( D_1 \), has acting set equal to the singleton \( m_1 \), and maps \( m_1[0] \) to the state \( p_1 \). Let \( m_2 = (m_1^1, m_1^2, \ldots, m_1^k) \). Then
the next automaton in the composition, $D_2$, has acting set

$A_{D_2} = \left\{ (s_1, s_2, \ldots, s_k) \in (S \cup \{p_1\})^k \mid \forall i:\right.

\begin{align*}
    s_i = p_1 &\Rightarrow m_i^2 = m_1[0] \\
    s_i \in S &\Rightarrow s_i = m_i^2
\end{align*}

\right\}$

and maps all states of $S$ to $p_2$. Note that the composition $D_2 \circ D_1$ has the same
neighbourhood $N_C$ of size $k$; if the neighbourhood of a cell of $C$ is in the configuration
of $m_2$, then by our choice of $D_2$'s acting set, we know we are guaranteed that $D_1$ will
map this configuration to an element of $A_{D_2}$. Also, if the neighbourhood of a cell is
not in the configuration of $m_2$, then we know that $D_1$ will not map the neighbourhood
anywhere in $D_2$’s acting set. So the condition of Proposition 2.3.2 holds, and we have
that $D_2 \circ D_1$ has neighbourhood $N_C$.

The next cellular automata in the composition up to $D_n$ have similar form. Let
$m_j = (m_j^1, m_j^2, \ldots, m_j^k)$, then for all $j$ up to $n$, $D_j$ has acting set

$A_{D_j} = \left\{ (s_1, \ldots, s_k) \in (S \cup \{p_1, \ldots, p_{j-1}\})^k \mid \forall i:\right.$

\begin{align*}
    s_i = p_1 &\Rightarrow m_i^j = m_1[0] \\
    s_i = p_2 &\Rightarrow m_i^j = m_2[0] \\
    \vdots
    s_i = p_{j-1} &\Rightarrow m_i^j = m_{j-1}[0] \\
    s_i \in S &\Rightarrow s_i = m_i^j
\end{align*}

$\right\}$

and maps always to $p_j$. In this way, $D_j$ performs the $j^{th}$ state transition, regardless
of other cells in the neighbourhood which are changed by other state transitions,
since they are changed to members of the temporary set of $p$ states. The composition
$D_n \circ D_{n-1} \circ \cdots \circ D_1$ has neighbourhood $N_C$, since each automaton which is added to the composition has an acting set which guarantees that the condition of Proposition 2.3.2 holds. The last automaton in the composition, $D_{n+1}$, has acting set equal to $(S \cup \{p_1, p_2, \ldots, p_n\})^k$ (it always acts), and applies the following mapping to the state $s$ of a cell:

$$f_{D_{n+1}}(s) = \begin{cases} f_C(m_i) & \text{if for some } i, s = p_i \\ s & \text{otherwise} \end{cases}$$

The automaton $D_{n+1}$ maps the temporary $p$ states back to the original state space $S$. The result is that the composition $D_{n+1} \circ D_n \circ \cdots \circ D_1$ mimics the behaviour of the $C$ automaton exactly for inputs which only contain states from $S$, because of the use of temporary states which store state transition information in a way that does not interfere with other state transitions.

The upper bounds of Proposition 2.3.3 are not meant to be tight at all in terms of the number of cellular automata required in the composition or the number of extra states required. An extension to this work could be to create an algorithm which, for a given cellular automaton, automates this construction and approximates or finds exactly the minimal number of automata and extra states required.

### 2.3.2 Reversibility

We now discuss the reversibility of FDM CA. In the following text we use the notation $s[0]$ to refer to the state in the ‘zero’ position of a neighbourhood vector $s \in S_N$.

**Lemma 2.3.4.** Let $C = (d, S, N, A_C, f)$ be an FDM CA, and assume the cell itself
is part of $N$ ($0 \in N$) without loss of generality. Denote

$$B = \{ a \in A_C \mid f(a[0]) \neq a[0] \}.$$ 

Then $(d, S, N, B, f)$ is equivalent with $C$.

**Proof.** The proof is immediate because tuples of $A_C$ not in $B$ do not affect the computation in any way. \qed

We say that an FDM CA $C$ with active set $A_C$ is *reduced* if for every $a \in A_C$, $f(a[0]) \neq a[0]$. By Lemma 2.3.4, without loss of generality we can assume that an arbitrary FDM CA is reduced.

We now generalize some of the definitions from Proposition 2.3.2 for an arbitrary FDM CA $C=(d,S,N,A_C,f)$.

Let an arbitrary neighbourhood be denoted by $N'$. Then let the $(N, N')$-neighbourhood of a neighbourhood $s \in S_N$ be the configuration containing the $N'$ neighbours of each of the elements in the $N$ neighbourhood. Let the set of all $(N, N')$-neighbourhoods of $s$ be denoted by $\tilde{S}_{(N,N')}(s)$. Note that an $N$-neighbourhood of an $N'$-neighbourhood is the same as an $N'$-neighbourhood of an $N$-neighbourhood, which can be more formally stated as follows. Let $r$ be a vector in $\tilde{S}_{(N,N')}(s)$ and let $r_{N'}$ be the ‘restriction’ of $r$ to the neighbourhood $N'$, that is, $r_{N'}$ is an $N'$-neighbourhood around the zero-position. Then the $(N', N)$-neighbourhood of $r_{N'}$ equals $r$.

Let the transition function of $C$ from $(N, N')$-neighbourhoods to $N'$ neighbourhoods be denoted by $\tilde{h}_C$, which takes an $(N, N')$-neighbourhood configuration and the neighbourhood $N'$ as input, and outputs $C$’s action with that configuration on the neighbourhood of size $N'$. An illustration of an $(N, N')$-neighbourhood and how
Figure 2.1: An example of an \((N,N')\)-neighbourhood mapping to an \(N'\) neighbourhood.

It maps to an \(N'\) neighbourhood is given in Figure 2.1.

Let the transition function from neighbourhoods to sets of possible output neighbourhoods be denoted by

\[
\text{next}_\text{neighbourhood}_C(s, N') = \{ \text{h}_C(\bar{r}, N') | \bar{r} \in S_{(N,N')}(s) \}.
\]

The following result characterizes when a given FDM CA with neighbourhood \(N\) has an FDM CA inverse with neighbourhood \(N'\).

**Proposition 2.3.5.** Let \(C\) be a reduced FDM CA \((d,S,N,A_C,f)\). Denote

\[
X = \bigcup_{a \in A_C} \text{next}_\text{neighbourhood}_C(a, N').
\]

Then \(C\) has an FDM CA inverse with state set \(S\) and neighbourhood \(N'\) if and only if

\[
(\forall a \notin A_C) \ f(a[0]) \neq a[0] \Rightarrow \text{next}_\text{neighbourhood}_C(a, N') \cap X = \emptyset.
\]
Proof. Assume condition (2.4) holds. Let us choose \( C^{-1} = (d, S, N', A_{C^{-1}}, f^{-1}) \) where \( A_{C^{-1}} = X \), and show that it inverts \( C \). Consider an arbitrary \( \vec{r} \in S_{(N,N')} \) of a \( \neighbourhood \ a \in S_N \), such that \( \vec{h}_C(\vec{r}, N') = b \in S_{N'} \). If \( a \in A_C \), then we know that \( b \in A_{C^{-1}} \) so \( C^{-1} \) will map \( b[0] \) to \( f^{-1}(b[0]) = a[0] \). Now consider the case where \( a \notin A_C \). In this case we know that \( a[0] = b[0] \). Since \( D \) is an inverse of \( C \), the function used by \( D \) must be \( f^{-1} \). Since \( b \in X \subseteq A_D \), the FDM CA \( D \) applies the function \( f^{-1} \) to \( b[0] \), but the result cannot be \( a[0] \) since that would imply \( f(a[0]) = b[0] = a[0] \), a contradiction. This means that \( D \) is not the inverse of \( C \). 

Conversely, assume that \( C \) has an inverse FDM CA \( D \) with \( \neighbourhood \ N' \) and let \( A_D \) be the active set of \( D \). Since \( D \) must correctly ‘map back’ all states where \( C \) applied the function \( f \), it is clear that \( X \) (as defined in (2.3)) is a subset of \( A_D \).

It remains to show that (2.4) holds. For the sake of contradiction assume that \( b = \vec{h}_C(\vec{r}, N') \in X \), where \( \vec{r} \) is an \( (N, N') \)-neighbourhood of a \( \neighbourhood \ a \notin A_C \), and \( f(a[0]) \neq a[0] \). Since \( a \notin A_C \), we know that \( a[0] = b[0] \). Since \( D \) is an inverse of \( C \), the function used by \( D \) must be \( f^{-1} \). Since \( b \in X \subseteq A_D \), the FDM CA \( D \) applies the function \( f^{-1} \) to \( b[0] \), but the result cannot be \( a[0] \) since that would imply \( f(a[0]) = b[0] = a[0] \), a contradiction. This means that \( D \) is not the inverse of \( C \). 

The following corollary addresses the uniqueness of FDM CA inverses.

**Corollary 2.3.6.** Let \( C \) and \( X \) be as defined in Proposition 2.3.5, and let \( C \) have some inverse \( C^{-1} \) with \( \neighbourhood \ N' \). Then \( C^{-1} \) is the only reduced FDM CA with \( \neighbourhood \ N' \) that inverts \( C \).

**Proof.** Any inverse of \( C \) must have function \( f^{-1} \). Assume for the sake of contradiction that there exists an FDM CA \( D \) with \( \neighbourhood \ N' \) that inverts \( C \) and has acting set \( A_D \neq X \). In the proof of Proposition 2.3.5 we have observed that \( X \) must be a
subset of $A_D$. Thus it is sufficient to show that there cannot be an element $b \in A_D$ with $b \notin X$.

Let $b = \overline{h}_C(\overline{r}, N')$, where $\overline{r} \in \overline{S}_{(N,N')}(a)$, $a \notin A_C$ since $b \notin X$. Then $D$ cannot be the inverse of $C$ because $C$ maps $a[0]$ to itself, but $D$ maps $b[0] = a[0]$ to $f^{-1}(b[0])$, which cannot be equal to $b[0]$ since $b \in A_D$ and $D$ is reduced.

\[\square\]

2.4 A Public-Key Cryptosystem

We want to use the idea of composing together many simple RCAs to form a complex RCA that is hard to invert, as outlined in the paper by Kari [39]. In order to make this idea work, we need to have some way to randomly generate a sequence of simple CAs such that the neighbourhood size of their composition remains small (or constant), and each CA in the composition is reversible.

We will demand that the neighbourhood size of each cellular automaton in the composition must be the same, and that the entire composition has the same neighbourhood as any of the components. The components will all be FDM CAs. Note that a composition of FDM CAs is not necessarily an FDM CA. Since the neighbourhood, state set, and dimension are fixed, we must design an algorithm which generates acting sets and transition functions for each of the $n$ components $C_1, C_2, \ldots, C_n$. From the theory in the previous section, we can now state some requirements for such an algorithm.

