Evolution of Cellular Automata for Image Processing

Christopher D Thomas
Supervisor: Riccardo Poli
School of Computer Science
University of Birmingham (UK)
Student No.: 0292406
April 2000

Abstract

Cellular automata are dynamical systems in which time and space are discrete. A cellular automaton consists of an n-dimensional grid/array of cells. Each of these cells can be in one of a finite number of possible states, updated in parallel according to a state transition function.

Traditionally cellular automata have been implemented as uniform cellular automata (all cells having the same state transition function). It is only recently that non-uniform cellular automata (each cell having a possible different state transition function) have been used. The purposes that they have been used for include: hydrodynamics, thermodynamics, A-life, synchronization, simple image processing and control problems. Also genetic algorithms are being used to search the space of state transition functions in drastically smaller time periods.

Because of the quite obvious similarities between 2-dimensional cellular automata and images (pixels mapping to cells) uniform cellular automata have been used quite a lot in some areas of image processing or similar research.

In this paper it describes the attempt, and my results from evolving both uniform and non-uniform cellular automata, using genetic algorithms, to do both simple and more complicated image processing – focusing mainly on edge detection and thinning.

Contents

1 . Introduction *1*

2. Further background information *2*

2.1 Basics of Cellular Automata

2.1.1 State transition functions

2.1.2 An example of how a cellular automata operates

2.1.3 The Size of the rule/search space

2.2 Basics of Evolutionary Algorithms

2.2.1 Introduction

2.2.2 Genetic Algorithms

2.2.3 Selection and fitness functions

2.2.4 Crossover

2.2.5 Mutation

2.2.6 The standard genetic algorithm

3. Research and existing work in this specialised field *6*

3.1 Classes of Cellular Automata

3.2 A Universal Computation Class

3.2 Classes of Cellular Automata

3.3 Related research in Cellular Automata to Image Processing

3.4 Image Processing at a low-level

3.5 Conventional Image Processing algorithms

3.6 Relevance of the work to Artificial Intelligence

3.7 Limits imposed by current technology

3.8 Current research into Cellular Automata

3.9 Concluding remarks

4. The basic Cellular Algorithm *10*

5. The cellular automata simulator *11*

5.1 Designing the cellular automata simulator

5.2 Requirements of the cellular automata

Contents, continued …

6. **Evolution of cellular automata for image processing** ***13***

6.1 The game of life
6.2 Traditional uniform cellular automata
6.3 Preliminary non-uniform cellular automata experiments
6.4 Preliminary 100 by 100 edge detection cellular automata experiments
6.5 Non-uniform cellular automata experiments
6.5.1 Static Image Edge Detection with crossover and mutation
6.5.2 Preliminary generic image edge detection
6.6 Using cellular automata for the thinning task
6.6.1 Introduction
6.6.2 Static horizontal thinning
6.6.3 Generic image thinning in both horizontal and vertical planes
6.6.4 Varying the fitness function with generic thinning
6.6.5 Varying *M* – the number of time steps to utilize rules
6.6.6 Concluding remarks on this section
6.7 Reducing the rule space
6.8 Scalability
6.9 Thoughts on three-dimensional cellular automata
6.10 Using diagonal cells for thinning
6.11 Using diagonal cells for edge detection
6.12 A sub-iterative algorithm
6.13 Greyscale edge detection
6.14 Three-dimensional cellular automata with time as the third dimension

7. **Final Results** ***30***

8. **Appraisal** ***32***

8.1 Appraisal of my Specification
8.1.1 Appraisal of the objectives
8.1.2 Appraisal of the implementation
8.1.3 Appraisal of the key stages
8.1.4 Appraisal of the fall back positions
8.2 Appraisal of my image processing results
8.2.1 Appraisal with respect to my criteria
8.2.2 Appraisal with respect to other algorithms

9. **Possible extensions to this research** ***36***

10. **Conclusions** ***37***

11. **Acknowledgements** ***39***

Contents, continued …

12. References *38*

Appendices

Appendix 1: *42*
1 Table of figures

Appendix 2: *43*
2.1 Extended examples of results for synthetic images
2.2 Extended examples of results for natural images
2.3 Extended examples of results for text images

Appendix 3: *51*
3 Various graphs of results.

Appendix 4: *54*
4 How to run my program

Appendix 5: *56*
5.1 Rule-run code-fragment
5.2 Cross-over code fragment
5.3 Mutation code-fragment
5.4 Cell class code-fragment

Appendix 6: *59*
6.1 Original specification
6.2 Original work-plan

Appendix 7: *64*
7 A revised work-plan

Appendix 8: *65*
8 The outline of my edge detection algorithm

1. Introduction

In this project my aim was to demonstrate whether or not cellular automata are useful for forms of image processing. The most important part of this project is the understanding of what Cellular Automata can and cannot do and why. The actual piece of software that I have developed is not so important, the fact that it actually works quite well is not important – it would have been just as significant if it hadn't worked. To be able to start to find whether or not cellular automata can be used for image processing I have had to do extensive research into cellular automata and evolutionary computation, which has made up a large proportion of this project.

The basic model of a cellular automaton consists of an array of cells each of which can be in one of a finite number of possible states, updated synchronously in discrete time steps, according to a local identical interaction state transition function. The state of a cell at the next time step is determined by the current states of a surrounding neighbourhood of cells. [Wolfram, 1984; Toffoli and Margolus, 1987].

The main model that I employ in this research is an extension to the basic model called "non-uniform cellular automata". It is identical to the basic model apart from one fundamental difference; the state transition functions need not be identical for all cells. This model of cellular automata still shares the basic "attractive" properties of uniform ones; massive parallelism, locality of cellular interactions and simplicity of cells.

The original model is essentially programmed at an extremely low level. A single state transition function is sought that must be universally applied to all cells in the grid. This task can be extremely complex even when using evolutionary algorithms. Using non-uniform cellular automata the search space is immensely larger than that of uniform cellular automata but because of this there is much more scope of what can be done with them.

One of the reasons I chose cellular automata for my project is that very little work has been carried out in this area, this enables me to do some original research rather than just summarising and drawing conclusions from other's work.

2. Further background information

To understand my research into Cellular Automata I will explain the basic concepts of how they work and then the basic concepts of evolutionary algorithms.

2.1 Basics of Cellular Automata

2.1.1 State transition functions

Throughout this paper I refer to rules, these rules are more formally known as state transition functions. A function that depending on what is inputted to it, the new state of the cell is given.

2.1.2 An example of how a cellular automata operates

To demonstrate exactly how a cellular automaton works with respect to its rules I will use a simple 1-dimensional, 2 state, 8 cell uniform cellular automaton with a cell neighbourhood of 2.

The cellular automaton uses the set {0,1} for its two states. Figure 1 is an example of an initial configuration of the cellular automaton:

1	0	1	0	0	0	1	1

Figure 1: An example of a cellular automata

The 2 cell neighbourhood, in this case, means looking at the cell in question and the cell to the right of it. Figure 2 shows this:

1	0	1	0	0	0	1	1

Figure 2: An example of cell neighbourhood in a cellular automata

The two cells surrounded by the bold rectangle are the cell neighbourhood for the leftmost cell. When you look at the rightmost cell, the other cell (other than itself) in its neighbourhood is the leftmost cell – forming a circular arrangement of cells. If we were using a three cell neighbourhood, we would look at the cell on the left and the right of the cell in question.

I have used a two cell neighbourhood example so that the rule space is small enough to demonstrate its effects. The two cell neighbourhood holds a binary 2-bit number (00 or 01 or 10 or 11). This corresponds to which bit of the cell's rule is looked up to give the cell its state in the next time step. This means that the rule for this two cell neighbourhood must contain four bits. Figure 3 shows an example of one of the possible sixteen rules, the rule displayed is 1011 (11 in decimal format):

Rule:	Cell:
1	00
0	01
1	10
1	11

Figure 3: The rule 1011 for the example cellular automata

As the cell neighbourhood in question is “10”, we look at the corresponding bit of the rule, which is 1 in this case. So the leftmost cell will be in state 1 in the next time step. This process is then carried out for each cell in the cellular automata, once this is done each cell has a new state and the process in done again for a fixed number of “runs”. As the leftmost cell’s state affects the state of another cell (the rightmost cell) you must make sure that the changing of the cells’ states are carried out in parallel – otherwise some very spurious results may occur.

2.1.3 The Size of the rule/search space

The size of the rule (as a decimal number) is found by the following equation where n is the size of the cell neighbourhood:

$$rulesize = 2^{2^n}$$

This means that the rule space grows very exponentially as the neighbourhood size increases, examples of the rule space sizes are given below in figure 4:

n	Decimal Rule Size	Rule Bit Size
2	16	4
3	256	8
4	65536	16
5	4294967296	32
6	1.84 EE 19	64
7	3.40 EE 38	128
8	1.16 EE 77	256
9	1.34 EE 154	512

Figure 4: The size of rules depending upon neighbourhood

I have left the formal definition of Cellular Automata out of this report, as it is quite mathematically complicated and not necessary to know to understand this report. The formal definition was made by Codd in 1968 [Codd, 1968].

Note in this paper I may refer to the fitness of a cell and the fitness of a rule, in the context I am using they are synonymous – as the cell contains only one rule.

2.2 Basics of Evolutionary Algorithms

2.2.1 Introduction

Evolutionary Algorithms are stochastic search methods that mimic the metaphor of natural biological evolution. Evolutionary algorithms operate on a population of potential solutions applying the principle of survival of the fittest [Darwin, 1866] to produce better and better approximations to a solution.

Normally at each generation, a new set of approximations are created by the process of selecting individuals according to their fitness in the problem domain and breeding them together using operators borrowed from natural genetics.

2.2.2 Genetic Algorithms

In this report we use "Genetic Algorithms" – a form of evolutionary algorithm. Genetic Algorithms are adaptive heuristic search algorithms premised on the evolutionary ideas of natural selection and genetic inheritance. The concept of genetic algorithms was designed by Holland [Holland, 1975]. Although Holland didn't use genetic algorithms to specifically solve problems, much work has been done in this area since 1975. For more information on this, see Tomassini's review [Tomassini, 1995]. A genetic algorithm is an iterative process that consists of a constant-size population of individuals, each of which is represented by a finite string of symbols known as the genome. These strings of symbols encode possible solutions in a given problem space referred to as the search space. This search space consists of all the possible solutions to a problem. Genetic algorithms are usually applied to search spaces that are too large to be searched exhaustively.

2.2.3 Selection and fitness functions

A standard genetic algorithm would proceed as follows: an initial population of individuals is generated randomly or heuristically. Every generation (evolutionary step) the individuals in the current population are decoded and evaluated according to some fitness function (a pre-defined quality criterion). To form the next generation (a new population), individuals are selected according to their fitness. Many selection methods are used, possibly as many as there are people researching genetic algorithms, the simplest being Holland's original fitness–proportionate selection [Holland, 1975]. This is where individuals are selected with a probability proportional to their relative fitness – this ensures that the expected number of times an individual is selected is about proportional to their relative performance in the population. This means high-fitness individuals stand a better chance of "reproducing", while low fitness individuals are more likely to "die". Selection alone cannot introduce any new individuals into the population, i.e., it cannot find new points in the search space. These are generated by genetically inspired operators, of which the most well known are crossover and mutation. [Poli 1999]

2.2.4 Crossover

Crossover is performed with probability p_{cross} (the crossover probability or crossover rate) between two selected individuals, called parents, by exchanging sections of their genomes (encoded solutions) to form two new individuals, called offspring; in its simplest form, sub-strings are exchanged at a randomly selected crossover point. This operator tends to enable the evolutionary process to move toward better solutions in the search space. If we take two 8-bit individuals and their genomes are as follows, the vertical bar shows the selected crossover point:

Individual 1:	1001 | 1110
Individual 2:	1100 | 0011

Then the offspring would be:

Offspring 1:	10010011
Offspring 2:	11001110

These two new individuals would be placed into the new population, replacing lower fitness solutions.

The example shown above (one-point crossover) is only one of many forms of crossover, others include two-point crossover. [Poli 1999].

2.2.5 Mutation

The mutation operator is introduced to prevent premature convergence to local optima by randomly sampling new points in the search space. It is carried out by flipping bits at random, with some (small) probability p_{mut} (usually one in every 1000 bits). One might expect that important information could be lost through mutation (as solutions are being randomly changed), but as long as the mutation rate is not too high, the high fitness solutions propagate and replace the "wrongly" mutated ones.