To maintain neighbourhood size during composition, the FDM CA $C_j$ must have an acting set $A_j$ and transition function $f_j$ such that the composition $C_j \circ (C_{j-1} \circ \ldots \circ C_1)$ has the same neighbourhood, for all $j \in \{2, \ldots, n\}$. Referring to the condition from Proposition 2.3.2, we need to guarantee that for each neighbourhood,
the next neighbourhood set of \( C_{j-1} \circ C_{j-2} \circ \cdots \circ C_1 \) is either completely contained in \( A_j \) or is disjoint from \( A_j \). Denote by \( T \subseteq S \) the “change set”, that is, the set of all states that the composition \( C_{j-1} \circ C_{j-2} \circ \cdots \circ C_1 \) can possibly change. One way we can be sure to retain neighbourhood size during composition is by setting \( A_j \) equal to the set of all neighbourhoods which contain a state in \( T \). The condition from Proposition 2.3.2 is satisfied since all neighbourhoods containing states in \( T \) will certainly be mapped (by \( C_{j-1} \circ C_{j-2} \circ \cdots \circ C_1 \)) to neighbourhoods which also contain states in \( T \) (assuming \( f_1, f_2, \ldots, f_{j-1} \) are one-to-one mappings), and neighbourhoods which do not contain any states in \( T \) will clearly be mapped to neighbourhoods which do not contain any states in \( T \). We use a less restricted version of this principle (which still satisfies the neighbourhood size preservation condition) in our algorithm to determine the acting set of each FDM CA in a composition.

The need for each of the FDM CAs in the composition to be invertible puts additional restrictions on their form. In order to be sure that the FDM CA is invertible, the set \( T \) which is used to find the acting set of each FDM CA must contain all states that the function \( f \) can change. The functions \( f_1, f_2, \ldots, f_n \) must also be permutations (one-to-one mappings). We discuss the key generation algorithm in more detail in Section 2.4.1.

Once the component FDM CAs are generated, the public key is determined by sequentially applying \( C_1, C_2, \ldots, C_n \) to each possible neighbourhood (using the neighbourhood as the starting configuration). The final state of the cell is recorded, and the public key is this mapping of neighbourhoods to states. The private key is not calculated explicitly; the CAs \( C_1^{-1}, C_2^{-1}, \ldots, C_n^{-1} \) are simply applied sequentially for decryption. The message is encoded in a \( d \) dimensional grid and is evolved for a fixed
number of iterations of the public key to produce the ciphertext. The ciphertext and number of iterations are sent as the encrypted message.

2.4.1 The Key Generation Algorithm

Our key generation scheme is given in Algorithm 1. We should note that in this algorithm, the random_element function returns a random element from a given set, the random function returns a floating point number between 0.0 and 1.0, the random_permutation function returns a random permutation mapping of a given set, and the random_binary function returns a random binary string of a given length. Note that the randomness in these functions probably must come from a pseudorandom number generator in most implementations, a possible security concern. Also, to be clear, ‘random’ in these functions means ‘uniform random’. The get_all_possible_neighbourhoods function returns all possible neighbourhoods given a state set $S$ and neighbourhood $N$.

Initially $T$ is a set $T \subseteq S$ of two random elements of $S$. The FDM CAs in the composition are then constructed in order from $C_1$ to $C_n$. During the generation of each $C_i$, with probability $p$ a new element from $S$ is added to the set $T$, and otherwise $T$ stays the same. The function $f$ is chosen for each FDM CA to be a random permutation of the set $T$, and $f$ applies the identity map to states in $S - T$. We should note that one should choose $n$ to be sufficiently large so that $T = S$ at some point (i.e. no states are left completely unchanged), but we will discuss this more later.

The only remaining task is to select the acting set. For each FDM CA in the
Input: State space $S$, Neighbourhood $N$, Number of FDM CA $n$, $0 < p, q < 1$
Output: Set of reversible FDM CAs $C_1, C_2, \ldots, C_n$

Initialization
$T \leftarrow \emptyset$
$T$.add (random_element ($S$))
$T$.add (random_element ($S-T$))
all_possible_neighbourhoods $\leftarrow$ get_all_possible_neighbourhoods ($S,N$)

for $i \leftarrow 1$ to $n$ do
    The following code determines $f_i$
    if random () $< p$ and $T \neq S$ then
        $T$.add (random_element ($S-T$))
    end
    $f_i \leftarrow$ random_permutation ($T$)

    The following code determines $A_i$
    binary_string $\leftarrow$ random_binary ($|N|$)
    $A_i \leftarrow \emptyset$
    for neighbourhood $\in$ all_possible_neighbourhoods do
        unchanging_neighbourhood $\leftarrow$ True
        for $j \leftarrow 1$ to $|N|$ do
            if neighbourhood [$j$] $\in$ T then
                unchanging_neighbourhood $\leftarrow$ False
                if binary_string [$j$] = 1 then
                    $A_i$.add (neighbourhood)
                    break
                end
            end
        end
        if unchanging_neighbourhood $=$ True and random () $< q$ then
            $A_i$.add (neighbourhood)
        end
    end
    $C_i \leftarrow \{ S,N,A_i,f_i \}$
end

Algorithm 1: The public-key generation algorithm, discussed in Section 2.4.1.
composition, a random binary string of length $|N|$ is chosen. Every possible
neighbourhood is then considered as a candidate element of the acting set. If the candidate
neighbourhood has a state which is an element of $T$ and is also in a position corre-
sponding to a ‘1’ of the binary string, then it is added to the acting set. Also, if
the neighbourhood contains only states which are not in $T$, then the neighbourhood
is added to the acting set with probability $q$. For example, consider the case where
$S = \{a, b, c\}$, $N = \{-1, 1\}$, $T = \{a, b\}$, and the random binary string is 01. Then the
neighbourhood $ca$ is a member of the acting set while $ac$ is not, and $cc$ is a member
of the acting set with probability $q$. Note that if the neighbourhood $N$ contains the
zero element, then clearly the case where the neighbourhood is added to the acting
set with probability $q$ is irrelevant since not even the state of the cell can change.

We now discuss the correctness of this algorithm, and begin by showing that the
condition for constant neighbourhood size during composition holds. Assume we are
attempting to determine the acting set of the $i^{th}$ FDM CA in the composition, $A_i$,
and let us first consider neighbourhoods which have at least one state in $T$. If a
neighbourhood is in $A_i$, then at least one element of $T$ occurring in the neighborhood
corresponds to a ‘1’ in the binary string. Since $C_i \circ C_{i-1} \circ \cdots \circ C_1$ is $T$-invariant
(states in $T$ are mapped to states in $T$), the neighbourhood will certainly be mapped
to a neighbourhood in $A_i$. On the other hand, if a neighbourhood is not in $A_i$, then
all occurrences of states in $T$ correspond to ‘0’ elements of the binary string. This
neighbourhood is mapped to a neighbourhood where states in $T$ also correspond to ‘0’
elements of the binary string, and hence it is mapped to a neighbourhood which is not
in $A_i$. Finally, if we consider a neighbourhood which contains no elements of $T$, then
clearly the condition of Proposition 2.3.2 is satisfied regardless of whether the neighbour-
hood is in $A_i$, or whether $N$ contains the zero element, since the neighbourhood must map to itself.

It remains to show that the condition for FDM CA reversibility holds for each $C_i$. Rather conveniently, the previous conditions actually allow (or demand) that $A_i$ is also the acting set of the inverse FDM CA. Since any addition to the set $T$ during the construction of each FDM CA happens before we choose $A_i$, we are guaranteed that elements of $A_i$ will be mapped to elements of $A_i$, and elements not in $A_i$ will not be mapped to $A_i$. So the condition from Proposition 2.3.5 also holds. Note that we could not be sure of this if $A_i$ was constructed with some $T$ that did not correspond with the states that $f_i$ changes.

### 2.4.2 Security Concerns and Practical Considerations

Since the FDM CA compositions follow a specific form and are not general two-
dimensional RCA, we cannot directly use Kari’s result [40] to justify the security of the system, and hence the security of this cryptosystem is largely unknown to us. However, we do not believe that straightforward brute force attacks will work. If one attempted to guess at a composition of FDM CAs which resulted in the same public key, there are many choices for each CA and there are $n!$ ways to arrange them, since $n$ is the number of CAs in the composition. One could also attempt to keep track of all global inputs and outputs for a fixed grid size in order to invert the composed CA. In this case the number of possible global configurations is $|S|^g$ where $g$ is the number of grid cells, so as long as the grid (the message) is relatively large this method will not work.
We also do not believe that the private key $C_1^{-1} \circ C_2^{-1} \circ \cdots \circ C_n^{-1}$ can be guessed very easily. Although we do not calculate it explicitly, this CA must have a fairly large neighbourhood because for each composition in the sequence, the condition from Proposition 2.3.2 does not hold in general. Each time $T$ changes during the generation of the FDM CA, the inverse automaton’s neighbourhood size may increase, and this can happen at most $|S| - 2$ times. So there is a computable upper bound for the neighbourhood size of the inverse, given $C_1, C_2, \ldots, C_n$, but for reasonably large $S$ and $d > 1$ this probably does not pose a security threat.

A user must choose the parameters of our algorithm with some care in order to prevent these brute force attacks and also to be able to encrypt and decrypt within a reasonable amount of time on a normal computer. One such setup might be $N = \{(0, 1), (1, 0)\}$ (the top and the right neighbours), $|S| \approx 40$, grid size $g \approx 500$, number of FDM CA in the composition $n \approx 100$, $p = q = 0.5$, and number of iterations $\approx 100$. These sizes can probably be increased significantly if the algorithm were implemented on specialized parallel hardware (especially the grid size and number of iterations). Unfortunately, because the size of the public key is on the order of $|S|^{|N|}$, a neighbourhood size of larger than 2 may be infeasible, since at least 20 or more states are required for security. We should note that the expected number of CA needed in the composition to just achieve $T = S$ is $(|S| - 2) \cdot 1/p$, and so $n$ should be chosen so that it is significantly larger than this quantity. If $n$ is too small, then the composition will only change states in $T$, and all elements in $S - T$ that occur in the original message will occur in the same places in the ciphertext.

One security issue related to the last point is that with our key generation algorithm as written, it is very easy for an attacker to determine which state was last
added to $T$. The public key will map this state to some other state regardless of the neighbourhood. Not much can be immediately done with this information, but perhaps it could be a starting point for a clever cryptanalytic algorithm to find each of the FDM CA in the composition in backwards order.

Let us attempt to explore this idea. We know that the last state added to $T$ is guaranteed to map to some other state 100% of the time. The state that was added to the set $T$ second-last will map to a particular state most (90-100%) of the time as well. This seems bad for security, but we need to look at a more complete picture. Let us refer to the percentage amount of the 'most commonly mapped to' state from a particular state as the highest mapping frequency for that particular state. Figure 2.2 shows this statistic for a typical example of a public key which used 50 states, the top and right immediate neighbours as a cell’s neighbourhood, 250 FDM CA in the composition, and algorithm parameters $p = 0.25$, $q = 0.5$. The data is sorted by descending highest mapping frequency. We must point out that these percentages are by no means probabilities - the public key we used as an example is deterministic, and an attacker still may be able to exploit its structure to deduce more information. However, the fact that many of these percentages are fairly low is good, since it means that many states have 'unpredictable' transitions. This combined with the fact that the CA can be evolved for several hundred iterations (and this is a computationally inexpensive operation on parallel hardware) speaks well against the notion that the cryptosystem might be vulnerable to a statistics-based attack. If one still believes that the states with a high 'highest mapping frequency' are a weakness, then one could simply include redundancy in the form of several 'dummy' states in $S$ which are not allowed to be in the plaintext, and are added last to $T$. This way, the 'dummy'
Figure 2.2: A graph of the highest mapping frequency for the 50 states of an example public key.

states have a high 'highest mapping frequency', but do not help to directly decipher any plaintext.

2.5 Summary and Future Work

We presented conditions which guarantee that compositions of fixed-domain marker cellular automata have the same neighbourhood as each of the individual components. We showed that, under certain technical assumptions, an FDM CA has a unique inverse with a given neighbourhood. We used these results to design, present, and show the correctness of a working key generation algorithm for a public-key cryptosystem originally conceived by Kari [39]. We also provided some preliminary cryptanalysis and gave some practical implementation notes.
This work provides several avenues for further research. We have given perhaps a more manageable definition of marker cellular automata, which could facilitate or help with additional theoretical development in related areas. The security of the cryptosystem presented in this work is largely unknown, and in order for cryptanalysis to be possible the various algorithm parameters, key storage and generation procedures, and fully optimized implementation specifications must be fixed and explicitly stated. Also, there may be some alternate or more general way to choose the acting sets of each CA in the composition, which could result in a more secure or efficient system. If the cryptosystem does not break easily then it would make sense to try to design an optimal hardware implementation and to do a corresponding feasibility analysis for real-world applications. On the other hand, if the cryptosystem does not stand up to cryptanalysis then the weakness should be studied and understood so that perhaps some alternate, superior scheme using Kari’s general approach could be conceived.
Chapter 3

CA and the Convex Hull Problem

3.1 Overview

Given a set of planar points, the two-dimensional convex hull problem is to find the convex polygon with the smallest possible area which completely contains all of the points. This problem has been solved efficiently using standard methods, and an $O(n \log h)$ algorithm exists [16], where $n$ is the number of points and $h$ is the number of vertices of the convex hull. We present a cellular automata based algorithm to solve a version of the convex hull problem which can be represented exactly on a finite grid like a two-dimensional cellular automaton. It is known as the 45-convex hull problem: the polygon forming the convex hull must be composed of only horizontal lines, vertical lines, or 45 degree perfect diagonals.

Our algorithm is based on the cellular automaton algorithm described in the paper by Torbey and Akl [63]. They solve the 45-convex hull problem using a 3-state, Moore neighbourhood cellular automaton with semi-totalistic transition rules. The algorithm requires a global transition rule change after a certain number of
generations, which depends on the size of the grid. The process is thereby separated into two distinct stages, and so the algorithm technically does not solve the problem with a standard (uniform) two-dimensional cellular automaton. We define several additional states and transition rules that enable us to be certain that the first stage has finished, and hence we do not require a global rule change to transition to the next stage of the algorithm. We also define simpler transition rules for the first stage of the algorithm that correct errors that occurred with certain special types of input configurations.

3.2 The Algorithm

In this section we will discuss the specifics of the proposed algorithm. The algorithm has three main components or stages:

- Using 4 states and starting with an initial configuration that has black cells marking the input points and grey cells everywhere else with a white cell border, the grey area shrinks so that it is completely contained within the 45-convex hull of the black cells.

- Using 17 additional communication states, the algorithm identifies a point in time where it can be sure that the grey area has finished shrinking.

- Using one more state, a slightly modified version of the technique proposed by Adamatzky [1] is employed, and the shrunken grey area expands to the 45-convex hull of the input points.

The initial configuration is all grey cells except for a white cell outer boundary and any number of black cells, which are the input points. Note that the white states
at the boundary could be simulated by standard CA rules that are applied at the edges of the (finite) cellular array. Having white states at the boundary spares us the trouble of defining separate boundary conditions for the simulations.

The first and third stages of the algorithm actually work toward finding the 45-convex hull: the first stage contracts the grey area so that it is completely contained within the 45-convex hull of the black cells in such a way that the third stage expands the shrunken grey region to the exact 45-convex hull. The only function of the second stage is to detect that the first stage is in fact complete. It runs in parallel with the first, and uses a set of communication states (which do not interfere with the operation of the first stage) in order to verify that the first stage is complete.

Our implementation has longer running time on average and uses many more states than the one proposed by Torbey and Akl, but does not require a global rule change at any time. Also, our algorithm addresses the problem of a set of special cases which their algorithm does not solve correctly.

3.2.1 Stage 1

There are initially only white, grey and black cells on the grid for this stage. An example input is depicted in Figure 3.2. The black cells represent the points for which we are trying to find the 45-convex hull. In this stage we want to 'shrink' the grey area (make grey cells transition to white cells) in such a way that the grey area is completely contained within the 45-convex hull of the input points. Later on, in the third stage of the algorithm, the grey area expands to cover exactly the 45-convex hull.

The challenge is to come up with a set of simple local state transition rules for
CHAPTER 3. CA AND THE CONVEX HULL PROBLEM

Figure 3.1: Rules for transitioning from the grey state (G) to the white state (W). All reflections and 90 degree rotations of these rules apply. GB means the cell can be either grey or black. Blank cells mean that the state of that cell is irrelevant to the rule.

the CA so that the grey area shrinks smaller than the 45-convex hull of the set of input points, but stays connected. In the Torbey-Akl algorithm, a complex set of pseudo-totalistic (meaning that the input to the state transition function depends on the sum of certain subsets of the neighbour set, not the entire neighbour set) transition rules almost achieved correct behaviour in all cases. We found that a much more simply represented set of rules achieves the same result: all rotations and reflections (mirror images) of the two rules shown in Figure 3.1 cause the first stage to be completed correctly in almost all cases. We must note that this stage of the algorithm treats the states introduced in the second stage of the algorithm (Finder state, Communication states) as if they were the white state, since they do not affect the grey area’s shrinking. In Figure 3.2, one can see these rules in action.

However, there are some extreme special cases usually involving a very small number of input points where using just these rules can cause a disconnected grey area. A local cell configuration depicting this problem is given in Figure 3.3.

The algorithm proposed by Torbey and Akl has the same problem. We resolve this issue by introducing a fourth state, which we will refer to as the yellow state. This state is best understood as an intermediate state between grey and white. Yellow states exist in order to signify a temporary lack of local knowledge as to whether a
CHAPTER 3. CA AND THE CONVEX HULL PROBLEM

Figure 3.2: An example of the first stage of the algorithm in action - the grey area is contracted around the black cells. Generations 0, 10, 40, and 90 are shown.

Figure 3.3: A problem with using only the rules in Figure 3.1. Two of the grey cells will both turn white in the next generation, disconnecting the grey area. Note that there is a white cell boundary around the CA which is not shown.
CHAPTER 3. CA AND THE CONVEX HULL PROBLEM

Figure 3.4: Rule for going from the grey state to the yellow state. This rule applies in the situation where we are not sure whether or not a grey cell should turn white or stay grey. Again, all reflections and 90 degree rotations of this rule apply, and blank cells mean that the state of that cell is irrelevant to the rule.

Figure 3.5: Rules for transitioning from the yellow state to the white or grey state. Y means the yellow state. Blank cells mean that the state of that cell is irrelevant to the rule.

particular grey cell should transition to white or stay grey, so a cell in the yellow state will either transition to the white state or the grey state on the next generation. If a grey cell has neighbouring states corresponding to any rotation/reflection of the rule in Figure 3.4, then it will transition to the yellow state. Note the white cell in the bottom right corner; this is the only way that the situation in Figure 3.3 can occur.

The transition rules for yellow cells are shown in Figure 3.5. These rules contain a tie-breaking protocol so that if two yellow cells are adjacent to each other, one will turn white (topmost or leftmost) and the other grey (bottommost or rightmost). It
Figure 3.6: The yellow states in action, allowing the grey area to stay connected in the same situation as was shown in Figure 3.3.

should be clear that in cases where two yellow cells are adjacent to each other, it does not matter which one turns white and which one turns grey since the grey area will definitely stay connected, and both cells in question must be inside the 45-convex hull of the input points.

The problem shown in Figure 3.3 is resolved by the yellow states, as shown in Figure 3.6.

### 3.2.2 Stage 2

We want to make sure that there are no grey areas still shrinking before the third stage of the algorithm begins. If we can construct ‘Communicator’ states which travel in one direction (counterclockwise, say) along the edge of a grey area only if it is not shrinking, and can somehow detect when they have traveled once around a stationary grey area, then this detection is the signal to begin the third stage of the algorithm.

We should note that we require some room for the communication cells to propagate along; therefore we impose the restriction that no black cells (input points) be placed along the outer edge of the grey cells.

In order to facilitate this communication and detection, we want to start communicating counterclockwise from some cell that is guaranteed to be on the grey cell boundary when it has finished shrinking. Any of the outer black cells meet this
Figure 3.7: Rules for transitioning from the white state to the Finder state. All 90
degree rotations of these rules apply, but their reflections do not, since
we only want the Finder state to move in a counterclockwise direction.
F means the cell must be in the Finder state. Blank cells mean that the
state of that cell is irrelevant to the rule.

requirement. However, we do not want to impose the restriction that one must des-
ignate one of the outer black cells as this communication starter cell (let us refer to
it as the Anchor cell). We construct the algorithm to automatically 'find' one of the
black cells that is on the edge of the grey area and designate it as the Anchor cell.

We introduce a 'Finder' state which travels counterclockwise along the edge of a
(possibly shrinking) grey cell boundary. Finder states always transition to white cells.
White cells which are adjacent to Finder states change to Finder states in such a way
that there can be at most one Finder state in the automaton at any given generation.
The Finder state’s propagation along the grey boundary stops when it is adjacent to
any black cell, since when this happens it means that a suitable Anchor cell has been
found. The transition rules for white cells changing to the Finder state are shown in
Figure 3.7.

We must note that there are two 'dummy' states required to facilitate the initial-
ization of the Finder state; the first dummy Finder state starts in the top left corner
of the grid on the first generation, and on the next two generations it transitions to
the second dummy state and then to the Finder state. This was just a simple way
to have the Finder state start on the third generation in the top left corner, where
it will still be able to follow the shrinking grey area while not affecting the shrinking process in any way. The first dummy state (part of the initial configuration) appears as a grey cell to its neighbours and the second dummy state appears as a white cell to its neighbours, so the behaviour of the first stage of the algorithm does not change. The first dummy state could be obtained from the CA boundary conditions applied at the upper left corner and hence this behaviour can be achieved on a standard two-dimensional CA.

When the Finder state is directly adjacent to a black cell, the black cell changes to the Anchor cell state. The conditions for this occurrence are shown in Figure 3.8. Note the tie-breaking rule; a black cell may see a Finder state directly above it, yet it will not claim itself the Anchor cell if the Finder state has another black cell to its immediate left or right. So in tie-breaking situations, the left and right cells 'win' the Anchor state. The white cells around this interaction also notice that a Finder is adjacent to a black cell and do not continue the Finder state’s propagation (this behaviour is implied by the previously mentioned rules of Figure 3.7). Note that it is impossible for a Finder state to ever be in between two black cells (i.e. to have a black cell on its left and right, or top and bottom) since a grey cell must be in such a location, and grey cells never transition to the Finder state. A small example of the Finder state successfully 'finding' an Anchor cell is given in Figure 3.9.

We should also note at this point that it is impossible for the grey area to shrink faster than the Finder state can follow it. This is because the Finder state always moves to be directly adjacent to grey cells, and any grey cell that turns white can turn to the Finder state on the next generation since it will have both the Finder cell and the grey cell as neighbours. One of the latter two rules of Figure 3.7 will apply.
Figure 3.8: Rules for transitioning from the black state to the Anchor state. ‘!B’ means that the cell can be in any state except for the black state, and F denotes the Finder state. Blank cells mean that the state of that cell is irrelevant to the rule.

Figure 3.9: The first few generations of a small test case showing the Finder state identifying a particular black cell as the Anchor state. Note that the outer boundary of white cells (part of the initial configuration) is not shown. F1 and F2 refer to the two ’dummy’ finder states.
Once the Anchor cell is found, the communication begins. The goal of the communication is to send a signal counterclockwise around the edge of the grey area, and this signal may only propagate if the grey edge it is travelling along is not in the process of shrinking. The Anchor cell changes to one of 8 states based on the white cells around it (the Finder state is considered a white cell as well, since it will be turning white on the next generation anyway). Each of these states defines a different transmitter and receptor, as shown in Figure 3.10. The transmitter and receptor labels are not states - they are simply there to clarify the mechanism for starting and stopping communication. The T and R labels just signify that white cells who notice one of T1-T8 in a certain location in their neighbourhood will recognize themselves as the transmitter or receptor and will therefore transition to an appropriate state. That is, when a cell D is in the white state and the cell to the lower right of D is in state T1, the cell D transitions to the communication state. The same happens if the state to the left of D is in state T2, and so on. Figure 3.10 attempts to explain the intuitive idea behind the construction, since if written out in full, the rules are slightly more complicated. The behaviour of our cellular automaton guarantees that any configuration can have only one occurrence of the states T1-T8, however, formally the rules for the white cell D need to define what happens in cases where more than one of the neighbours of D is in states T1-T8.