Mutation can be used as the only genetic operator in a genetic algorithm (i.e. not using crossover), but this is rarely done as it means that every solution has to be checked with its predecessors to ensure a high fitness solution is not lost. This introduces a large overhead in computation. [Poli, 1999], [Yao, 2000].

2.2.6 The standard genetic algorithm

Below is the standard genetic algorithm pseudo-code showing how the details in sections 2.2.1 - 2.2.5 are implemented.

```
begin GA
  g:=0  { generation counter }
  Initialize population P(g)
  Evaluate population P(g)  { i.e., compute fitness values }
  while not done do
    g:=g+1
    Select P(g) from P(g-1)
    Crossover P(g)
    Mutate P(g)
    Evaluate P(g)
  end while
end GA
```

[Sipper, 1996] .

3. Research and existing work in this specialised field

Cellular automata were originally conceived in the 1940s by Ulam and von Neumann. Their purpose was to provide a framework to investigate the behaviour of complex, extended systems [von Neumann, 1966]. Since then there has been a lot of research made into different applications of cellular automata.

3.1 Classes of Cellular Automata

Cellular Automata have often been used to demonstrate the full spectrum of dynamical behaviour from fixed points to limit cycles (periodic behaviour) to unpredictable "chaotic" behaviour. Wolfram considered a classification of cellular automata behaviour in these terms of these categories [Wolfram 1984]. He proposed four classes to attempt to classify all possible behaviour in cellular automata. These are as follows:

Class 1: Almost all initial configurations relax after a transient period to the same fixed configuration.

Class 2: Almost all initial configurations relax after a transient period to some fixed point or some temporally periodic cycle of configurations, but which one depends on the initial configuration.

Class 3: Almost all initial configurations relax after a transient period to chaotic behaviour (chaotic meaning apparently unpredictable space time behaviour).

Class 4: Some initial configurations result in complex localized structures, sometimes long lived.

Most uniform cellular automata that have been used for problems are class 3 or below. Class 4 behaviour basically means that cellular automata can be shown to be a universal computation class. The fact that a lot of cellular automata cannot simulate a universal computation class severely limits their uses.

3.2 A Universal Computation Class

By being a universal computation class it means that some class 4 cellular automata can simulate virtually any feasible task.

If a cellular automaton is a universal computation class it means that it can simulate the same computational power as a Turing machine [Kozen, 1997] – this means that a cellular automaton that is class 4 has the same computational power as any computer. This in itself isn't necessarily too useful, but many cellular automata can do this with a very high level of fitness, which means a class 4 cellular automata can simulate anything any computer can compute.

Mitchell disputes that all class 4 rules have the capacity for universal computation [Mitchell *et al.,* 1993] because of the informal description of class 4 cellular automata.

This is more a debate in uniform cellular automata, as Sipper [Sipper, 1995] has shown that non-uniform cellular automata can simulate Minsky's two-register universal machine [Minsky, 1967].

It was partly because of this evidence that non-uniform cellular automata were used extensively in this project, as to me it implied that non-uniform cellular automata should be able to perform edge detection and other image processing tasks.

3.3 Related research in Cellular Automata to Image Processing

Mitchell carried out a very useful review of current research into uniform cellular automata to solve the density and synchronization tasks in 1996 [Mitchell et al, 1996]. The density task is basically the attempt to change a random initial configuration of cell states all to the majority of the initial configuration. The synchronization task is the similar to the density task but the result is alternate states of 0's and 1's in each consecutive time step. The density task is described in detail in section 6.3.

Both these tasks are examples of cellular automata performing emergent computation. For a system to demonstrate emergent computation the system must use the actions of simple components using local information and communication to give rise to co-ordinated global information processing. Emergent computation is what is described to occur in some natural systems such as hive minds or within our own brain, although we don't fully understand how these natural systems operate emergent computation can be simulated using some artificial systems. Cellular automata are just one of these systems.

Although not directly related, Chady's work [Chady, Poli, 1997] in feed-forward cellular automata is very interesting. He used a cellular automaton with the cell neighbourhood connections implemented in a similar manner to that of a neural network.

In the domain of image processing, Orovas and Austin have extensively researched cellular associative neural networks [Orovas, Austin, 1997a], [Orovas, Austin, 1997b], [Orovas, Austin, 1997c]. This research was of interest as it uses neural networks with configurations designed to simulate cellular automata neighbourhoods. Most of the research was for image interpretation and pattern recognition. Most of the principles applied to the neural networks also follow for 2-dimensional cellular automata.

3.4 Image Processing at a low-level

It became obvious quite quickly that cellular automata would not be able to produce a very high level of image processing. By not "high-level" it is meant that the cellular automata will not produce equations and corner co-ordinates of lines and shapes. What was much more likely to be produced is a "low-level" of image processing – with the cellular automata acting more like a filter. This is useful as the result can be passed to a high-level image processing routine will then easily be able to produce equations of the lines and edge co-ordinates. This conclusion was reached in Poli's research in genetic programming for use in image Analysis [Poli, 1996].

3.5 Conventional Image Processing algorithms

Image processing has been a major topic of research for years. One of the domains of image processing is edge detection. Many edge detection algorithms have been devised over the years. The algorithms renowned as being some of the most effective and efficient are Sobel, Kirsch, Prewitt and LaPlace of Gaussian.

3.6 Relevance of the Work to Artificial Intelligence

One might think that cellular automata are much more akin to a mathematical discipline rather than Artificial Intelligence. A top/down approach is normally associated with Artificial Intelligence where as a bottom/up approach is normally associated with cellular automata – a bridge between the two is really needed. In previous work carried out with cellular automata, they used to be "hard-coded" to produce a solution. A uniform rule was calculated to perform the task that was needed. But recently genetic algorithms have been used to evolve these rules.

The reason image processing was decided for this project was that low level image processing is often a vital pre-processing task for any kind of work in the "vision" field in Artificial Intelligence. If cellular automata could efficiently perform this pre-processing task it may help to advance other fields of research in Artificial Intelligence.

3.7 Limits imposed by current technology

Unfortunately software simulated cellular automata are severely limited by current computer technology. If computers were computationally faster and could process numbers up to 512-bits, then much more could easily be done with software cellular automata. As section 2.1.3 shows, rule spaces grow exponentially with increases in the neighbourhood. If these rule spaces could be processed faster, then much more could be done in image processing.

A hardware based cellular automaton would solve most of these problems – with a processor in parallel for each cell. A few of these do exist but are quite rare, such as CAM-6 and CAM-8.

3.8 Current research into Cellular Automata

From my research it seems as if many prominent researchers such as Mitchell and Crutchfield seem to have given up on research into applications that are directly applicable to cellular automata, like image processing and are trying to solve much more complicated problems using cellular automata. There also seems to be very little work done on cellular automata since 1996. It seems surprising that researchers have not explored many more basic tasks using cellular automata. A similar comment is made by Wolfram [Wolfram 1983] when he was considering cellular automata as a universal computation class, and his conclusions seem to still hold – "The development of a new methodology is a difficult but important challenge. Perhaps tasks such as image processing, which are directly suitable for cellular automata, should be considered first." [Wolfram 1983].

3.9 Concluding remarks

This concludes most of what would be called research in a conventional sense; the rest of this report contains the attempts that were made to implement this research into a cellular automata that is successful at forms of image processing. There is more research done later in this report, but this was as a result of trying to implement a cellular automaton as an image processing system.

4. The basic cellular algorithm

The algorithm that was used as a basis for a large proportion of the work in this project is called "The Cellular Programming Algorithm". The algorithm was developed in a paper titled "Co-evolving architectures for cellular machines" [Sipper, Ruppin 1996]. The algorithm is shown in figure 5.

```
for each cell i in cellular automata do in parallel
        initialise rule table of cell i
         $f_i$ = 0{fitness value}
end parallel for
c = 0 {initial configurations counter}
while not done do
        generate a random initial configuration
        run cellular automata on initial configuration for M time steps
        for each cell i do in parallel
                if cell i is in the correct final state then
                        $f_i = f_i + 1$
                end if
        end parallel for
        c = c +1
        if c mod C = 0 then {evolve every C configurations}
                for each cell i do in parallel
                        compute $nf_i$ (c) {number of fitter neighbours}
                        if $nf_i$ (c) = 0 then rule i is left unchanged
                        else if $nf_i$ (c) = 1 then replace rule i with the fitter neighbouring rule,
                                        followed by mutation
                        else if $nf_i$ (c) = 2 then replace rule i with the crossover of the two fitter,
                                        neighbouring rules, followed by mutation
                        else if $nf_i$ (c) > 2 then replace rule i with the crossover of two randomly
                                        chosen fitter neighbouring rule, followed by mutation
                        end if
                         $f_i$ = 0
                end parallel for
        end if
end while
```

Figure 5: Psuedo-code of the cellular programming algorithm.

This algorithm was the starting point for virtually all of the research done here into cellular automata; from this several different approaches were developed to evolve cellular automata. As it is the basis for the majority of the work in this project it should be included here.

5. The cellular automata simulator

5.1 Designing a cellular automata simulator

In order to carry out any constructive research, rather than just reading what other people have done in the past, an environment to research cellular automata had to be created. Note that the reason that this is being described is because it did take a significant amount of time to code and without this simulator would have been unable to do any kind of useful research. Cellular automata can be implemented using a CAM (cellular automata machine), this is a massively parallel machine with a processor for each cell in the cellular automaton. This is without a doubt the fastest way to run and implement cellular automata but unfortunately there was no access to a CAM for this project, so a software simulation of a cellular automaton was the solution. A brief search for a cellular automaton simulator in the public domain was carried out but a piece of software which had the functionality that was needed, couldn't be found.

So a skeleton piece of code was developed to implement the cellular automata that were being researched. It was decided to restrict the simulator so that it would only implement 2-dimensional cellular automata, as there is really no need for anything else in the domain of image processing. A one dimensional cellular automata would be pointless as to do any kind of complicated processing in 2 dimensions you have to be able to look at pixels in more than one direction. A 3 dimensional cellular automaton may have had potential (possibly to map fitness back through time) but unfortunately the rule space would have been huge. This will be justified later on. The idea of a 3-dimensional cellular automaton only came up when researching a paper and was not something that had even considered when the project was started.

It proved quite hard to have a program that could use any size (2-dimensional) cellular automata and any size rule. The problem with the size was that the corners of a traditional 2-dimensional cellular automata join with the other side producing a toridoral shape, something looking a little like a doughnut. The rule space was a big problem because a 32-bit rule (5 cell neighbourhood) had to use a long integer variable where as a 16-bit rule (4 cell neighbourhood) only required a normal integer variable, when evolution is being carried out speed is one of the most important factors.

It would probably be the equivalent of a project to design a generic cellular automaton simulator in itself, so the main piece of code used throughout the project was still a skeleton piece of code that was easily modified. The simulator that was used throughout this project was all written in Java. The general consensus about Java and any evolutionary algorithms is that they shouldn't be used together. It was found that with the increased speed of computers now and using some clever algorithms that can exploit Java's weaknesses, that Java was fast enough to carry out the research for this project. There was always the option of using native C methods if the computation became too much for Java (there was never this need). In optimising the Java code several sources were used, the most useful were written by Bell [Bell 1997] and Hardwick [Hardwick 1997].

Several pieces of important code are included as appendices to this report – these are a few important procedures for the evolving and running of the cellular automata. Quite a large amount of code was written for this project, most of this is experimental code. This has had frequent changes to its structure and lacks some comments this is because it is less important than the actual research that has been carried out using this code. This type of code is loosely termed as "proto-code".

5.2 Requirements of the cellular automata

Specified here are the requirements of what a cellular automaton must produce for it to be deemed successful as being able to carry out image processing. If most of these requirements are not met by the cellular automaton it will be concluded that they are not a feasible means of image processing at the current time – because in the future with faster computation power it may mean that a task that is currently intractable may indeed become possible.

1) Be able to perform a low-level image processing technique.
2) To do this with a high level of fitness.
3) To do this on a non-trivial technique.
4) To do this in a tractable finite time.
5) To be able to act as a filter and not have to evolve for every image.
6) To be able to process any size image – to be scalable.

A non-trivial technique means an image-processing task such as edge detection or contour tracing, which requires more than just simple image processing. A high level of fitness here means a result that is visually or mathematically acceptable. A tractable finite time here means to be able to process an image in a relatively short time period – such as a few hours.