It does not matter which of the 8 states the Anchor cell chooses to transition to, as long as both the transmitter and receptor cells are white. So as soon as the Anchor state is found, it transitions to one of the T1-T8 states. We should note at this point that the algorithm prototype includes an explicit rule for transitioning to each of the T1-T8 states: the Anchor cell will transition to T1 if the top-left and top cells of its
Figure 3.10: T1-T8 are the possible states that the Anchor cell may transition to. T and R refer to transmitter and receptor, which are not states, simply cells which must be in the white state in order for the transition to occur, and whose behaviour after the transition changes. The anchor state will transition to state T1 if the transmitter and receptor are in the white state; if not, it will try to transition to T2, and so on.

neighbourhood are in the white state, but if this is not the case then it will transition to T2 if the left and top-left cells of its neighbourhood are in the white state. That is, the rule transitioning into T2 requires that the top cell is not white, and the top-left and left cells are both white. Otherwise it will try to transition to T3, etc. Because of the way the grey area shrinks, we are guaranteed to always be able to transition to at least one of T1-T8, since there will always be at least two adjacent white cells in the neighbourhood of the Anchor cell.

Depending on which state the Anchor cell changes to, the white cells around it know if they are supposed to start the transmission. The white cell who notices that it is the transmitter (say, if its top-right neighbour is in state T3) changes to the Communication state on the next generation. An example is given in Figure 3.11, which shows the next two generations of the computation continued from Figure 3.9.

The Communication state always changes to a state which corresponds to the sum of the grey and black neighbours it can count in its neighbourhood (the T1-T8 states count as a black state in this sum). From this state, if the number of grey
Figure 3.11: An example of a white cell starting the counterclockwise communication. It notices that the T3 state is its upper-right neighbour, and so it transitions to the Communication state.

or black (GB) neighbours has changed, it transitions to the white state. Otherwise, it transitions back to the Communication state. This ensures that Communication states cannot continue to propagate along a grey area that is still in the process of shrinking, because there will be changes in the number of GB neighbours along the edge. So the cells involved in communication are constantly flipping back and forth between their 'GB sum' state and the communication state.

The communication goes counterclockwise from the transmitter cell. The rule shown in Figure 3.12 defines the conditions under which a cell in the white state will transition to one of the 'GB sum' states, a state which corresponds to the total number of a cell’s GB neighbours. Note that we only need 4 of these 'GB sum' states, since if a cell has more than 4 grey or black neighbours it is clear that the first stage of the algorithm must not be complete, and therefore communication must not continue (the cell in question transitions to the white state). Also, if the cell has zero grey or black neighbours then it is clear that again, the first stage of the algorithm is not complete and the cell must transition back to the white state.

If the GB sum stays the same on the next generation, then the cell goes back
CHAPTER 3. CA AND THE CONVEX HULL PROBLEM

Figure 3.12: Rule for transitioning from the white state to a state corresponding to
the total number of grey or black states in a cell’s neighbourhood. All 90
degree rotations of this rule apply. C refers to the communication state.
Blank cells mean that the state of that cell is irrelevant to deciding
whether this rule should be applied, but if the blank cells in this figure
are in the grey or black state, then clearly they will influence which ’GB
sum’ state that the white state will transition to.

to the communication state, allowing the communication signal to propagate fur-
ther while ensuring that the edge of the grey area is not changing. If the GB sum
changes at all on the next generation, the cell will go from the GB sum state to the
white state. Continuing the example from Figures 3.9 and 3.11, the operation of the
Communication and GB sum states are shown in Figure 3.13.

The total number of states needed for this stage is 17; 3 states to find an Anchor
cell (2 dummies and the actual Finder state), 9 states for the Anchor cell (8 for the
various possible positions of the transmitter and receptor, and 1 is the actual Anchor
state), and 5 communication states (1 Communication state, and 4 ’grey/black sum’
states).

3.2.3 Stage 3

Once the communication makes it all the way around the static grey area, the Anchor
cell (which is in one of states T1-T8) notices that its receptor cell has changed to the
Communication state, and so it changes to a ’Detect’ state. All cells which are either
grey or black that have a detect state anywhere in their neighbourhood also turn
CHAPTER 3. CA AND THE CONVEX HULL PROBLEM

Figure 3.13: An example of the communication states in action. Note that the T3 state here counts as a black cell in the computation of the 'GB sum' state transition.

Figure 3.14: A rule for changing from the white state to the Detect state. All rotations/reflections apply.

into the Detect state. Communication type cells change to the white state if there is a Detect state in their neighbourhood. White cells in the presence of Detect states follow the rules outlined in the expanding stage of the algorithm discussed in Torbey and Akl’s paper [63] - Adamatzky’s method with a slight modification. The basic rule is that any white cell with 4 or more neighbours in the Detect state will transition to the Detect state. The modification Torbey and Akl make to this scheme is that all rotations and reflections of the rule in Figure 3.14 also call for a change to the Detect state - this rule simply catches the special cases where the grey area is connected only by a thin 'thread' of cells on a diagonal.
Figure 3.15: An example of the third stage of the algorithm working. The communication states are not shown here. Detect states are the same colour as the black state for simplicity. In this example the detection occurred at the bottom left black cell. Generations 344, 364, 384, and 406 are shown.

It should be clear that it is irrelevant to the correctness of the algorithm that the Detect states propagate outwards from a certain point instead of occurring all at once, globally. An example of this expansion of the grey area to the 45-convex hull is given in Figure 3.15.
3.2.4 Transition Function

In this section the complete state transition function for our algorithm is given. The state space is

\[ S = \{W, G, Y, B, F, F_1, F_2, A, T_1, T_2, \ldots, T_8, C, GB_1, \ldots, GB_4, D\}. \]

The state transition function \( f \) takes the form

\[ f : S \times S^8 \to S \]

where the first input is the state of the cell itself, the second is the two-dimensional Moore neighbourhood of the cell listed clockwise from the top neighbour, and the output is the next state of the cell. We define \( f \) below, for all \( s \in S \) and \( n \in S^k \).

\[
f(W, n) = \begin{cases} 
F & \text{if } n \text{ matches any reflection or 90 degree rotation of the configurations in Figure 3.7} \\
C & \text{if the cell is the ‘transmitter’ of } T_i \text{ (as described in Figure 3.10)} \\
GB_i & \text{if } n \text{ matches any 90 degree rotation of the configurations in Figure 3.12 (where } i \text{ is the number of } G \text{ or } B \text{ cells in } n) \\
D & \text{if } n \text{ matches any reflection or 90 degree rotation of the configuration in Figure 3.14, or} \\
& \text{if } n \text{ has 4 or more cells in the } D \text{ state}
\end{cases}
\]
\[ f(G, n) = \begin{cases} 
W & \text{if } n \text{ matches any reflection or 90 degree rotation} \\
Y & \text{of the configurations in Figure 3.1} \\
D & \text{if } n \text{ has 1 or more cells in the D state} 
\end{cases} \]

\[ f(Y, n) = \begin{cases} 
W & \text{if } n \text{ matches one of the configurations in Figure 3.5} \\
G & \text{otherwise} 
\end{cases} \]

\[ f(B, n) = \begin{cases} 
A & \text{if } n \text{ matches one of the configurations in Figure 3.8} \\
D & \text{if } n \text{ has 1 or more cells in the D state} 
\end{cases} \]

\[ f(F1, n) = F2, \ f(F2, n) = F, \ f(F, n) = W, \text{ for all } n \in S^8 \]

\[ f(A, n) = T_i \quad \text{where } i \in \{1, \ldots, 8\}, \text{ according to Figure 3.10} \]

\[ f(Ti, n) = D \quad \text{if the receptor of } Ti \text{ is in the C state (Figure 3.10)} \]

\[ f(C, n) = \begin{cases} 
GBi & \text{if there are } i \text{ G or B states in } n \ (i \in \{1, \ldots, 4\}) \\
W & \text{if } n \text{ has 1 or more cells in the D state} 
\end{cases} \]

\[ f(GBi, n) = \begin{cases} 
C & \text{if } n \text{ has exactly } i \text{ cells in the G or B states} \\
W & \text{otherwise} 
\end{cases} \]

Note that if none of the above rules apply, \( f \) maps a state to itself by default. We should reiterate that the states T1-T8 and the anchor state count as black states when counting the number of GB neighbours or when applying the shrinking rules from the first stage of the algorithm, and that the grey states consider Finder or
Communication states to be the same as white states in the first two stages of the algorithm. A working prototype of the algorithm is freely available\(^1\).

### 3.2.5 Time Complexity

Our algorithm has introduced many additional states to the Torbey-Akl algorithm [63]. We briefly compare the time complexity of the two algorithms.

In the following analysis we are assuming the input is given as a square grid having edges of length \( m \). In this case the time used by the algorithm from [63] is at most \( 3.5m \).

The shrinking phase of the algorithm from [63] uses time at most \( 2.5m \). In order to deal with the discussed problematic inputs, our algorithm sometimes uses yellow intermediary states when deciding whether to transition from a grey state to a white state. So using a very conservative estimate, the running time of Stage 1 is upper bounded by \( 5m \).

Stage 2 of the algorithm sends a communication signal around the grey area that is possibly still shrinking. Note that if the grey area continues shrinking, this disrupts the communication signal and potentially increases the time bound. But since we know that the shrinking must have stopped after \( 5m \) steps, when estimating the time needed for the communication phase, without loss of generality we can assume that the grey area has stopped shrinking. The length of the outer edge of the grey area is upper bounded by \( 4m \) which means that the communication signal takes time at most \( 8m \). (The signal alternates between the communication state and 'GB sum' states which means that it travels one cell in two cycles.) However, the communication signal

\(^1\)Contact Adam Clarridge by email.
travels slowest along diagonals (since it does not travel diagonally, only horizontally and vertically) so we must consider the case where there are two input points in opposing corners, causing the maximum amount of diagonal travel. In this case the algorithm would also take at most $8m$ cycles to communicate around the diagonal.

There is one last issue to consider for this stage: the Finder state may not find an Anchor cell by the time the first stage has finished. Experimental evidence as well as intuition seem to indicate that this can only happen if the first stage of the algorithm finishes very quickly, and hence this case would not change our upper bound for Stage 2. If any counterexample were to exist though, it would take time at most $m$ for the Finder state to find an Anchor cell. We conclude that Stage 2 of our algorithm takes time at most $9m$.

Finally, Stage 3 of our algorithm is very similar to the expanding stage in [63] which is completed in time $m$. The only difference is that our algorithm may take at most an additional $m$ cycles to distribute the Detect state throughout the grid.

By combining the above estimates, we note that the total running time of our algorithm is upper bounded by $16m$. Here we have just wanted to establish that the running time remains linear in $m$. By using a more detailed analysis, the upper bound estimate could clearly be improved. For example, the upper bound for Stage 1 corresponds to a situation where the grey area initially shrinks to be very small, whereas the upper bound estimate for the next two stages uses a worst case example where, even after the shrinking has stopped, the grey area remains relatively large.
3.3 Summary

In summary, the two-stage algorithm proposed by Torbey and Akl [63] has been modified so that it does not require a global rule change to transition between the two stages, has simpler rules for shrinking the grey area in the first stage, and keeps the grey area connected in a not-so-obvious special case, using 22 states in total. A mechanism has been described for finding an anchor point (a point on the edge of the grey area), communicating around the grey area in such a way as to ensure that it has not changed, and detecting this successful communication to enable a smooth transition to the last stage of the algorithm.
Chapter 4

Modelling Physical Systems with CA

4.1 Traffic Flow Modelling

Here we investigate a modification of the well-known Nagel-Schreckenberg (NaSch) \[49\] and Benjamin-Johnson-Hui (BJH) \[9\] CA models for single-lane highway traffic. These models are able to correctly capture several of the macroscopic characteristics of real traffic using very simple and computationally fast cellular automata, and as a result, have been studied extensively and incorporated into several complex traffic simulators (see Section 1.3.3).

The NaSch model is defined on a one-dimensional cellular space of \( N \) cells, usually with the toroidal (periodic) boundary condition. On a particular time step each cell either contains a car or is empty, and each car has an integer velocity \( v \) between 0 and \( v_{\text{max}} \) inclusive. Given some global configuration of cars at various velocities, the NaSch model dictates that cars are advanced along the road on the next time step.
according to the following rules, which are performed in order and in parallel for all cars. The quantity \( d \) is the distance in cells to the next car ahead.

1. Acceleration: if \( v < v_{\text{max}} \) and \( d > v + 1 \), then velocity increases \((v \leftarrow v + 1)\).

2. Slowing down (collision avoidance): if \( d \leq v \), then velocity decreases appropriately \((v \leftarrow d - 1)\).

3. Randomization: if \( v > 0 \), with probability \( p_{\text{fault}} \), velocity decreases by one \((v \leftarrow v - 1)\).

4. Motion: the car advances \( v \) cells.

These velocity rules implicitly do not allow collisions or overtaking.

The BJH model is a fairly straightforward extension of the NaSch model - the authors attempt to more accurately simulate the behaviour of drivers which have come to a complete stop in traffic jams on the highway. Cars which have velocity 0 either accelerate at their first available opportunity (as soon as there is an empty space ahead of them) with probability \( 1 - p_{\text{slow}} \), or on the time step immediately after that with probability \( p_{\text{slow}} \). Otherwise, they follow the NaSch model. This scheme is intended to reflect the fact that drivers take longer to accelerate from a complete stop, perhaps because they do not immediately notice the car ahead of them moving, or because of the slow pick-up of their car’s engine. So the BJH model is essentially the NaSch model with the addition of a 'slow-to-start' rule. An example of cars following the BJH model on a small road is given in Figure 4.1, and a more complete picture on a larger road for a longer period of time is given in Figure 4.2. In these examples, the initial configuration is a random placement of \( \rho N \) cars with velocity 1, where \( N \) is the size of the road in cells.
Figure 4.1: A small example of cars following the BJH model. The dots refer to empty cells, and the numbers represent the velocities of cars. Here the density $\rho = 0.2$, $p_{\text{fault}} = 0.1$, and $p_{\text{slow}} = 0.5$. Cars drive from left to right.
Figure 4.2: A 'zoomed-out' view of a larger simulation of the BJH model. Black dots refer to cars, while white space is empty road. Here the density $\rho = 0.15$, $v_{\text{max}} = 5$, $p_{\text{fault}} = 0.1$, and $p_{\text{slow}} = 0.5$. The road is 1000 cells wide and the last 1000 evolutions out of 2000 are shown (to reach a steady state). Cars drive from left to right, and time 0 is at the top.
We noticed that cars following these models behave in an unrealistic fashion when approaching a jam; if a car $B$ ahead has velocity 0, then a car $A$ may drive up to $B$ at velocity $v_{\text{max}}$ only to brake down to 0 velocity in one time step in the cell right behind $B$. This microscopically inaccurate behaviour may not be a big issue since these models are only meant to be macroscopically realistic in some ways, but we believe it could be interesting to explore the addition of a 'slow-to-stop' rule. That is, we want to modify the BJH model so that cars look farther ahead than $v$ cells and slow down earlier in certain situations. People typically pay attention to the velocity of the car directly ahead of them, so we use this information to aid in the decision of how much and when to slow down. A car’s change in velocity is then a function of its current velocity, the velocity of the car ahead of it, and the distance between them.

In our model, the cars’ velocities are adjusted at each time step according to the following rules. Recall that $d$ is the distance to the next car, $v$ is the velocity of the current car, $v_{\text{next}}$ is the velocity of the next car, $p_{\text{slow}}$ is the probability that the slow-to-start rule is applied, and $p_{\text{fault}}$ is the probability that the car slows down randomly. We fix $v_{\text{max}} = 5$.

1. Slow-to-Start: As in the BJH rule, if $v = 0$ and $d > 1$ then with probability $1 - p_{\text{slow}}$ the car accelerates normally (this step is ignored), and with probability $p_{\text{slow}}$ the car stays at velocity 0 on this time step (does not move) and accelerates to $v = 1$ on the next time step.

2. Deceleration (when the next car is near): if $d \leq v$ and either $v < v_{\text{next}}$ or $v \leq 2$, then the next car is either very close or going at a faster speed, and we prevent a collision by setting $v \leftarrow d - 1$, but do not slow down more than is necessary. Otherwise, if $d \leq v$, $v \geq v_{\text{next}}$, and $v > 2$ we set $v \leftarrow \min(d - 1, v - 2)$ in
order to possibly decelerate slightly more, since the car ahead is slower or the same speed and the velocity of the current car is substantial.

3. Deceleration (when the next car is farther): if \( v < d <= 2v \), then if \( v >= v_{\text{next}} + 4 \), decelerate by 2 \((v \leftarrow v - 2)\) - otherwise if \( v_{\text{next}} + 2 <= v <= v_{\text{next}} + 3 \) then decelerate by 1 \((v \leftarrow v - 1)\).

4. Acceleration: if the speed has not been modified yet by one of rules 1-3 and \( v < v_{\text{max}} \) and \( d > v + 1 \), then \( v \leftarrow v + 1 \).

5. Randomization: if \( v > 0 \), with probability \( p_{\text{fault}} \), velocity decreases by one \((v \leftarrow v - 1)\).

6. Motion: the car advances \( v \) cells.

These rules prevent collisions and overtaking. We now attempt to justify the second and third of these rules, which differ from the BJH model.

Consider the following scenario: a car with velocity 5 has a car 5 spaces ahead of it with velocity 0. The BJH model would change the car’s velocity to 4, and assuming the car ahead still has not moved, the car would be forced to decelerate to 0 on the next time step. Our model’s second rule decelerates the car to 3 in this case so that it is two spaces away, then on the next time step to 1 so that it is one space away, then finally to 0. We believe this is much more realistic behaviour, since cars which see a stopped car ahead of them would certainly attempt to slow down gradually. In less extreme situations, our model behaves the same way as the BJH model in terms of collision avoidance. Note that we are assuming for both models that the car ahead does not move and the randomization rule has not been applied.
Now consider another situation: a car with velocity 5 has a car 6 spaces ahead of it with velocity 0. The BJH model would not change the velocity of the car, resulting in a very sharp deceleration on the next time step as it decelerates from 5 to 0. Our model’s third rule decelerates the car to 3 so that it is 3 spaces away on the next time step, then the second rule decelerates the car to 1 so that it is two spaces away, then the car continues at 1 to the last space, then stops. Again, we believe that this type of gradual deceleration is typical of real drivers, and again we have assumed in this scenario that the car ahead does not move and that the randomization rule has not been applied.

Although both examples involved cars ahead which were stopped, the deceleration rules apply whenever a car is going significantly faster than the car ahead of it. While the car ahead with velocity 0 is the most illustrative case, the above examples could also be considered for different ‘car ahead’ speeds of 1 or 2.

An example of cars following our ‘slow-to-stop’ model is given in Figure 4.3. In this example, the simulation parameters are exactly the same as in Figure 4.2.

One would think that on a real highway with a fairly low car density, where a small jam is visible from a distance, drivers would slow down enough beforehand to allow the stopped cars to continue. The ‘slow-to-stop’ rule causes drivers to go slower when approaching jams, and as we conjectured this added foresight seems to help to slow down cars enough before the jam so as to let it dissipate on its own over time. There are fewer long jams with many cars at a complete stop, and instead there appear to be many slowdowns to avoid these situations, which we think is fairly accurate behaviour at medium traffic densities.

In Figure 4.4 we give the so-called ‘fundamental diagram’ for our model.
Figure 4.3: A ‘zoomed-out’ view of a larger simulation of our ‘slow-to-stop’ model. Black dots refer to cars, while white space is empty road. The simulation parameters used to produce this output are the same as those used for Figure 4.2. Cars drive from left to right, and time 0 is at the top.
Figure 4.4: The ‘fundamental diagram’ for our model. Each point represents the result from the latter 1000 iterations out of 2000 iterations (to reach steady state) on a road of length 1000 starting from a random configuration. Car density was set from 0 to 0.8, in intervals of 0.02, and ten simulations were performed for each density. $p_{\text{fault}}$ was set to 0.1, and $p_{\text{slow}} = 0.5$. 
We were interested to discover the impact on fuel economy that the 'slow-to-stop' rule would have on the BJH model, so the average number of acceleration cycles and loops driven per car were recorded. The number of accelerations per car was recorded by simply incrementing a counter at each time step by an amount equal to the number of cars whose velocity increased by 1 on that time step. The number of loops driven per car was counted by incrementing a counter each time a car reached the end of the road and started back at the beginning of it. These two quantities provide at least a rough idea of fuel economy. For the simulation parameters used in Figures 4.2 and 4.3 averaged over 10 iterations, it was found that the average number of acceleration cycles per car for the BJH model and the slow-to-stop model was 134.3 and 216.7 respectively, and average number of loops driven per car was 3.7 and 3.4 respectively. It is very interesting that although the 'slow-to-stop' cars had several more acceleration cycles (about 61% more), cars travelled a very similar distance in the same amount of time. Since 'slow-to-stop' cars tend to slow down more often, the two models probably had similar distance results because in the BJH model cars spend more time in complete jams, whereas in our model cars tend to slow down rather than stop completely.

This type of fuel economy indicator (comparing average number of acceleration cycles per car among simulations with a similar average car velocity) can be seen more clearly in Figure 4.5. We can see that for very low or very high average car velocities (resp. very high or very low $\rho$ values), the two models have fairly similar fuel consumption characteristics, but in the middle range our slow-to-stop model causes cars to accelerate much more often. We think this is probably more realistic, since in the BJH model cars are mostly either at a complete stop, or are going at maximum
Figure 4.5: A fuel economy diagram comparing our model with the BJH model. Each point represents the result from the latter 1000 iterations out of 2000 iterations (to reach steady state) on a road of length 1000 starting from a random configuration. The same simulation parameters as in Figure 4.4 were used, but average car speed and average number of acceleration cycles per car were recorded instead.

In summary, we have presented a modification of the well-known BJH model for single lane car traffic, designed to simulate the braking behaviour of cars more correctly. We have provided the fundamental diagram for our model as well as some supplemental simulation results, and have recorded a statistic proportional to fuel economy and the amount of pollution generated. The simulator we have constructed is fairly simple to understand and modify, and could be a useful tool for future researchers to incorporate into their work in this area. Comparison with empirical traffic data is needed in order to tell if our model provides realistic figures for fuel
economy and general driving and jamming characteristics. We believe it may be interesting to compare traffic data from North American traffic networks, since there currently appears to be a shortage of this type of comparison in the literature.