6. Evolution of cellular automata for image processing

6.1 The game of life

As the very first starting point in research into cellular automata the most well known cellular automaton was implemented – Conway's game of life [Conway, 1976]. A 100 by 100 cellular automaton was used, using the rule that if only two or three neighbouring cells are alive the cell in question stays alive (state is equal to 1) otherwise (less than two or more than three) it dies (state is equal to 0). A cell can only come back to life when surrounded by three alive cells. This is when looking the 8 surrounding cells. There is no problem with looking at the 8 surrounding cells (normally a 512-bit rule) because the object isn't to find the rule - it is to see what results that rule produces.

In the following examples, which demonstrate a few of the possible situations in the game of life, it is the middle cell that is having the rule "run" on it. Figure 6 shows an example of a cell surrounded by three alive neighbours staying alive (the cell's state stays as 1). Figure 7 shows an example of a cell surrounded by one alive neighbour dying (the cell's state changes to 1). Figure 8 shows an example of a cell surrounded by two alive neighbours becoming alive (the cell's state changes to 1).

0	1	0		0	1	0
1	1	0	⇒	1	1	0
0	0	1		0	0	1

Figure 6: A cell in the game of life staying alive.

0	0	0		0	0	0
0	1	0	⇒	0	0	0
0	0	1		0	0	1

Figure 7: A cell in the game of life dying

0	1	0		0	1	0
1	0	0	⇒	1	1	0
0	0	1		0	0	1

Figure 8: A cell becoming alive in the game of life.

It was very useful in testing different array structures and operators to increase the efficiency before the research moved onto a more complicated problem. Quite a lot of time was spent focusing on improving the efficiency.

6.2 Traditional uniform cellular automata

The first serious task attempted was using a traditional uniform cellular automaton, following the example from Mitchell's work [Mitchell et al. 1993]. Standard fitness proportional selection with a population of 100 individuals, one point crossover and mutation were used. A mutation rate of 1 in every 1,000 was used. A solution of fitness 96.4% was evolved, unfortunately as you can tell from figure 10, the solution turned the image into a mostly white image with a little noise. This is not useful at all, but it is a high fitness solution – this is something that proved to be a problem in many of the other cellular automata experiments. Figure 9 shows a graph of fitness against time (generations).

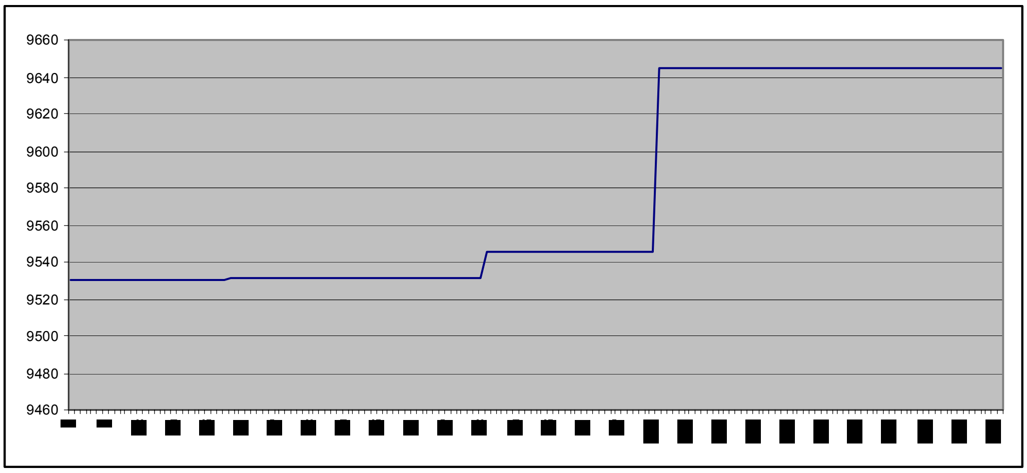


Figure 9: A graph of fitness against time for fitness proportional selection.

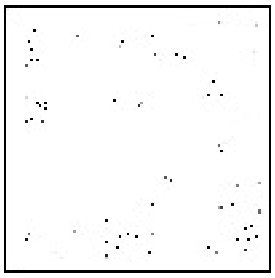


Figure 10: The result of uniform cellular automata edge detection.

One would logical think that to perform edge detection on a generic image, that a uniform rule cellular automaton would be better than a non-uniform rule cellular automaton. As for the cellular automaton to operate with around the same level of fitness on any image it would have to use the same rule on every pixel (as one doesn't know in advance where the edges will be in the image). The reason that non-uniform cellular automata were used in this project was twofold, one because very little work has been carried out using non-uniform cellular automata, and two because the idea, that a non-uniform cellular automaton can find a uniform rule in less time steps, was made. This works on the principle of there being 10,000 different starting rules in a 100 x 100 non-uniform cellular automaton, and it is rare to have a population of uniform cellular automata anywhere near this size. So the search space can be explored much faster.

6.3 Preliminary non-uniform cellular automata experiments

An attempt was made to solve the two-dimensional density task. This is the task of changing the global state of the cellular automaton (every cell) to the majority of black (state equal to 1) or white (state equal to 0) cells. So if there are more black cells than white cells to start with, every cell in the cellular automaton should change to black after running the rules a set number of times – figure 11(a) demonstrates this. Also if there are more white cells than black cells to start with, every cell in the cellular automaton should change to white after running the rules a set number of times – figure 11(b) demonstrates this.

The rules used here had a five-cell neighbourhood, the centre cell in question and the cells above, below, left and right of it. This produces a rule set with rules of the order of 32-bits, producing quite a large search space.

The fitness function for the rules was very simple, if after running the rules through for *M* time steps (*M* being the variable in the Cellular Algorithm in section 4) the state of a cell is in the correct state increase it's the cell's (and hence the rule's) fitness by one.

The results were promising but not one hundred percent perfect, as the actual fitness only reached eighty-seven percent. No doubt if this had been pursued further, results with higher fitness may have been found, but the density task does not really relate to any kind of image processing so continuing to work on it further would be wasteful in the time available. Also in a paper Land [Land, 1994] argued that no two state cellular automata for density classification exists.

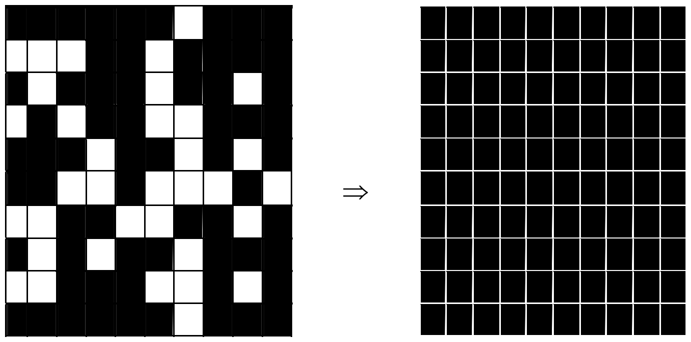


Figure 11(a): The ideal output from the density Task (1)

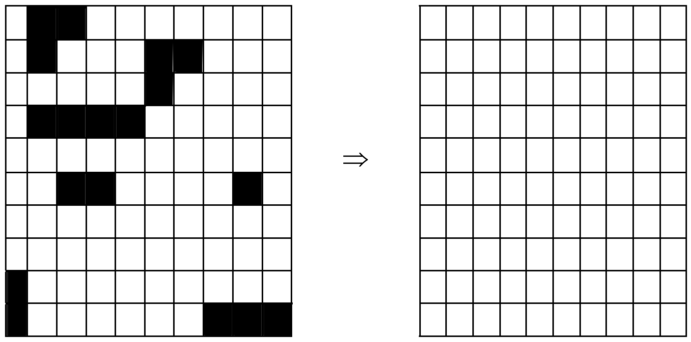


Figure 11(b): The ideal output from the density Task (2)

6.4 Preliminary 100 by 100 edge detection cellular automata experiments

This is where the first attempt at an image processing non-uniform cellular automaton was designed. It was decided upon to focus on edge detection as a form of image processing, because this is quite a low level form of image processing (as described in section 3) and also is an important pre-processing step in many Artificial Intelligence disciplines (as described in section 3). The code from the density task was used to do this. Instead of the cellular automaton having random starting states, it was now given a 100 by 100 pixel image as its starting states – each pixel mapping to a cell in the cellular automaton. The format that was decided to be used meant that the image format the cellular automaton program would accept was in raw format. It was decided upon to use raw format as the file can be read into a two-dimensional array with each array element corresponding to the 256-greyscale level of each pixel.

As the rules are binary rules, and hence have to operate on binary states, the greyscale image is converted to a binary image while my program is reading it in. Anything below a 128 value goes to a state of 0, and anything above 128 goes to a state of 1.

As a fitness function it was decided upon to use the edge detection results on the inputted image from Adobe Photoshop and follow the implementation of the cellular programming algorithm exactly. This was just for these preliminary experiments.

No mutation was implemented here, partly to see what the effects of just using crossover were.

The rules used the same neighbourhood configuration as in section 6.3, figure 12 demonstrates which cells are used. The rule used is the one in the centre cell, so if cells 1 2 3 4 5 had states 1 0 1 1 0 respectively, then bit 21 of the rule is the cell's state in the next time step (21 instead of 22 (10110) because the 32-bit rule lookup is from 0 to 31 and not 1 to 32).

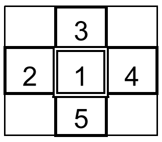


Figure 12: The 5 neighbouring cells used to lookup a specific bit in a 32-bit rule

The results from this were not too inspiring, 82% fitness after 3000 generations; the cellular automata seemed to vaguely find a few of the edges but there is a lot of noise and would be fairly useless for any higher purpose. The results are shown in figure 14, figure 13 shows what the cellular automaton is attempting to find. It was decided not to evolve this any further as it was unlikely to improve.

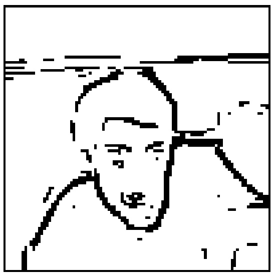


Figure 13: Edge detection algorithm

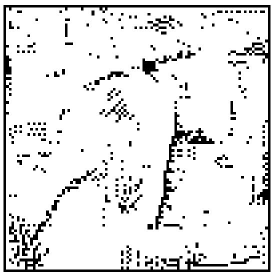


Figure 14: Cellular automata result.

6.5 Non-uniform cellular automata experiments

There are essentially two different tasks to solve here. The first is to get a cellular automaton to learn the rules to produce the edges of a specific image. The second is to get a cellular automaton to learn the rules to be able to be used on any image – like a filter. The problem with the first method (effectively evolving the pixels of the image) is that although the results from it may be very good it is not particularly useful because of the variable time taken to reach a solution.

The task of solving the first method is much easier than the second so I set out to solve the first method first which will be called static, and then move on to the second which will be called generic.

6.5.1 Static Image Edge Detection with crossover and mutation

After a little thought the problem with the method described in section 6.4 was probably to do with the local maxima being found – which could be totally incorrect solutions. On an "average" image existing edge detection algorithms produce an image that is mainly white (or black if inverted) with a few pixels black for the edges. So a solution that is 80% fit (i.e. 80% of the cells end up in the correct state) could be one where rules are evolved that just change any state to be white in the next time step – so you end up with a completely white image, which isn't useful at all. One solution to this was to introduce mutation, as this aids in traversing the rule space quicker and to produce more varied rules. This was one of the features that Sipper implemented in his cellular programming algorithm [Sipper, Ruppin 1996].

After implementing mutation this was the result from the improved cellular automaton. The ideal edge detection is shown above in figure 13 and the results, after 3000 generations, from the currently described cellular automaton had a fitness of 88% and are shown in figure 15 below. Although the evolution carried on for 3000 generations, the 88% fitness result was found in 400 generations, the evolution was continued just to make sure that no fitter rules could be found.

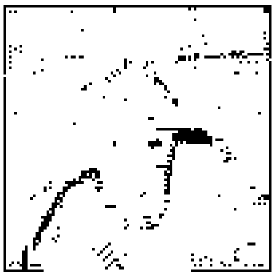


Figure 15: Cellular automata result with mutation.

As you can see by comparing figure 13 and figure 15, the cellular automaton edge detection is closer to the standard edge detection algorithm than figure 14 (without mutation) and has distinguished some of the edges. There is less noise on the image, and the edges have not really been reduced to particularly fine lines. The reason that mutation improved the result is that mutation adds randomness into the rules and helps the cellular automaton evolve from local maxima towards the global maximum.

Following from this the mutation rate was doubled from the standard one in every thousand bits being flipped to one in every 500, this hopefully would serve to get nearer to the global maximum. The results after 3000 generations where very promising, at a fitness of 90%. Figure 16 shows that although the fitness was only raised by 2% the edges are much clearer with a lot less noise in the image. Also of the finer lines are also being distinguished. The time that the cellular automata took before the fitness stopped improving was a lot longer – 1,600 generations compared to 400 before.