4.2 Coverage in Wireless Sensor Networks

The coverage problem for wireless sensor networks (WSNs) has been studied. Since there are usually many nodes in fairly close proximity in a WSN, there is no need for all of them to be on at once; many should sleep for quite some time until their immediate neighbours run out of energy. The goal of a good coverage algorithm is to extend the life of the network as long as possible by turning off nodes whose operation would be redundant, while maintaining a complete sensor and communication covering of the entire sensing area. All points in the sensing range must be 'covered' by at least one node, and ideally nodes which are more critical to efficient routing would be chosen over less important nodes to stay on. See [37] for more on the theory of coverage. Cunha et al. [22] investigated the use of CA to simulate large WSNs, and used a 01/01 rule to govern the sleep-wake conditions - this means that a sleeping cell wakes up if and only if there are 0 or 1 of its neighbours awake, and an awake cell stays awake if and only if there are 0 or 1 of its neighbours awake. There is also a condition that a cell only consults this rule on a given time step with probability 1/5, otherwise it simply does not change state on that time step. This is to prevent constant on/off switching, which may drain battery life unnecessarily. Percent coverage, number of nodes alive, and number of active nodes were measured, and the figures from this paper are shown in Figure 4.6.

Clearly, in some applications it may be desirable to increase coverage at the cost
Figure 4.6: The figures from the paper by Cunha et al. [22]. The legends for the top two graphs refer to the size of the grid (1000x1000 down to 250x250). The legend for the bottom graph refers to the CA rule used: the 1 means 01/01, the 4 is 01234/01234, and the 8 is 012345678/012345678.
of network lifetime or vice versa. It is for this reason that the results from Cunha et al. [22] were verified and extended using a CA simulator. The same setup was used, but several different transition rules were implemented instead of the 01/01 rule. The results can be seen in Figures 4.7, 4.8, and 4.9.

The 0/0 rule seems to be the best for conserving battery life, but does not do a good job of maintaining a high level of coverage (above 95% coverage may be required for some applications). A good compromise may be rule 0/01. Rule 012/012 maintains a very high degree of coverage for a short period of time, but it seems as if it does not really do any better than the 01/01 rule, which lasts longer. Rule 01/0 seems to actually outperform 01/01 in terms of percent coverage, lasting slightly longer and otherwise staying about even. Several rules are probably not desirable in any way for this application, such as 012/012 or 012/01, since there exist other rules which maintain better coverage for a longer period of time. More investigation is needed on this topic before any definite conclusions can be drawn, but we can conclude that this type of CA-based simulation was computationally efficient, simple to understand, and easy to design.
Figure 4.7: Number of nodes alive over time with several different local rules. Each of the curves shows this statistic for a different CA rule.
Figure 4.8: Number of active nodes over time with several different local rules. Each of the curves shows this statistic for a different CA rule.
Figure 4.9: Percent coverage over time with several different local rules. Each of the curves shows this statistic for a different CA rule.
Chapter 5

Conclusion

After a brief introduction to cellular automata and its applications, we presented a working CA-based public-key cryptosystem based on the idea by Kari, and gave some justification for its security. In the process, we developed some new theory concerning composition and reversibility of marker automata. We also detailed an improvement to the CA algorithm for the 45-convex hull problem by Torbey and Akl which lifts the requirement for a global rule change, provides slightly simpler transition rule descriptions, and corrects the behaviour in a special case. The algorithm uses several communication states to explicitly detect when the first (shrinking) stage has ended, and relying only on local state information the cellular automaton is able to begin the next (expanding) stage of the computation in such a way that correctness is ensured.

We used CA as a modelling tool to study the incorporation of a 'slow-to-stop' rule in a well-known CA-based car traffic flow model for the single-lane highway case, and investigated the use of different local sleep/wake rules for the coverage problem in large wireless sensor network design. Each of these contributions has a functional and tested software implementation, and source code for the 'FDM CA Cryptosystem' of
Chapter 2 is included in Appendix A.

There are several extensions to each of the topics presented in this dissertation. The CA-based public-key cryptosystem outlines a key generation algorithm which is by no means required for a working or secure system - the design of a different and perhaps faster or more secure algorithm could be attempted, using our theory of composition and reversibility of FDM CA. The theory itself could be extended to perhaps say something about cellular automata more general than FDM CA. The security of the existing system could (and should) also be studied much more thoroughly. The first piece of work that should really be done is to provide a complete specification for the cryptosystem. One needs to specify key sizes and methods of storage, perhaps optimize the key generation, composition, and encryption algorithms, and fix all of the parameters for them. Once this has been done, cryptanalysis can be attempted. In the best case, one may be able to prove in some way that determining a private key from a public key is an NP-hard problem. More realistically, one may either be able to simply break the system by finding some fast way to invert a public key, or one may be able to give some good justification for why that sort of thing cannot be done. In the case where the system gets broken, the successful attacker may also find some way to correct whatever weakness caused the vulnerability. If the system does stand up to cryptanalysis, then a practical implementation should be constructed in order to determine whether a working implementation would be feasible for some application.

The modification to the algorithm for the 45-convex hull problem could be improved further by requiring fewer states to do the communication stage, or simply by doing the communication in a different or faster way. The time bounds for the
operation of the algorithm could also be tightened. An interesting extension could be to implement the algorithm in three dimensions.

Our extension to the Benjamin-Johnson-Hui model for single-lane highway traffic was designed so that it exhibits more accurate microscopic behaviour, but it remains to be seen whether the model will have similar macroscopic characteristics to real car traffic. In this investigation, one could possibly obtain some empirical traffic data from a North American highway, since in the literature it seems that most of the real traffic data used in CA modelling analysis comes from European highways. The investigation of the coverage problem in wireless sensor networks was very preliminary. We only looked at the application of different sleep/wake rules in the design; many other schemes for coverage could be simulated using CA, including different types of local rules, neighbourhoods, states, or hierarchical network configurations.

CA modelling of physical systems such as car traffic or wireless sensor networks is currently a popular research area, with many recent published papers. For a brief survey of a sample of the work in these areas one could refer to Sections 1.3.3 and 1.3.4, but it should not be difficult to find many ideas and open questions in recent papers.
Bibliography


Appendix A

FDM CA Cryptosystem: Code Listing

# FDM CA Cryptosystem Simulator. Written by Adam Clarridge
# (3agc AT queensu DOT ca)
# Usage: Simply run this script and use the on-screen menu.
# For more information, see the README.txt file.

import math,random,copy
from xml.dom import minidom

#The Fixed-Domain Marker Cellular Automaton class.
class FDM_CA:

    #constructor
    def __init__(self, dimension, state_set, neighbourhood):
        self.dimension = dimension
        self.state_set=state_set[:]

        #the neighbourhood is a tuple of tuples representing
        # the set of dimension-
        # sized vectors corresponding to the CA's
        # neighbourhood.
# Example: neighbourhood = ((1,0),(0,1),(-1,0),
# (0,-1)) is the 2-dimensional von Neumann
# neighbourhood without the cell itself
self.neighbourhood = neighbourhood

# the transition_rules dictionary works as follows:
# if the neighbourhood of a cell c is part of this
# CA's acting_set, then if the state of c is s, c
# will change to transition_rules[s] on the next
# generation.
self.transition_rules=
# initialize transition_rules to the identity map.
for s in state_set:
    self.transition_rules[s] = s

# the acting set is a list of tuples corresponding to
# the neighbourhood positions in the order of the
# self.neighbourhood tuple.
# Example: acting_set = [(s1,s2,s3,s4),(s1,s1,s1,s2)
# ,(s3,s1,s1,s2)] (here (s1,s2,s3,s4) corresponds to
# s2
# s3 c s1
# s4
# with the neighbourhood as in the example above.)
self.acting_set = []

# stores the inverse transition function
self.inverse_transition_rules = {}

# apply this CA to a given message for one generation.
# the "message" input is a dictionary keyed by cell
# location. So a message encoded into a 2x2 array may
# have the following entries:
# message[(0,0)] = s1
# message[(1,0)] = s2
# message[(0,1)] = s3
# message[(1,1)] = s4
# messages in any rectangular shape are acceptable
# (higher dimensions than 2 are OK as well)
def apply(self, message, sizes):
tmp={}
for key in message.keys():
    nbhood = []
    for nb in self.neighbourhood:
        nbhood.append(message[vector_add(
            (key,nb,sizes))])
    if tuple(nbhood) in self.acting_set:
        tmp[key] = self.transition_rules[
            message[key]]
    else:
        tmp[key] = message[key]
return tmp

#calculate the inverse transition function
def calculate_inverse(self):
    for state in self.state_set:
        self.inverse_transition_rules[
            self.transition_rules[state]]=state

#apply the inverse of this CA for one generation
def apply_inverse(self, message, sizes):
    tmp={}
    for key in message.keys():
        nbhood = []
        for nb in self.neighbourhood:
            nbhood.append(message[vector_add(
                (key,nb,sizes))])
        if tuple(nbhood) in self.acting_set:
            tmp[key] = self.inverse_transition_rules[
                message[key]]
        else:
            tmp[key] = message[key]
    return tmp

#Creates a sequence of FDM RCA at initialization and
#composes them together to form one CA. Also provides
#encryption/decryption functions.
class FDM_CA_Cryptosystem():
    def __init__(self):
        pass
self.generated = False
self.read_private = False
self.read_public = False
self.have_parameters = False
self.rows=0
self.cols=0
self.num_generations=0

# method to write a public or private key to an xml file
def write_key(self, filename, key_type):
    if key_type=="public":
        print "Writing public key to "+filename+"...",
        # write a public key
        xml = minidom.Document()
        pubkey = xml.createElement("public_key")
        state_set_str = ""
        for state in self.state_set:
            state_set_str = state_set_str + state
        nbhd_str = ""
        for nb in range(0,len(self.neighbourhood)):
            for num in range(len(self.neighbourhood[nb])):
                nbhd_str = nbhd_str + str(self.neighbourhood[nb][num])
            if num<len(self.neighbourhood[nb])-1:
                nbhd_str = nbhd_str + ","
            if nb<len(self.neighbourhood)-1:
                nbhd_str = nbhd_str + ";"
        pubkey.setAttribute("state_set", state_set_str)
        pubkey.setAttribute("neighbourhood", nbhd_str)
        pubkey.setAttribute("dimension", str(self.dimension))
        for nbhd in self.composed_CA:
            nbhd_str = ""
            for state in nbhd:
                nbhd_str = nbhd_str + state
            rule = xml.createElement("neighbourhood")
            rule.setAttribute("configuration",nbhd_str)
            for state in self.composed_CA[nbhd]:
rule2 = xml.createElement("map")
rule2.setAttribute("state",str(state))
rule2.setAttribute("next_state",\
str(self.composed_CA[nbhd][state]))
rule.appendChild(rule2)
pubkey.appendChild(rule)
xml.appendChild(pubkey)
f = open(filename,"w")
f.write(xml.toprettyxml())
f.close()

print "Done."
elif key_type=="private":
    print "Writing private key to "+filename+"...",
    #write a private key
    xml = minidom.Document()
    privkey = xml.createElement("private_key")
    state_set_str = ""
    for state in self.state_set:
        state_set_str=state_set_str + state
    nbhd_str = ""
    for nb in range(0,len(self.neighbourhood)):
        for num in range(len(\n            self.neighbourhood[nb])):
            nbhd_str=nbhd_str + \n            str(self.neighbourhood[nb][num])
            if num<len(self.neighbourhood[nb])-1:
                nbhd_str=nbhd_str + ","
            else:
                nbhd_str = nbhd_str + ";"
    privkey.setAttribute("state_set", state_set_str)
    privkey.setAttribute("neighbourhood", nbhd_str)
    privkey.setAttribute("num_FDM_CA", \n    str(len(self.FDM_CA_list)))
    privkey.setAttribute("dimension", \n    str(self.FDM_CA_list[0].dimension))
    for i in range(0,len(self.FDM_CA_list)):
        fdmca = xml.createElement("FDM_CA")
fdmca.setAttribute("number",str(i))
cia = self.FDM_CA_list[i]
actingset = xml.createElement("acting_set")
for nbhd in ca.acting_set:
    nbhd Elem = xml.createElement("nbhd")
nbhd_str = 
    for state in nbhd:
        nbhd_str = nbhd_str+state
    nbhd Elem.setAttribute("conf",nbhd_str)
actingset.appendChild(nbhd Elem)

function = xml.createElement("function")
for state in ca.transition_rules:
    map Elem = xml.createElement("map")
    map Elem.setAttribute("state",str(state))
    map Elem.setAttribute("next_state",str(ca.transition_rules[state]))
function.appendChild(map Elem)

fdmca.appendChild(actingset)
fdmca.appendChild(function)
privkey.appendChild(fdmca)

xml.appendChild(privkey)
f = open(filename,"w")
f.write(xml.toprettyxml())
f.close()
print "Done."
else:
    print "Invalid key type. Please enter '+' 'public' or 'private'."

def write_parameters(self,filename):
    print "Writing parameters to "+filename+"...",
xml = minidom.Document()
params = xml.createElement("parameters")
state_set_str = 
for state in self.state_set:
    state_set_str=state_set_str + state
nbhd_str = ""
for nb in range(0, len(self.neighbourhood)):
    for num in range(len(self.neighbourhood[nb])):
        nbhd_str=nbhd_str + str(self.neighbourhood[nb][num])
        if num<len(self.neighbourhood[nb])-1:
            nbhd_str=nbhd_str + "",
        if nb<len(self.neighbourhood)-1:
            nbhd_str = nbhd_str + ";"
CA_params = xml.createElement("CA_parameters")
CA_params.setAttribute("dimension", str(self.dimension))
CA_params.setAttribute("neighbourhood", nbhd_str)
CA_params.setAttribute("state_set", state_set_str)
alg_params = xml.createElement("algorithm_parameters")
alg_params.setAttribute("numCA", str(self.num_CA))
alg_params.setAttribute("p", str(self.p))
alg_params.setAttribute("q", str(self.q))
params.appendChild(CA_params)
params.appendChild(alg_params)
xml.appendChild(params)
f = open(filename,"w")
f.write(xml.toprettyxml())
f.close()
print "Done."

#method to read the algorithm parameters from a file
def read_parameters(self, filename):
    try:
        print "Reading parameters from "+filename+"...",
        go=True
dom = minidom.parse(filename)
    except:
        print "Error: File either cannot be read or "+"is malformed XML."
        go = False
    if go:
        params = dom.firstChild
CA_params = params.getElementsByTagName("CA_parameters").item(0)
alg_params = params.getElementsByTagName("algorithm_parameters").item(0)
self.state_set = list(str(CA_params.getAttribute("state_set")))
self.dimension = int(str(CA_params.getAttribute("dimension")))
tmp_nbhd = str(CA_params.getAttribute("neighbourhood")).split(";")
self.neighbourhood = []
for nb in tmp_nbhd:
  n = []
  for num in nb.split(","): n.append(int(num))
  self.neighbourhood.append(tuple(n))
self.num_CA = int(str(alg_params.getAttribute("numCA")))
self.p = float(str(alg_params.getAttribute("p")))
self.q = float(str(alg_params.getAttribute("q")))
self.have_parameters = True
print "Done."

#method to read a public or private key from an xml file
def read_key(self,filename,key_type):
  if key_type="public":
    print "Reading public key from "+filename+"...",
    #read a public key
    dom = minidom.parse(filename)
    pubkey = dom.firstChild
    self.state_set = list(str(pubkey.getAttribute("state_set")))
    self.dimension = int(str(pubkey.getAttribute("dimension")))
    tmp_nbhd = str(pubkey.getAttribute("neighbourhood")).split(";")
    self.neighbourhood = []
    for nb in tmp_nbhd:
n = []
for num in nb.split("",""):  
n.append(int(num))
self.neighbourhood.append(tuple(n))
self.composed_CA={}  
for nbhd_node in pubkey.getElementsByTagName("neighbourhood"):  
nbhd = tuple(list(str(nbhd_node.getAttribute("configuration"))))
self.composed_CA[nbhd]={}  
for mapping_node in nbhd_node.getElementsByTagName("map"):  
self.composed_CA[nbhd][mapping_node.getAttribute("state")]=mapping_node.getAttribute("next_state")

self.read_public=True
print "Done. Public key is now in memory."  
elif key_type="private":  
print "Reading private key from "+filename+"...",  
#read a private key  

dom = minidom.parse(filename)  
privkey = dom.firstChild  
state_set = list(str(privkey.getAttribute("state_set")))  
tmp_nbhd = str(privkey.getAttribute("neighbourhood")).split(";")  
neighbourhood = []  
for nb in tmp_nbhd:  
n = []  
for num in nb.split("",""):  
n.append(int(num))
neighbourhood.append(tuple(n))
num_FDM_CA = int(str(privkey.getAttribute("num_FDM_CA")))
self.dimension = int(str(privkey.getAttribute("dimension")))
self.FDM_CA_list = []
```python
for i in range(num_FDM_CA):
    self.FDM_CA_list.append(FDM_CA(
        self.dimension,state_set,neighbourhood))

for ca_node in privkey.getElementsByTagName("FDM_CA"):
    num = int(str(ca_node.getAttribute("number")))
    for acting_set_node in ca_node.getElementsByTagName("acting_set"):
        for nbhd_node in acting_set_node.getElementsByTagName("nbhd"):
            nbhd_tup = tuple(list(str(nbhd_node.getAttribute("conf"))))
            self.FDM_CA_list[num].acting_set.append(nbhd_tup)
    for function_node in ca_node.getElementsByTagName("function"):
        for map_node in function_node.getElementsByTagName("map"):
            self.FDM_CA_list[num].
            transition_rules[str(map_node.getAttribute("state"))] = str(
                map_node.getAttribute("next_state"))
            self.FDM_CA_list[num].calculate_inverse()

    self.read_private = True
    print "Done. Private key is now in memory."
else:
    print "Invalid key type. Please enter '+/ 'public' or 'private'.'"

def generate_keys(self,dimension,state_set,\neighbourhood,num_CA,p,q):
    self.dimension = dimension
    self.state_set = state_set[:]
    self.neighbourhood = neighbourhood
    self.neighbourhood_size = len(neighbourhood)
    self.num_states = len(state_set)
    self.num_CA = num_CA
```
self.p=p
self.q=q
#the list of FDM CA objects to be composed together
self.FDM_CA_list = []
self.all_possible_neighbourhoods = self.
generate_all_possible_neighbourhoods()
#initialize the composed rule
self.composed_CA = {}
for nbhd in self.all_possible_neighbourhoods:
    tnbhd = tuple(nbhd)
    self.composed_CA[tnbhd] = {}
    for s in state_set:
        self.composed_CA[tnbhd][s]=s

print "Generating the random FDM CA list...",
#Begin the algorithm to generate a random FDM CA
#list
change_set = []
the_rest = self.state_set[:]
state = random.choice(the_rest)
the_rest.remove(state)
change_set.append(state)
state = random.choice(the_rest)
the_rest.remove(state)
change_set.append(state)
for i in range(0,self.num_CA):
    ca = FDM_CA(self.dimension,self.state_set,
    self.neighbourhood)

    if random.uniform(0.0,1.0)<p and the_rest!=[]:
        state = random.choice(the_rest)
        the_rest.remove(state)
        change_set.append(state)

    #choose the acting set for this FDM CA
    positions = self.random_binary(
    self.neighbourhood_size)
    for nbhd in self.all_possible_neighbourhoods:
        unchanging_neighbourhood = True
        rg = range(self.neighbourhood_size)
for n in rg:
    if nbhd[n] in change_set:
        unchanging_neighbourhood = False
        if positions[n]==True:
            #accept the neighbourhood for
            #sure in this case
            ca.acting_set.append(nbhd)
            break
        #if none of the neighbourhood states are
        #part of the change set, then accept the
        #neighbourhood with probability q
        if unchanging_neighbourhood == True and \
           random.uniform(0.0,1.0)<q:
            ca.acting_set.append(nbhd)

ca.transition_rules = \ 
self.random_transition_rules(change_set)
ca.calculate_inverse()
self.FDM_CA_list.append(ca)

print "Composing into one CA...",
self.compose_CA()
self.generated = True
print "Done. The public and private keys are "+\ 
"now in memory."

#generates a random binary string of a specified length
#the binary string is useful in randomly choosing the
#acting_set of an FDM_CA.
def random_binary(self,length):
    num = int(random.uniform(0.0,1.0)*(2**length))
    list = []
    for i in range(length):
        if num/(2**(length-i-1))!=0:
            num-=2**(length-1-i)
            list.append(True)
        else:
            list.append(False)
    return list
# Generates a list of all possible neighbourhoods.  
# This is useful to iterate through when finding the  
# composed_CA.

def generate_all_possible_neighbourhoods(self):
    nbhds = []
    total = self.num_states**self.neighbourhood_size
    for i in range(0,total):
        temp = []
        num=i
        for j in range(self.neighbourhood_size):
            temp.append(self.state_set[num/(self.num_states**(self.neighbourhood_size-j-1))])
            num-=(num/(self.num_states**(self.neighbourhood_size-j-1)))*(self.num_states**(self.neighbourhood_size-j-1))
    nbhds.append(tuple(temp))
    return nbhds

# this function generates a random permutation of the  
# given change_set, and returns the result as a  
# dictionary of outputs keyed by inputs.

def random_transition_rules(self,change_set):
    cs = change_set[:]
    left = change_set[:]
    tr = {}
    for state in self.state_set:
        if state in cs:
            rand = random.choice(left)
            tr[state] = rand
            left.remove(rand)
        else:
            tr[state]=state
    return tr

# encrypt a message using each of the CA in succession.  
# This has the same result as encrypting with the  
# composed CA and was only used for debugging and  
# proof-of-concept purposes.

def encrypt(self,message,num_iter):
    sizes = [0]*self.dimension
for key in message.keys():
    for i in range(0, self.dimension):
        if key[i] > sizes[i]:
            sizes[i] = key[i]
for i in range(num_iterations):
    for ca in self.FDM_CA_list:
        message = ca.apply(message, sizes)
return message

# encrypt a message using the public key, the composed CA.
def encrypt_with_composed_CA(self, message, num_iterations):
    sizes = [0] * self.dimension
    for key in message.keys():
        for i in range(0, self.dimension):
            if key[i] > sizes[i]:
                sizes[i] = key[i]
    for i in range(num_iterations):
        tmp = {}
        for key in message.keys():
            nbhood = []
            for nb in self.neighbourhood:
                nbhood.append(message[vector_add(
                    key, nb, sizes)])
            tmp[key] = self.composed_CA[tuple(nbhood)]
            [message[key]]
        message = tmp
    return tmp

# decrypts ciphertext by applying the inverse of each rule in reverse order
def decrypt(self, message, num_iterations):
    sizes = [0] * self.dimension
    for key in message.keys():
        for i in range(0, self.dimension):
            if key[i] > sizes[i]:
                sizes[i] = key[i]
    for i in range(num_iterations):
        for ca in reversed(self.FDM_CA_list):
message = ca.apply_inverse(message, sizes)
return message

def compose_CA(self):
    for neighbourhood in self.all_possible_neighbourhoods:
        actors = []
        # find out which rules will act on this
        # neighbourhood
        for c in range(0, len(self.FDM_CA_list)):
            if neighbourhood in self.FDM_CA_list[c].acting_set:
                actors.append(True)
            else:
                actors.append(False)