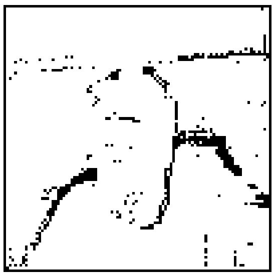


Figure 16: Double the mutation rate.

The fact that doubling the mutation rate improved the result means that it should follow that having the mutation rate should decrease the fitness. Figure 17 shows the effect of doing this so that the mutation is only one in 2000, the fitness was 87% after 3000 generations. These trends follow from what was learnt in the Evolutionary Computation module [Poli 1999]. It also supports the conclusions that Mitchell made [Mitchell *et al.,* 1993] about crossover only taking the fitness to a certain level, with mutation taking the fitness passed this point.

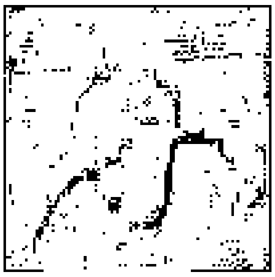


Figure 17: Half the mutation rate.

6.5.2 Preliminary generic image edge detection

Although what was attempted next only served to make the task more complex for the cellular automata it was attempted because it was necessary to meet the requirements outlined above. This was to present the cellular automata with different images and evaluate its fitness, so the cellular automaton could be used as a "generic filter". This required modifying the cellular automata simulator to be able to read in multiple images while evolving, this was a simple task, this meant changing *C* in the cellular programming algorithm to the number of images being fed in, which was 10. As each generation had to now do ten times more operations, this made the code ten times slower. The results were as expected, very poor. Figure 18 shows them. Figure 19 shows a graph of fitness over time for this evolution.

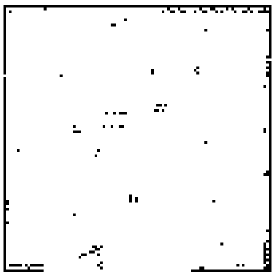


Figure 18: Generic image cellular automata result.

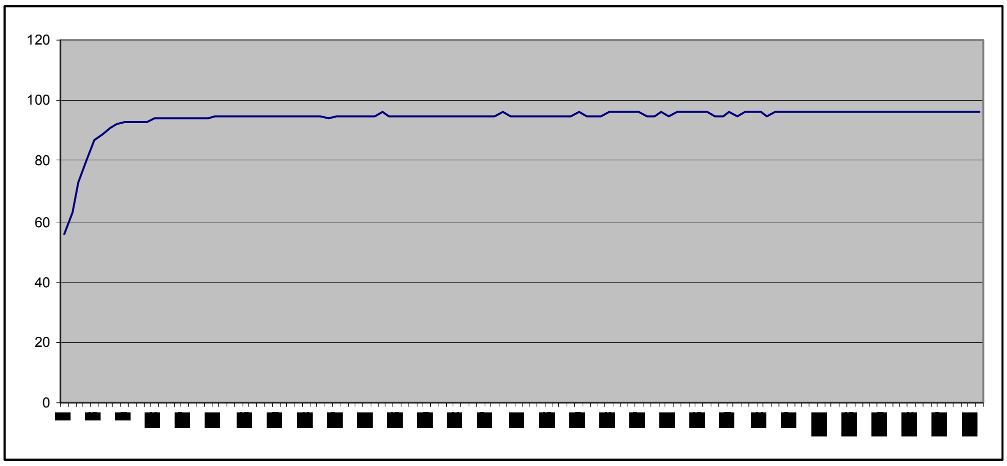


Figure 19: Fitness over time for generic edge detection using 32-bit rules

As using a cellular automaton to detect edges seemed like a too complex task to complete it was decided to attempt a simpler task, which was still technically an image processing task, the thinning task – trying to reduce a thick rectangle to a thin line. By using a simpler task it is easier to gauge the effectiveness of the cellular automaton more easily.

6.6 Using cellular automata for the thinning task

6.6.1 Introduction

Thinning (also known as skeletonization) is a fundamental pre-processing step in many image processing and pattern recognition algorithms. When an image consists of strokes or curves of varying thickness, it is usually desirably to reduce them to thin representations, located along the approximate middle of the original figure. Early examples using parallel systems were presented by Preston and Duff [Preston, Duff 1984]. These thinned images were shown to be easier to process in later stages by Guo and Hall [Guo, Hall 1989] – who used a sub-iterative algorithm. An example of the thinning task is given in figure 20, where (a) is the image to be thinned and (b) is the result.

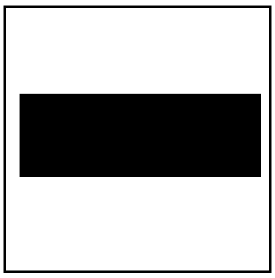


Figure 20 (a) Thinning task input

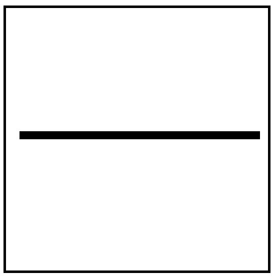


Figure 20 (b) Thinning task ideal output

6.6.2 Static horizontal thinning

Using the same setup of the cellular automaton that was used in section 6.4, the cellular automaton was set to solve the thinning task, using figure 20(a) as the input and figure 20(b) as the fitness test. Figure 21 shows the result of the cellular automaton acting upon the image after 3000 generations – the fitness of the result is 97%. There is still some random noise produced by the cellular automaton, but the thinned image is good enough to be able to be used by a higher-level image processing technique that requires a thinned input. How well the cellular automaton works as a general filter (on any position rectangle) was investigated next.

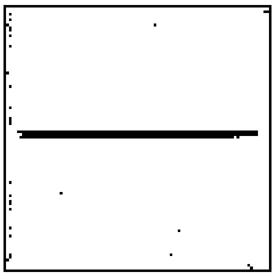


Figure 21: The cellular automata output for the thinning task.

6.6.3 Generic image thinning in both horizontal and vertical planes

This also allowed the possibility that vertical lines can be found, as Guo and Hall [Guo, Hall 1989] did in their research. I started with a set of 20 images: 5 single horizontal lines, 5 double horizontal lines, 5 single vertical lines and 5 double vertical lines. Using a mutation rate of 1 in 500 bits and evolving for 10,000 generations a result of 94% fitness was produced. This seems a high figure, unfortunately figure 22 shows that this high fitness does not result in a particular useful solution.

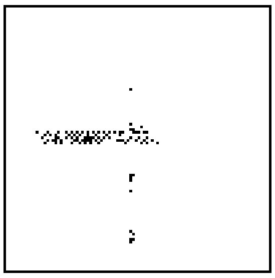


Figure 22: Cellular automata output after learning on generic images.

6.6.4 Varying the fitness function with generic thinning

Next the fitness function was investigated. Previously the fitness function worked as follows: if after 100 time steps the cell was in the correct state that cell's fitness was increased by one. This should be done for C different starting configurations; unfortunately only one configuration was being feed in – so by increasing C (in the algorithm) the only effect would be to inflate the fitness of cells. It doesn't help testing the same thing ten times rather than one time – you will get directly proportional results. So to try and find rules that thin well, black pixels in the correct state propagate more - the fitness function was changed to add two to a rule's fitness rather than one, like if a rule produces a white pixel in the correct state. The results are shown in figure 23, which had a fitness of 94% but after only 1,500 generations (the fitness did not improve after another 5,000 generations). This was half the evolutionary time of the un-altered method. Then the same process was tried with three being added to the fitness of an "edge finding rule" and then again with adding five. The results were worse (in terms of time, not fitness – which was the same) than that of figure 23, but took 2,650 generations and 3,720 generations respectively. Adding five to each rule actually slowed the evolution – this is because certain rules that were supposedly good at finding edge pixels but not white pixels were artificially inflated in fitness and took time to overcome.

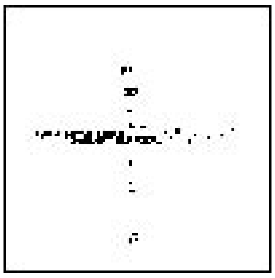


Figure 23: Cellular automata output after varying fitness function on generic images.

To hopefully increase the fitness of the cellular automaton it was decided to add fitness penalties. This meant to penalise the fitness of a cell if the cell was in the incorrect state. Using a fitness penalty of 1, 2 and 5 were attempted. The best result is shown in figure 24, with a fitness of 95% with a fitness penalty of 1.

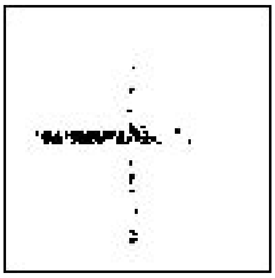


Figure 24: Cellular automata output after penalty fitness function on generic images.

Next, this penalty system was refined, so that a penalty was only given if the cells state was wrong when at an edge – when the cell's state should have been black. This was to attempt to combat the high fitness solutions that produced a completely white image. Using a fitness penalty of 1, 2 and 5 were attempted. Some of these worked slightly better giving fitness's of 96%, 95% and 92% respectively. The fittest solution is shown in figure 25. Also the average number of generations taken until the solution was evolved was reduced to 1,170 using a penalty of 1.

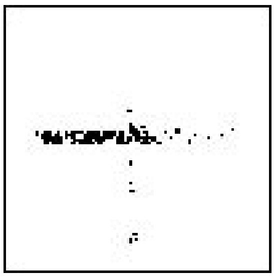


Figure 25: Cellular automata output after refined penalty fitness function on generic images.

6.6.5 Varying *M* – the number of time steps to utilize rules

Next the effect of varying *M* in the cellular programming algorithm was investigated. *M* is the number of time steps that the rules are run for before evaluating the fitness of the each cell's fitness. In Sipper's work [Sipper, Ruppin 1996] he came up with the value of *M* to be the same as the number of cells in the cellular automaton – in the current cellular automaton this would be 10,000. The value of *M* matters, as it is how long it takes to get cell's state to the ideal state using a near ideal rule. In most of Sipper's work he was implementing tasks similar to passing a signal from the left hand side of the cellular automaton to the right hand side using only one-dimensional cellular automata. This meant that *M* had to be about the size of the number of cells otherwise a signal could not be propagated from one side of the cellular automaton to the other, to produce an output. In the cellular automata used in this project, having a larger cell neighbourhood to look at means that the connectivity between the cells is dramatically increased. Various values of *M* were implemented on static images, from 1 to 10,000. The best result was 96% using 100 as a value for *M*. Figure 26 shows a graph of fitness over time for the values of *M* being 100 implemented. More graphs for other values of *M* are shown in appendix 3.

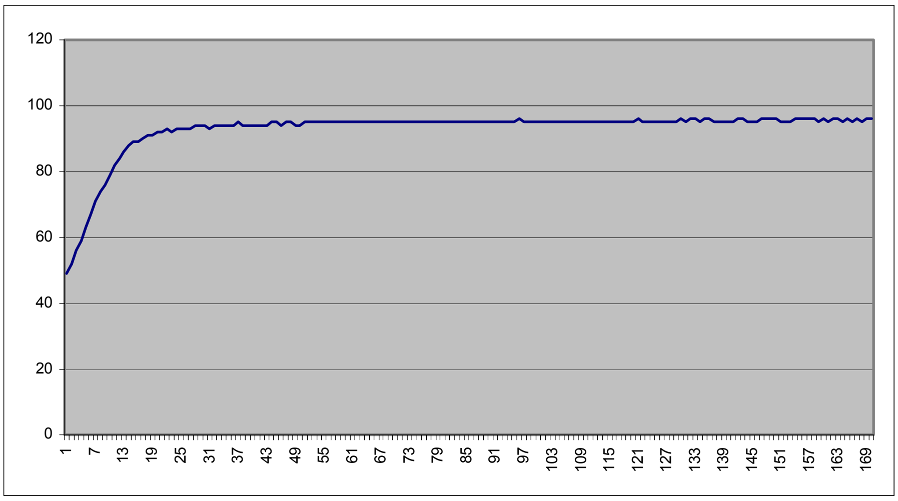


Figure 26: A graph of fitness against time for $M = 100$

6.6.6 Concluding remarks on this section

From all of the previous results from the cellular automata it was obvious that either a new approach was needed or that cellular automata are not very good at image processing. Since there were a lot more approaches that could be attempted accepting the later was not an option. One major consideration was the size of the rule space – about 4.2 billion rules. It was possible that the ideal rules were not being found due to the immense size of the search space. To reduce this meant looking at less neighbours. So the cellular automata were changed to only look at the centre cell, the cell below, the cell to the left and the cell to the right. This reduced the rule size from 32-bit to 16-bit, from 4.2 billion to around 16 thousand.

6.7 Reducing the rule space

Using a 4 cell neighbourhood of the centre cell, the cell below, the cell to the left and the cell to the right dramatically reduced the search space as mentioned in section 6.6.6. The results though were just seemingly random pixels; figure 27 shows the result from the cellular automaton with this neighbourhood configuration.

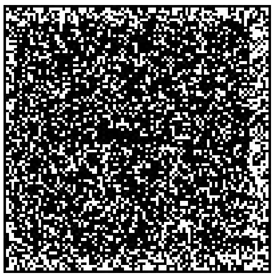


Figure 27: An attempt to reduce the rule space.

Although the search space had been dramatically reduced, cellular automata still seemed unable to perform thinning at a high fitness level. There still seemed to be a fundamental problem. This problem was discovered and solved in section 6.10.

6.8 Scalability

For cellular automata to be useful as a general means of image processing, the rules would have to be scalable. This means that whatever rules a cellular automaton contains must be able to be "magnified" or scaled to a larger cellular automaton (or reduced to a smaller cellular automata). This would allow a cellular automaton to perform image processing on any size image.

The task of scalability should be easy for uniform cellular automata, as the rule can be copied into all the cells in any size cellular automaton, but this is not the case. This simple scaling does not bring about task scaling – the performance decreases as grid size increases. This is demonstrated by Crutchfield and Mitchell [Crutchfield, Mitchell 1995]. For non-uniform cellular automata scalability can be much more of a problem as the rules often work in combination in large numbers. This can mean that some non-uniform cellular automata may not be scalable.

This is confirmed by Sipper and Tomassini's work [Sipper et al. 1996b], [Sipper et al. 1996c].

6.9 Thoughts on three-dimensional cellular automata

It is possible to have a 3-dimensional cellular automaton. Although its connectivity at edges is hard to visualise it can be represented in a computer. Very little research has been carried out in 3-dimensional cellular automata, this is for the reasons given below.

For a 2-dimensional cellular automaton to produce any kind of ordered behaviour the rules have to examine at least 3 cells, the cell in question and 2 others. This produces a rule of 2^8 in magnitude (by how I have set out the rules above). If we extend this to three planes we have to look at 3 cells in each of the 3 planes, this produces a rule using 9 cells and of the magnitude 2^{512}, this is too large to be able to do any real work on as representing and working with rules this size is extremely slow in computational terms and requires special number systems to be programmed.

The reason one might want to use 3-dimensional cellular automata is that Chady's feed-forward type of cellular automata [Chady, Poli, 1997] could be implemented as the third dimension of the cellular automaton.

6.10 Using diagonal cells for thinning

There was a fundamental problem that was stopping cellular automata from performing their image processing at a high fitness level. From carefully examining how other edge detection algorithms operate, the one similarity most have is operating along diagonal lines. So the next design of neighbourhood in cellular automata that was implemented used a diagonal cell. Adding the diagonal cell to the original model would have increased the rule size to 64-bits. Ideally a using all 4 diagonal cells would have been implemented but unfortunately this would have increased the rule space to 512-bits. This was unacceptable so the model used in section 6.7 was modified to look at a diagonal cell, this meant the neighbourhood for the rules consisted of the centre cell, the left cell, the cell below and the diagonally left-down cell – causing a 16-bit rule to be used. Also it was realised that the edge cells in the cellular automata did not really want to be linking back to the cell on the opposite side – in a toridoral manner. This is because the cells that where being used to determine the state in the next time step had no relevance in the image itself – so imaginary white state cells were used instead in the side and corner cells' neighbourhoods. Figure 28 shows the effect of both of these methods.

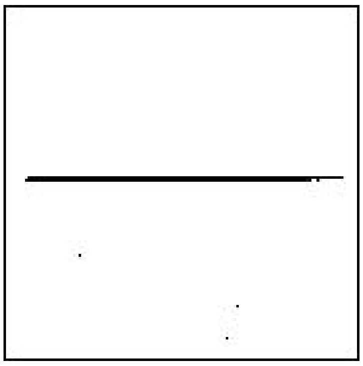


Figure 28: Diagonal cell neighbourhood performing the static Thinning task.

This solution was much better than anything previously developed. It was of a much higher fitness 98.6% in 1,000 generations. Of course this was still using a static thinning task – only one image was being presented to the cellular automaton. Next a generic thinning task was implemented – so any image could be presented to the cellular automaton. The results were surprisingly good, although it took 2,300 generations to evolve it to a fitness of 97.9%. The result is shown in figure 29, as can be seen, there is more noise than in figure 28.

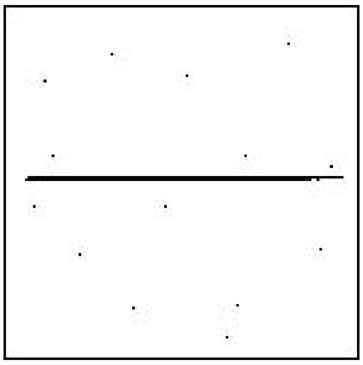
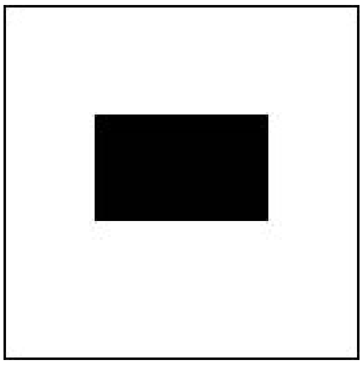


Figure 29: Diagonal cell neighbourhood performing the generic Thinning task.

It was decided to attempt to move back to edge detection, as a higher percentage solution to the thinning task would not help to prove anything new.

6.11 Using diagonal cells for edge detection

The images used in the diagonal neighbourhood cellular automaton were chosen to be much simpler – black rectangles. This was for two reasons, one so that the effectiveness of the edge detection could be measured more easily and so that an algorithm could generate a virtually infinite supply of test data for rules to evolve on. Static image edge detection (just one image presented) was attempted first. The results were of a high fitness, but as figure 30(b) shows, missed 50% of the edges. Figure 30(a) is the inputted image.



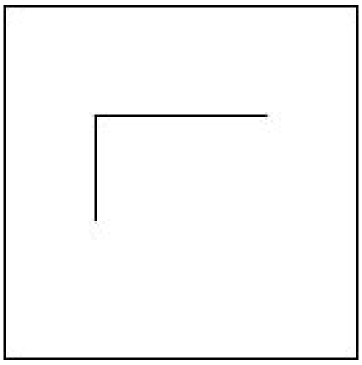


Figure 30(a): Diagonal edge detection input Figure 30(b): Diagonal edge detection result

From this there was obviously a problem with my diagonal cell neighbourhood. Because the cell neighbourhood only looked at a small section of cell states for each cell in question only the edges in two directions were being found. This was where the sub-iterative algorithm was developed to solve this problem – this is shown in section 6.12.

6.12 A sub-iterative algorithm

To try to use more of the cells in the cell neighbourhood the following idea was used: rotating the image 180 degrees and performing edge detection on both the original and the rotated image and combining the results. To combine the two resultant images an "or" operator was used. This produced a higher-level fitness solution than that in section 6.11. The result was a fitness of 100% shown in figure 31 below.

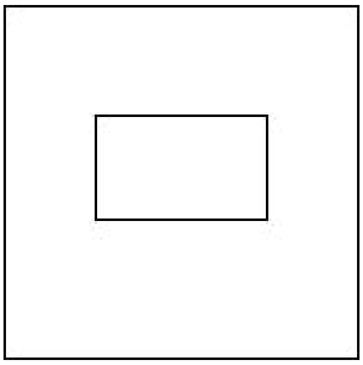


Figure 31: Diagonal edge detection result with rotation

From here, generic edge detection was attempted – edge detection on any image. After a slightly longer evolution period, than with the static method, a high fitness solution was developed. This was trained on generic rectangles and their outlines as the edges. On the rectangles themselves the fitness was still 100% - hence an example of a result is the same as figure 31. The fitness of the edge detection on other images was hard to measure – but in how near the result was to existing algorithms produced a similarity of 76%. Note this is not a level of fitness; this is just how similar the cellular automaton method of edge detection is to conventional edge detection. This is discussed more in the appraisal - section 8.2.

Figure 32(b) show the result of the cellular automaton edge detection on figure 32(a), figure 33 shows the output of a common edge detection algorithm on the same image.

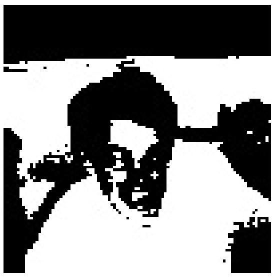


Figure 32(a): Inputted Image

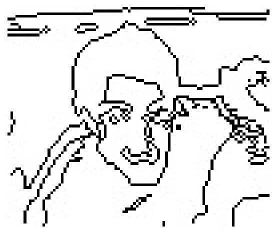


Figure 32(b): Sub-iterative result

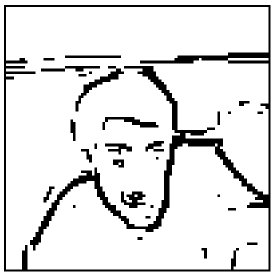


Figure 33: A conventional algorithm

This method was then increased in efficiency by one vital feature of the non-uniform cellular automata – the fact that is wasn't totally non-uniform. The fittest results consisted of either two rules working in tandem or one single uniform rule. This premise was outlined in section 6.2. So to increase the efficiency a much smaller cellular automaton was used – a 3 by 3 grid with 9 cells. The image was read in as 5 by 5 chunks – so that each cell had the correct number of neighbours and then reconstructed at the end of the process. This caused a massive decrease in computation time and caused a high fitness solution to be found in less than 500 generations. The algorithm was further optimised by performing the effective rotation in the processing of each run of the rules of each cell – this is by this method was named a sub-iterative algorithm – two runs of the rules were performed in each iteration of each cell. This algorithm is outlined in appendix 8. Because of how this algorithm operates using either one rule or two-rules working in tandem, it means that the cellular automaton is scalable to any size image. Graphs showing the differences in fitness over time for the original algorithm and the improved algorithm are shown in appendix 3.

6.13 Greyscale edge detection

Next, greyscale edge detection was attempted. There were two methods that were implemented. The first method used my previous work so far directly and the second method was quite different to what had been attempted so far.

The first method was to split the image into 8 different images based on 8 boundary levels in the 256-colour greyscale original image. Each of these 8 images had my edge detection cellular automaton "run" on them. Then the 8 images were recombined into one using an elaborate nested "or" function. Unfortunately the result of this seemed to just threshold the image into 8 greyscale levels – producing a 8-colour greyscale shaded image – which was no nearer to the edges than before the operation was performed.

The second method was involved not using a one to one mapping of one pixel to one cell. Instead each pixel was changed from its original 256-greyscale level to a 16-level greyscale value. Then for each pixel 4 cells were used to encode this value from 0 to 16 into 4 cells' states. Figure 34 shows this. Figure 34 has an image on the left consisting of 4 pixels each with a greyscale level between 0 and 15, on the right of the figure there is a 4 by 4 grid of 16 cells. Each 4 pixels surrounded by a bold line represent the value of one pixel in binary – read top left, top right, bottom left, and bottom right. Then at the end of the process the resultant states of the cells are decoded back into greyscale values between 0 and 15.

15	8
9	6

→

1	1	1	0
1	1	0	1
1	0	0	1
1	0	1	1

Figure 34: One pixel to 4-cell mapping.

Both the original 32-bit neighbourhood and the 16-bit diagonal neighbourhood were implemented and gave very similar results – which was a totally meaningless image. The result from the 16-bit diagonal neighbourhood is shown in figure 35(a) and the ideal result is shown in figure 35(b). The problem was probably to do with the fact that signals have to be passed across the image and these signals need a greater radius than one cell. Unfortunately to do this would result in rules that are massive in bit size – which would make them infeasible.

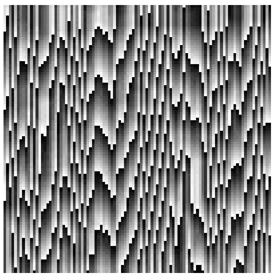


Figure 35(a): Greyscale attempt of edge detection

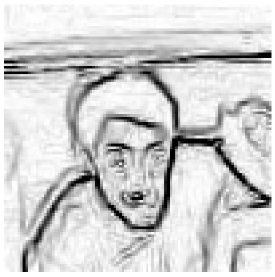


Figure 35(b): Ideal result

6.14 Three-dimensional cellular automata with time as the third dimension

This was an idea that was thought of during the course of this project, but unfortunately the time was not available to implement it – it is a possible extension to this work.