        # simulate each rule acting on the neighbourhood
        # in succession,
        # and record the result in the composed rule.
        for s in self.state_set:
            state = s
            for ca in range(0, len(self.FDM_CA_list)):
                if actors[ca]:
                    state = self.FDM_CA_list[ca].transition_rules[state]
            self.composed_CA[neighbourhood][s] = state

# helps the user to create a 2D message with a specified
# number of rows and columns and the desired text, with
# random states filling up the rest of the message
def create_2D_message(text, state_set, rowsize, colsiz):
    msg = {}
    for i in range(0, rowsize):
        for j in range(0, colsiz):
            if len(text) == 0:
                msg[(i, j)] = random.choice(state_set)
            else:
                state = text[0]
                text.remove(state)
msg[(i,j)] = state
return msg

# prints a 2D message to the screen
def print_2D_message(message):
    rowsize, colsize = 0, 0
    for key in message.keys():
        i = key[0]
        j = key[1]
        if i > rowsize:
            rowsize = i
        if j > colsize:
            colsize = j
    for i in range(0, rowsize + 1):
        for j in range(0, colsize + 1):
            print message[(i, j)],
        print "\n"

def write_2D_message(message, filename):
    print "Writing message to " + filename + ". . . ",
    rowsize, colsize = 0, 0
    for key in message.keys():
        i = key[0]
        j = key[1]
        if i > rowsize:
            rowsize = i
        if j > colsize:
            colsize = j
    colsize += 1
    rowsize += 1
    xml = minidom.Document()
    msg = xml.createElement("message")
    msg.setAttribute("rows", str(rowsize))
    msg.setAttribute("cols", str(colsize))
    for i in range(rowsize):
        tmp = ""
        for j in range(colsize):
            tmp = tmp + message[(i, j)]
        row = xml.createElement("row")
        row.appendChild(xml.createTextNode(tmp))
        msg.appendChild(row)
    xml.appendChild(msg)
    xml.normalize()
    try:
        xml.writexml(open(filename, 'w'), indent = " \t ")
    except:
        print "Error writing XML file " + filename + "!

APPENDIX A. FDM CA CRYPTOSYSTEM: CODE LISTING

```python
    row.setAttribute("number",str(i))
    row.setAttribute("value",tmp)
    msg.appendChild(row)

    xml.appendChild(msg)
    f = open(filename,"w")
    f.write(xml.toprettyxml())
    f.close()
    print "Done."

def write_2D_ciphertext(ciphertext,filename,\   num_generations):
    print "Writing ciphertext to "+filename+"... ",
    rowsize,colsize=0,0
    for key in ciphertext.keys():
        i=key[0]
        j=key[1]
        if i>rowsize:
            rowsize=i
        if j>colsize:
            colsize=j
    colsize+=1
    rowsize+=1
    xml = minidom.Document()
    ctext = xml.createElement("ciphertext")
    ctext.setAttribute("rows", str(rowsize))
    ctext.setAttribute("cols", str(colsize))
    ctext.setAttribute("num_generations",\    str(num_generations))

    for i in range(rowsize):
        tmp=""
        for j in range(colsize):
            tmp = tmp + ciphertext[(i,j)]
            row = xml.createElement("row")
            row.setAttribute("number",str(i))
            row.setAttribute("value",tmp)
            ctext.appendChild(row)

    xml.appendChild(ctext)
```
f = open(filename,"w")
f.write(xml.toprettyxml())
f.close()
print "Done."

def read_2D_message(filename):
    try:
        print "Reading message from "+filename+"... ",
go=True
    dom = minidom.parse(filename)
    except:
        print "Error: File either cannot be read or "+\n"is malformed XML."
go = False
if go:
    message = dom.firstChild
    numrows = int(message.getAttribute("rows"))
    numcols = int(message.getAttribute("cols"))
    rows = message.getElementsByTagName("row")
    msg={} 
    for i in range(0,numrows):
        r = int(rows.item(i).getAttribute("number"))
        val = rows.item(i).getAttribute("value")
        for j in range(0,numcols):
            msg[(r,j)]=val[j]
    print "Done."
    return msg,numrows,numcols

def read_2D_ciphertext(filename):
    try:
        print "Reading ciphertext from "+filename+"... ",
go=True
    dom = minidom.parse(filename)
    except:
        print "Error: File either cannot be read or is "+\n"malformed XML."
go = False
if go:
    ctext = dom.firstChild
    numrows = int(ctext.getAttribute("rows"))
numcols = int(ctext.getAttribute("cols"))
num_generations = int(ctext.getAttribute("num_generations"))
rows = ctext.getElementsByTagName("row")
ct={}
for i in range(0,numrows):
    r = int(rows.item(i).getAttribute("number"))
    val = rows.item(i).getAttribute("value")
    for j in range(0,numcols):
        ct[(r,j)]=val[j]
print "Done."
return ct,num_generations

#adds two vectors (tuples) componentwise.
def vector_add(vec1, vec2, sizes):
    sum = []
    for i in range(0,len(vec1)):
        sum.append((vec1[i]+vec2[i])%(sizes[i]+1))
    return tuple(sum)

if __name__=="__main__":
    sys = FDM_CA_Cryptosystem()
    try:
        while True:
            print "Please make a selection:
            print " 1. Generate a random key pair"
            print " 2. Read the algorithm parameters from a file"
            print " 3. Read a public key from a file"
            print " 4. Read a private key from a file"
            print " 5. Write the algorithm parameters to a file"
            print " 6. Write a public key to a file"
            print " 7. Write a private key to a file"
            print " 8. Encrypt a 2D message"
            print " 9. Decrypt a 2D message’s ciphertext"
            print " 0. Quit"
choice = raw_input("Please make a selection.")
if choice=='1':
    if sys.have_parameters == False:
        print "You do not have the algorithm"+
        " parameters in memory. Please make"+
        " a selection:
        print " 1. Input parameters manually"
        print " 2. Input parameters from a"+
        " file"

choice = raw_input("Please make a "+
"selection.")
if choice == "1":
    dim = int(str(raw_input("Please "+
"enter the dimension size. "+
"Currently the visual display of "+
"non-2D messages is not "+
"supported.")))
    ss = list(str(raw_input("Please "+
"enter the state set in a single "+
"string, one state per character."+
" Ex. 'abcd'.")))
    nh = raw_input("Please enter the "+
"neighbourhood of each cell "+
"separated by semicolons, and do "+
"not include spaces. Ex. "+
"'0,0;1,0;-1,1' refers to the "+
"cell itself, the right neighbour"+
" and the top-left neighbour.")
    tmp_nh = str(nh).split(";")
    nh = []
    for nb in tmp_nh:
        n = []
        for num in nb.split(","):  
            n.append(int(num))
        nh.append(tuple(n))
    nCA = int(str(raw_input("Please "+
"enter the number of FDM CA you "+
"would like in the composition.")))
    prob1 = float(str(raw_input(\"
"Please enter the probability p "+
"(chance to add an element to the"
" change set'). Ex. '0.5'.")
prob2 = float(str(raw_input(
"Please enter the probability q "+
"(chance to add a neighbourhood "+
" containing no elements of the "+
" change set to the acting set). "+
" Ex. '0.5'.")))
sys.generate_keys(dim,ss,nh,nCA,\nprob1,prob2)
sys.have_parameters = True
elif choice=="2":
    fn = str(raw_input("Please enter "+\n" the name of the file."))
    #read parameters from a file
    sys.read_parameters(fn)
    sys.generate_keys(sys.dimension,\n    sys.state_set,sys.neighbourhood,\n    sys.num_CA,sys.p,sys.q)
else:
    sys.generate_keys(sys.dimension,sys.\n    state_set,sys.neighbourhood,sys.num_CA,\n    sys.p,sys.q)
elif choice == '2':
    filename = raw_input\n    ("Please enter the filename.")
    sys.read_parameters(filename)
elif choice=='3':
    filename = raw_input\n    ("Please enter the filename.")
    sys.read_key(filename,"public")
elif choice=='4':
    filename = raw_input\n    ("Please enter the filename.")
    sys.read_key(filename,"private")
elif choice=='5':
    if sys.have_parameters:
        filename = raw_input\n        ("Please enter the filename.")
sys.write_parameters(filename)
else:
    print "Error: No parameters are "+
    "loaded into memory."
elif choice=='6':
    if sys.read_public==False and \
        sys.generated==False:
        print "Generate or read a public key "+
        "before trying to write."
    else:
        filename = raw_input\n        ("Please enter the filename.")
        sys.write_key(filename,"public")
elif choice=='7':
    if sys.read_private==False and \
        sys.generated==False:
        print "Generate or read a private key"+
        " before trying to write."
    else:
        filename = raw_input\n        ("Please enter the filename.")
        sys.write_key(filename,"private")
elif choice=='8':
    if sys.generated or sys.read_public:
        while True:
            print "Please make a selection:"
            print "  1. Manually input a"+
            " message"
            print "  2. Read a message from"+
            " a file"
            choice = raw_input("Please make a"+
            " selection.")
            if choice == '1':
                sys.rows = int(str(raw_input\n                ("Please enter the number of "+
                "rows in your message.")))  
                sys.cols = int(str(raw_input\n                ("Please enter the number of "+
                "columns in your message.")))  
                message = str(raw_input\n                ("Please enter your message."))
"Please enter the message "+
"you want to encrypt. "+
"Remember to only use members"+
" of the state set in your "+
"message. If your message is "+
"smaller than the specified "+
"size, the rest of the "+
"configuration will be set "+
"randomly.")
msg = create_2D_message(list\(message),sys.state_set,\(sys.rows,sys.cols)\)
elif choice == '2':
  fn = raw_input\("Please enter the name of "+
  "the file containing the "+
  "message."")
msg,rows,cols = \(read_2D_message(fn)\)
if choice == 1 or choice== 2:
  sys.num_generations = int(str(\(raw_input("Please enter the "+
  "number of generations to "+
  "evolve the CA.")\))
print "Original Message:"
print_2D_message(msg)
print "Encrypted with "+
"Composed CA:"
encrypted_with_composed_CA = \(sys.encrypt_with_composed_CA\(msg,sys.num_generations)\)
print_2D_message\(encrypted_with_composed_CA\)
choice = raw_input\("Would you like to write the"+
" ciphertext to a file? (y/n)"")
if choice== 'y' or choice== 'Y':
  fn = raw_input\("Please enter the name "+
  "of the file."\)
write_2D_ciphertext(
    encrypted_with_composed_CA,\
    fn, sys.num_generations)
choice = raw_input("Would you"+\
    " like to write the original "+\
    "message to a file? (y/n)")
if choice=='y' or choice=='Y':
    fn = raw_input(
        "Please enter the name "+\
        "of the file.")
    write_2D_message(msg,fn)
browser
else:
    print "Invalid selection."+\
    " Please try again."
eaxl:
    print "Error: Can’t encrypt, no"+\
    " public key in memory."
elif choice=='9':
    if sys.generated or sys.read_private:
        fn = raw_input("Please enter the name"+\
            " of the file containing the "+\
            "cipherertext.")
    ctext, gens = read_2D_ciphertext(fn)
    print "Decrypted Message:"
    decrypted = sys.decrypt(ctext, gens)
    print_2D_message(decrypted)
    choice = raw_input("Would you like to"+\
        " write the decrypted message to a "+\
        "file? (y/n)")
    if choice=='y' or choice=='Y':
        fn = raw_input("Please enter the "+\
            "name of the file.")
        write_2D_message(decrypted,fn)
    else:
        print "Error: Can’t decrypt, no "+\
            "private key in memory."
elif choice=='0':
    break
else:
print "Invalid input. Please try again."
except:
    print 'Error: Invalid input.'