If 32-bit rules can be successfully implemented and are computationally feasible as has been shown, then it brings the scope for using a 16-bit rule 2-dimensional cellular automaton with an extra cell neighbourhood in a third dimension bringing the system to 32-bit rules. This third dimension would be the current cell's rule, state and/or fitness in the previous time step. This would allow a fitness through time selection for the rules. This would give another variable for selecting fit rules and would also allow a mutation only operator to be implemented effectively.

7. Final Results

This section shows a brief selection of results from the best image processing method developed – the edge detection using the sub-iterative algorithm in section 6.12. The images used here are for more detailed analysis of how good my edge detection method actually is. A larger selection of resultant images are shown in Appendix 2. The fact that the edge detection works on varying size images means that the cellular automaton's rules are scalable.

Figure 36(b) shows the results of my edge detection method on shapes, with figure X36(a) being the inputted image.

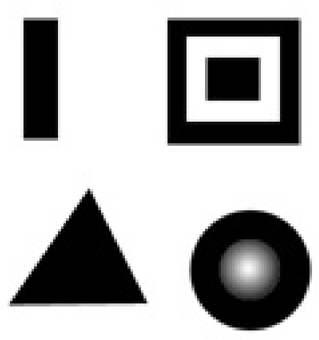


Figure 36(a): Inputted shapes

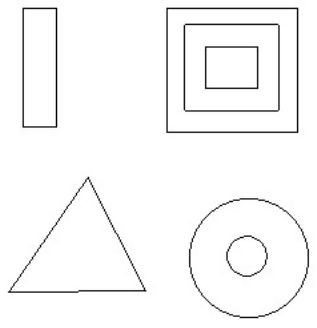


Figure 36(b): My edge detection method

Figure 37(b) shows the results of my edge detection method on a natural image (a photo), with figure 37(a) being the inputted image.

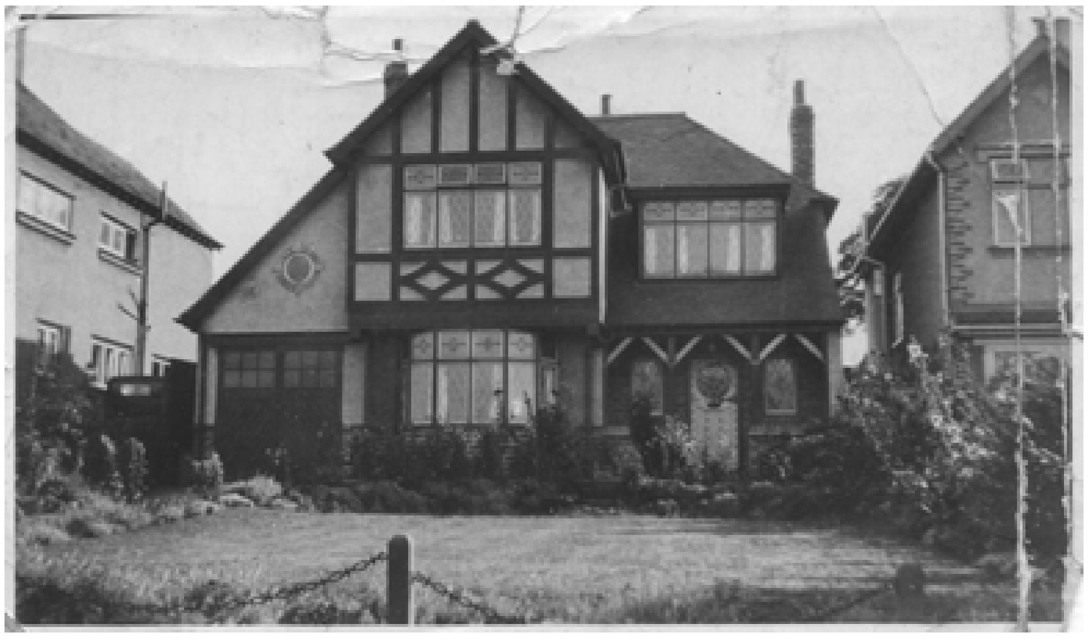


Figure 37(a): Inputted photo

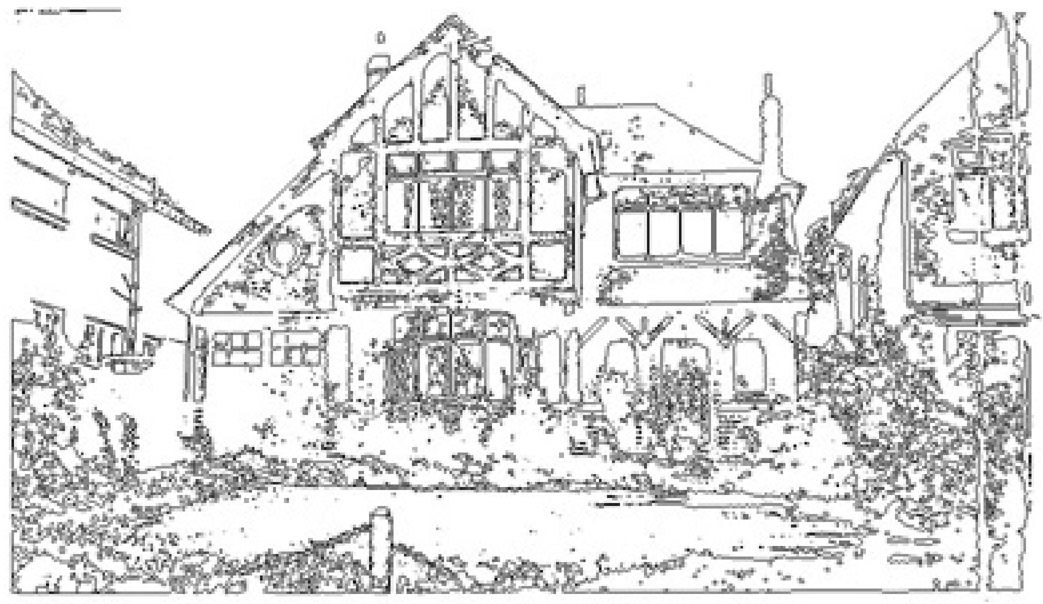


Figure 37(b): My edge detection method

Figure 38(b) shows the results of my edge detection method on text, with figure 38(a) being the inputted image.

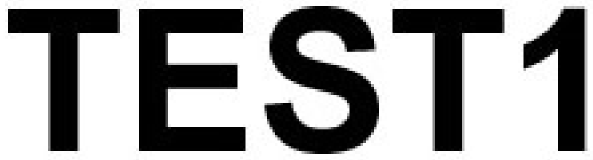


Figure 38(a): Inputted text image

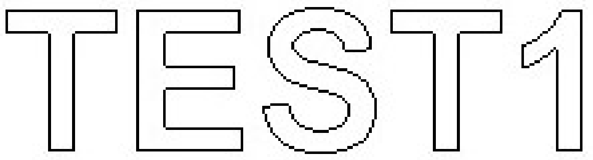


Figure 38(b): My edge detection method

8. Appraisal

8.1 Appraisal of my specification

A copy of the specification for this project has been included in Appendix 6.

Because of the fact that this is a research project, the specification depended on the results of some of that research. For this reason the specification that was written before much research had been carried out. This meant that certain parts of the specification, especially the work-plan were not particularly accurate.

8.1.1 Appraisal of the Objectives

The specification proposed that the majority of the research would be carried out into using cellular automata for edge detection - this was not the case. A large amount of research was carried out in other areas, such as the thinning task and the density task. But this research was leading to a better edge detection solution.

In the specification the main aim was outlined as whether cellular automata are a feasible means of carrying out edge detection in images. This was because edge detection is a complicated image processing technique, and if cellular automata can perform edge detection they can probably perform other forms of image processing. The research presented in this project shows that cellular automata can definitely perform a level of image processing; edge detection and thinning to a high degree of quality in binary images. The criteria set out as "a feasible means" have been met – to be fast and efficient, this is covered more in section 8.2.

Cellular Automata may not be able to perform image processing on greyscale images easily, but I believe with further research this should be possible. As was written in the specification, greyscale image processing was an optional extension to this research if time permitted.

The specification states that the findings were to be kept on the Internet – throughout this project updates of each stage that was reached were kept on my web site.

8.1.2 Appraisal of the Implementation

In regard to the implementation section of the specification, mainly binary images were used for the cellular automata to perform image processing. With regard to the two possible extensions of greyscale and colour image processing: Greyscale level images were attempt but with no success, and colour level images were not attempted due to the fact that greyscale image processing would have to be completed to implement colour image processing. Colour images in the specification meant RGB colour images.

The actual programs that have been written work in a very similar manner to the process outlined in the specification – the image is broken down into chunks, the image processing performed and then re-constructed into the resultant image.

The aim of displaying the original image to the user along with the edges of that image was not met. This was because the graphic user interface was not really an important part of the project especially because the code written for this project was “proto-code”, or loosely termed as experimental code. The aim of this project was not to produce a ergonomic graphic program – it was to research cellular automata and their uses for image processing.

A simulator was written as part of this project as the specification planned. The simulator is a skeleton piece of code with important efficient procedures.

8.1.3 Appraisal of the key stages

All but two of the key-stages outlined in the specification were met. Of the two stages not met, one was greyscale images and this was attempted. The other was the graphic user interface; section 8.1.2 describes why this was not implemented.

8.1.4 Appraisal of the fall back positions

Of the three fall back positions outlined in the specification, only one had to be used – this was the fallback from greyscale images to binary images, this is explained in section 8.1.1.

8.2 Appraisal of my image processing results

This section is an appraisal one how good the results of cellular automata image processing actually are especially with respect to existing algorithms and with respect to the criteria set out in section 5.2.

8.2.1 Appraisal with respect to my criteria

In section 5.2, six criteria were set for cellular automata to be deemed successful for image processing.

The first criterion was to be able to perform a low-level image processing technique – this has been met as section 6.10 shows cellular automata performing the thinning task, also section 6.12 shows cellular automata performing edge detection.

The second criterion was to be able to perform the first criterion to a high level of fitness. This has been met, as section 6.12 shows cellular automata performing edge detection on rectangles with a fitness of 100%.

The third criterion was for the first criterion to be a non-trivial technique – this has been met as section 6.12 shows cellular automata performing edge detection, which is a non-trivial technique.

The fourth criterion was to be able to perform the first criterion in a tractable finite time – this has been met as cellular automata in section 6.12 performs edge detection in a time period of a few seconds – depending upon image size.

The fifth criterion was for the cellular automaton to be able to perform the task in the first criterion on generic images – so that the cellular automaton does not have to evolve for every image presented to it. Section 6.12 shows a cellular automaton that is capable of performing edge detection on any image presented to it.

The sixth criterion was for the cellular automaton to be able to perform the task in the first criterion on any size image – to be scalable. This is shown to have been met in section 7, where different size images have edge detection performed upon them.

8.2.2 Appraisal with respect to other algorithms

This section addresses actually how good my edge detection method/algorithm is compared to other existing algorithms. It is noted here that the purpose of this project was not to develop a better edge detection algorithm than existing ones – the purpose was to try to develop an approximation of other algorithms to prove cellular automata are capable of image processing.

The best edge detection method developed here was the process outlined in section 6.12. This method is no longer an approximation of other edge detection algorithms - it is another different method in its own right.

There are two ways my method can be compared to other algorithms, one is to compare my method with respect to other algorithms' performance on binary images and the other is to compare my method with respect to other algorithms' performance on greyscale images. The latter can be carried out by changing the threshold value of the greyscale level that means a pixel will be a white or a black pixel in the binary image my method uses – this can increase the detail my method can find in an image. I will make comparisons on three types of images – synthetic, natural and text images.

Unfortunately there is not a mathematical why of comparing the well-known edge detection algorithms to my own, as to do that one would need to know what the best possible edge detector is in all situations. A standard deviation could be applied between my resultant image and that from another algorithm – but this would just tell you how different my image is to another – not compare the actual edge detection. This means one is reduced to using a visual comparison.

Synthetic images are those that are artificially created, normally simple shapes. Appendix 2.1 shows examples of my edge detection method on synthetic images, as does section 7. The first image in appendix 2.1 is the inputted image containing the shapes, the second image is the result of my edge detection and the third image the result of a conventional edge detection algorithm. The second image shows that my edge detection method can find edges on rectangles, triangles and circles. In comparison to the conventional algorithm, my method produces much finer lines – only one pixel wide, also my algorithm is not confused by the circular gradient (although this is a result of the threshold of the greyscale levels).

Natural images are ones that are normal photographs like the two examples in appendix 2.2, the aerial photo and the photo of a house. The aerial photo has the results of four of the well-known edge detection algorithms applied on it as well as my own. On this kind of image the LaPlace of Gaussian is quite useless (it is better with synthetic shapes), my edge detection makes it easier to distinguish the edges than Sobel's and Prewitt's edge detection. Although the outline of the road is lost in my edge detection, objects like the house is easier to distinguish from surrounding objects. From visual evidence, only Kirsch's detection seems to be better than mine on natural images. The other algorithms do have an advantage as they operate on the full greyscale range.

To give the edge detection presented in this project a fairer chance, the threshold value can be altered – this causes more of the detail to remain when the greyscale image is converted to a binary one. Unfortunately it can also cause the edges to blur into each other. In appendix 2.2, the second set of images, those of the partially timbered house, demonstrates the effect of changing the threshold value. The first image is the 256-level greyscale photo of the house; the second is the image with a threshold at 128 – this is what is feed into the cellular automata. The sixth and final image shows a conventional edge detection algorithm for comparison to my own, the fifth image is the output of the conventional algorithm on the binary version of the original image. The third image is the output of the cellular automaton on the 128-threshold image, it is virtually as good as the conventional algorithm's attempt (the fifth image) on the binary image, but does not come close to being as good as the sixth image – the full greyscale edge detection by the conventional algorithm. This is due to the fact the 128-threshold image has lost about half of the image when it is inputted to the cellular automaton. To improve the results from the cellular automaton, the threshold was changed to 196 (out of a possible 256). The edge detection from the cellular automaton was the fourth image – this has distinguished the edges on objects on the house far more effectively than the conventional algorithm (the sixth image), although the cellular automaton's output has far more "noise" now.

The text images shown in appendix 2.3 are images containing various fonts of text: Arial, Times new roman and script. The original image, my methods result and a conventional algorithm's result is shown for each font. My edge detection has one distinct advantage over conventional methods - it produces fine lines. This means that small text would not lose the detail of the edges – like for example on the serifs of the Times new roman text. The conventional algorithm has great trouble displaying the edges at the thin sections of the text – where the serifs are. My edge detection still performs well even with the script font; this demonstrates that it will work as effectively on curves as on straight lines.

Even though the cellular automaton cannot distinguish different levels of grey, only black and white, it has outperformed several of the conventional algorithms on binary images and has made a close approximation of the edges on greyscale images.

The major drawback of the cellular automaton's edge detection is the lack of greyscale in the resultant image, as conventional algorithms use this to emphasize certain features more than others. But in terms of binary edge detection the algorithm presented here performs as well as most conventional edge detection algorithms, if not outperforming a few.

9. Possible extensions to this research

If this research could be extended to the use of greyscale (and then colour) images it may be very useful as a plug-in for graphics packages and even more useful in image recognition as clear edges are often a vital step in recognising features in images.

What this research leads more towards in the future is the use of two-dimensional cellular automata in other applications other than image processing, as image processing is not really a computationally intensive enough task to use the full "power" of non-uniform cellular automata so following is the outline of where the research here has given possible other ideas for other two-dimensional cellular automata.

Another application of cellular automata for images is creating almost "living" effects. Image for instance, instead of a logo having a chrome or luminous effect cellular automata could produce a smooth curved surface made up of tiny moving "mosaics" of light.

With the immense extension of the computation power of a non-uniform cellular automata compared to a uniform cellular automata, it means that cellular automata could solve very computationally intensive problems. One such example is in cryptology (uniform Cellular Automata have been used for encryption, with reversible rules for the private key and irreversible rules for the public key). From the research done here it is conceivable that a cellular automata could be used to find patterns very well hidden in supposed chaos (randomness) – like for instance RSA encryption. With the left most row of cells as a input in a 2-dimensional cellular automata and the right most row of cells as the output; encrypted data could be feed into the cellular automata and with the correct set of non-uniform rules it may be possible to produce the un-encrypted data. As RSA encryption uses a public and private key system, if this principle held to be true, you have an endless source of ciphertext and plaintext to "train" the cellular automata rules to and you could produce a cellular automata configuration with the same effect as the private key. One might quickly think that this would be intractable task within a finite amount of time, but when you consider there are already parallel machines in existence with 16,384 processors, with a cell per processor this could produce 2^{524288} different possible configurations (using 32-bir rules). The ideal mechanism to search this rule space would be genetic algorithms as implemented in this project. As far as the research carried out in this project has found there is very little research done in using non-uniform cellular automata for decryption. This would possibly be a PhD proposal in itself.

10. Conclusions

This project has shown without any doubt that cellular automata can be used for image processing, as edge detection is one of the more complicated image processing tasks – so presumably cellular automata could be used for easier image processing techniques with very little effort.

The fact that cellular automata have outperformed the original hypotheses, which was to approximate the edge detection of existing algorithms, and actually show a more unique method of edge detection show that cellular automata have enormous potential in the image processing domain.

This report has shown that one of the major problems that previous researchers have had using cellular automata for image processing, such as the well researched thinning task was due to a conventional approach being tried – this was using the 5 cell: up, down, left, right configuration. When all that was needed was the diagonal cell – which in previous researchers analysis was ignored because they presumed it would result in a rule that is intractable to implement in a two-dimensional cellular automata, a 512-bit rule, where as my research has produced the same effect with only a 16-bit rule.

11. Acknowledgements

I would like to thank the following people for their help and support through this project: Mark Roberts, Gavin Brown, Alex Pim, Trent Rosenbaum, Nilesh Patel and of course Riccardo Poli for being my supervisor and allowing me to investigate this interesting research area.

12. References

[Bell 1997] Bell, D 1997. *Make Java fast: Optimize!* Web-page: http://www.javaworld.com/javaworld/jw-04-1997/jw-04-optimize_p.html

[Chady, Poli, 1997] Chady, M. Poli, R. 1997. *Evolution of Cellular Automaton-based Associative Memories.* School of Computer Science, University of Birmingham, UK.

[Codd, 1968] Codd, E. F. 1968. *Cellular Automata.* Academic Press, New York.

[Conway, 1976] Conway, John. 1976. *On Numbers and Games.* New York: Academic Press, Inc

[Crutchfield, Mitchell 1995] Crutchfield, J. P. and Mitchell, M. 1995. *The evolution of emergent computation.* Proceedings of the National Academy of Sciences USA, vol. 92, no. 23, pp 10742-10746.

[Darwin, 1866] Darwin, C. R. 1866. *The Origin of Species.* Penguin, London. 1st edition reprinted, 1968.

[Guo, Hall 1989] Guo, Z and Hall, R. W. 1989. *Paralell thinning with two-sub-iteration algorithms.* Communications of the ACM, vol. 32, no.3, pp 581-598.

[Hardwick 1997] Hardwick, J. 1997. *Optimizing Java for Speed.* Web-page: http://www.cs.cmu.edu/~jch/java/speed.html

[Holland, 1975] Holland, J. H. *Adaptation in Natural and Artificial Systems.* The University of Michigan Press, Ann Arbor, Michigan.

[Kozen, 1997] Kozen, D. C. 1997. *Automata and Computability.* Springer-Verlag, New York Inc.

[Land, 1994] Land, M and Belew, R.K. 1994. *No perfect two-state cellular automata for density classification exists.* University of California San Diego, La Jolla.

[Minsky, 1967] Minsky, M. L. 1967. *Computation: Finite and Infinite Machine.* Prentice-Hall, Englewood Cliffs, New Jersey.

[Mitchell et al, 1996] M Mitchell, J.P. Crutchfield, R. Das 1996. *Evolving Cellular Automata with Genetic Algorithms: A Review of Recent Work.* Santa Fe Institute, Santa Fe, New Mexico & IBM Watson Research Ctr, New York.

[Mitchell *et al.,* 1993] M. Mitchell, P. T. Hraber, and J. P. Crutchfield 1993. *Revisiting the edge of chaos: Evolving cellular automata to perform computations.* Santa Fe Institute, Santa Fe, New Mexico & Physics Dept., University of California, Berkeley, cellular automata

[Orovas, Austin, 1997a] Orovas, C. Austin, J 1997. *Cellular Associative Neural Networks for Image Interpretation.* Computer Science Department, University of York, UK.

[Orovas, Austin, 1997b] Orovas, C. Austin, J 1997. *A Cellular System for Pattern Recognition using Associative Neural Networks.* Computer Science Department, University of York, UK.

[Orovas, Austin, 1997c] Orovas, C. Austin, J 1997 *Cellular Associative Symbolic Processing for Pattern Recognition.* Computer Science Department, University of York, UK.

[Poli, 1996] Poli, R. 1996. *Genetic Programming for Image Analysis.* School of Computer Science, University of Birmingham, UK.

[Poli, 1999] Poli, R 1999. *Evolutionary Computation.* Lecture course, University of Birmingham, Birmingham, UK.

[Preston, Duff 1984] Preston, Jr., K, and Duff, M. J. B. 1984. *Modern Cellular Automata: Theory and Applications.* Plenum Press, New York.

[Sipper, 1995] Sipper, M. 1995. *Quasi-uniform computation-universal cellular automata.* Third European Conference on Artificial Life, vol. 929 of Lecture Notes in Computer Science, pp 544-554. Springer-Verlag, Heidelberg.

[Sipper, 1996] Sipper, M. 1996. *A brief introduction to genetic algorithms.* Web page, Swiss Federal Institute of Technology, Switzerland.

[Sipper, Ruppin 1996] Sipper, M., Ruppin, E, 1996. *Co-evolving architectures for cellular machines,* Logic Systems Laboratory, Swiss Federal Institute of technology.

[Sipper et al. 1996b] Sipper, M., Tomassini, M. 1996b. *Generating parallel random number generators by cellular programming.* International Journal of Modern Physics C, vol. 7, no. 2, pp 181-190.

[Sipper et al. 1996c] Sipper, M., Tomassini, M., and Capcarrere, M. 1996c. *Evolving asynchronous and scalable non-uniform cellular automata.*

[Toffoli and Margolus, 1987] Toffoli, T. and Marglus, N., 1987. *Cellular Automata Machines.* The MIT Press, Cambridge, Massachusetts.

[Tomassini, 1995] Tomassini, M. 1995. *A survey of genetic algorithms*. Technical Report 95/137, Dept. of Computer Science, Swiss Federal Institute of Technology, Lausanne, Switzerland.

[von Neumann, 1966] von Neumann, J. 1966. *Theory of Self-Reproducing Automata.* University of Illinois Press, Illinois. Edited and completed by A. W. Burks.

[Wolfram 1983] Stephen Wolfram 1983. *A Universal Computation Class of Cellular Automata.* Los Alamos Science.

[Wolfram, 1984] Wolfram, S. 1984. *Universality and complexity in cellular automata.* Physica D*,* vol. 10, pp 1-35.

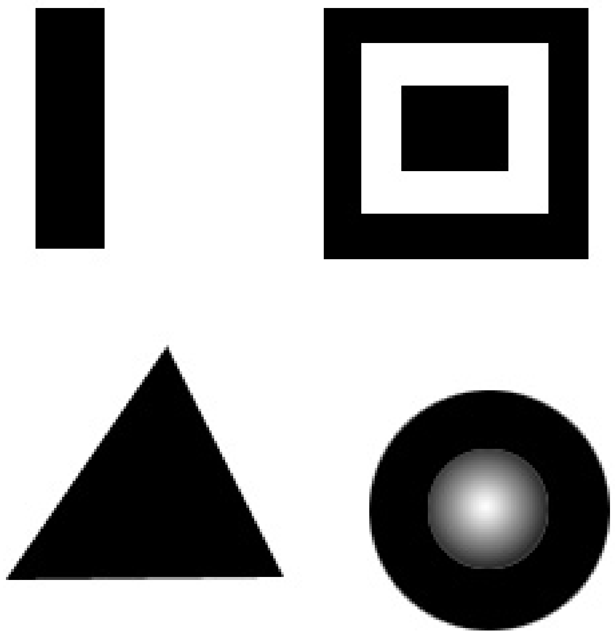
[Yao, 2000] Yao, X. 2000. *Advanced Topics in Evolutionary Computation*. Lecture course, University of Birmingham, Birmingham, UK.

Appendix 1: Table of figures

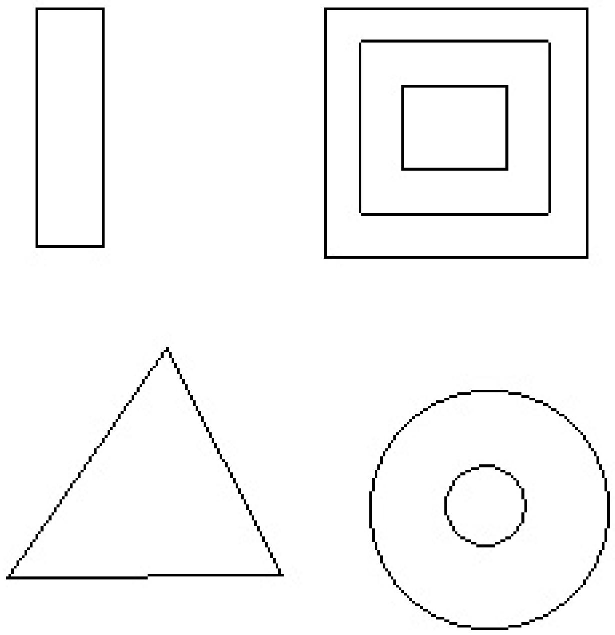
Figure:	Title:	Page:
1	An example of a one dimensional cellular automata	2
2	An example of the neighbourhood in a one dimensional cellular automata	2
3	A rule table for a one dimensional cellular automata	3
4	The size of rules depending upon nieghbourhood size	3
5	Psuedo-code of the cellular programming algorithm	10
6	A cell in the game of life staying alive	13
7	A cell in the game of life staying dying	13
8	A cell in the game of life staying becoming alive	13
9	A graph of fitness against time for a uniform cellular automata	14
10	The result of uniform cellular automata edge detection	14
11(a)	The ideal output from the density task(1)	15
11(b)	The ideal output from the density task(2)	15
12	32-bit cell neighbourhood diagram	16
13	Ideal edge detection result	17
14	Preminary 32-bit non-uniform edge detection result	17
15	Effect of implementing mutation on 32-bit non-uniform edge detection result	18
16	Effect of doubling the mutation rate 32-bit non-uniform edge detection result	18
17	Effect of halving the mutation rate 32-bit non-uniform edge detection result	19
18	Result of generic/dynamic 32-bit non-uniform edge detection	19
19	Graph of fitness for generic/dynamic 32-bit non-uniform edge detection	19
20(a)	The thinning tasks input	20
20(b)	The thinning task's indeal output	20
21	Static result for the thinning task from the cellular automata	21
22	Generic result for the thinning task from the cellular automata	21
23	Generic result for the thinning task with altered fitness function	22
24	Generic result for the thinning task with penalty fitness function	22
25	Generic result for the thinning task with refined penalty fitness function	22
26	A grapgh of fitness versus time for $M = 100$	23
27	Result from attempting to reduce the rule space	24
28	Static result for the thinning task from the diagonal neighbourhood	25
29	Generic result for the thinning task from the diagonal neighbourhood	26
30(a)	Rectangle input for diagonal edge detection, first attempt	26
30(b)	Rectangle result from diagonal edge detection, first attempt	26
31	Diagonal edge detection result using sub-iterative algorithm	27
32(a)	Image input for diagonal edge detection using sub-iterative algorithm	27
32(b)	Image result from diagonal edge detection using sub-iterative algorithm	27
33	The result from a conventional edge detection algorithm	27
34	One pixel to four cell mapping for greyscale images	28
35(a)	Result from greyscale cellular automata edge detection	29
35(b)	Result from greyscale conventional edge detection	29
36(a)	Inputted synthetic image in final edge detection algorithm	30
36(b)	My edge detection method on synthetic image in final edge detection algorithm	30
37(a)	Inputted photo image in final edge detection algorithm	30
37(b)	My edge detection method on photo in final edge detection algorithm	31
38(a)	Inputted text image in final edge detection algorithm	31
38(b)	My edge detection method on text in final edge detection algorithm	31

Appendix 2: Extended Results

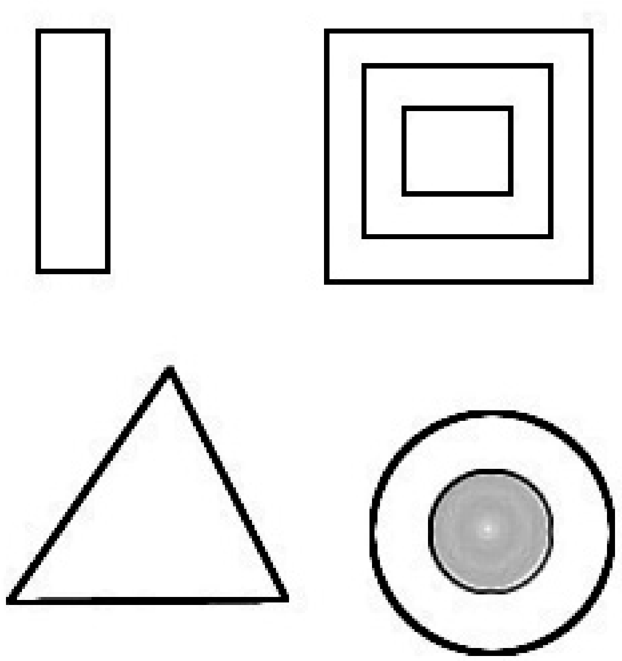
2.4 Extended examples of results for synthetic images



This is the inputted image.

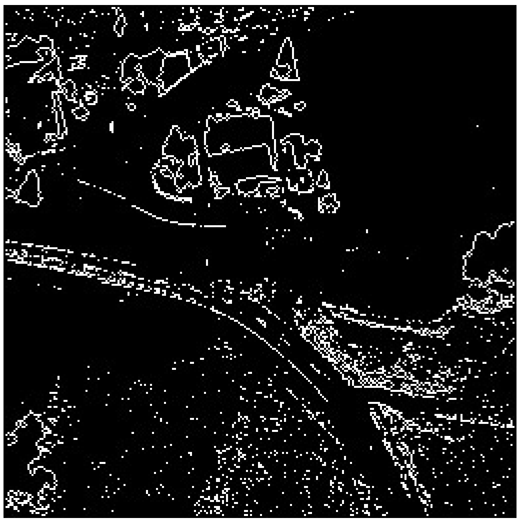


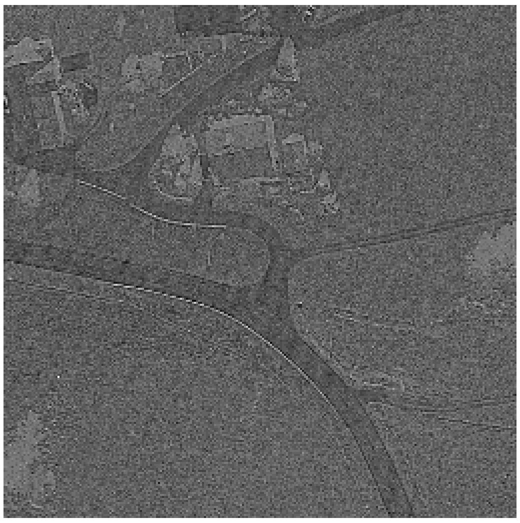
This is the output from my edge detection method.

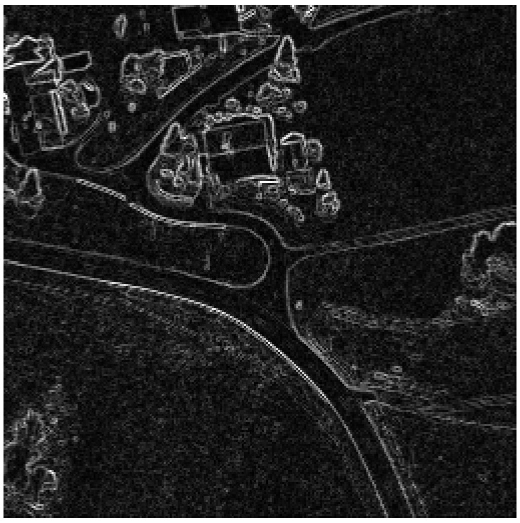

This is the result of an existing algorithm.

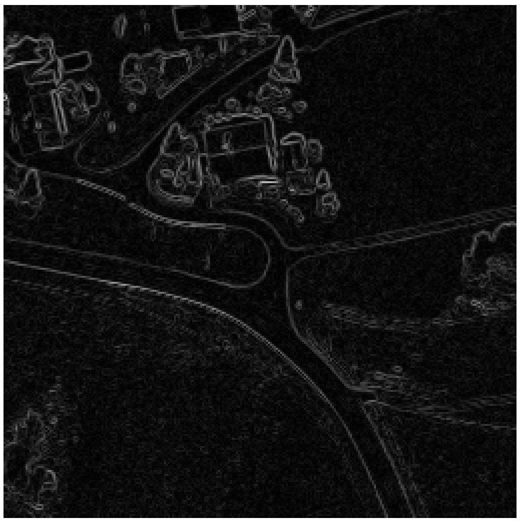
2.5 Extended examples of results for natural images

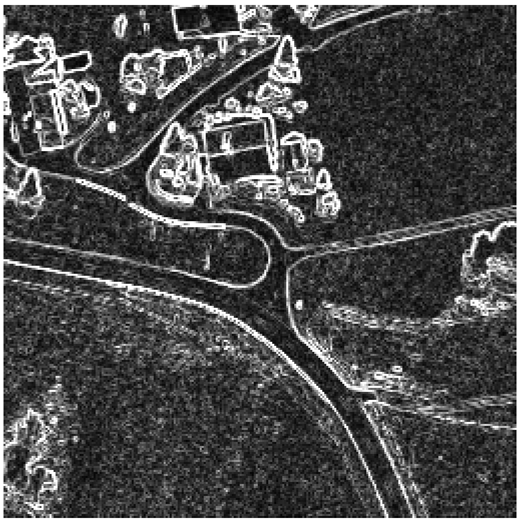
This is an inputted natural image.

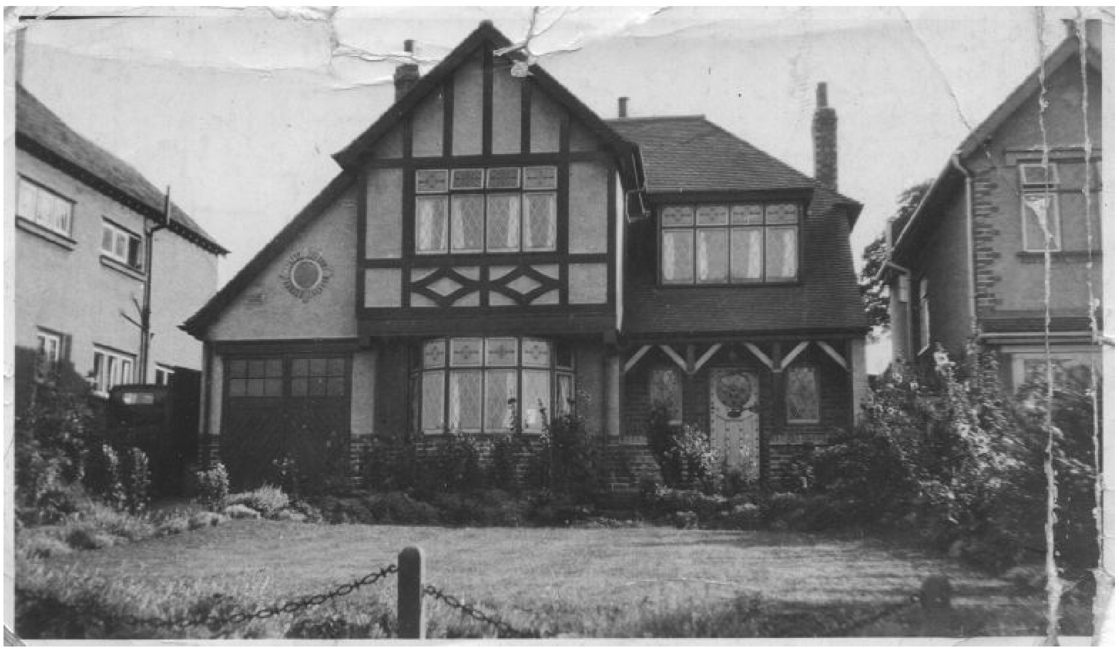

This is the result of my algorithm.


Using LaPlace of Gaussian.

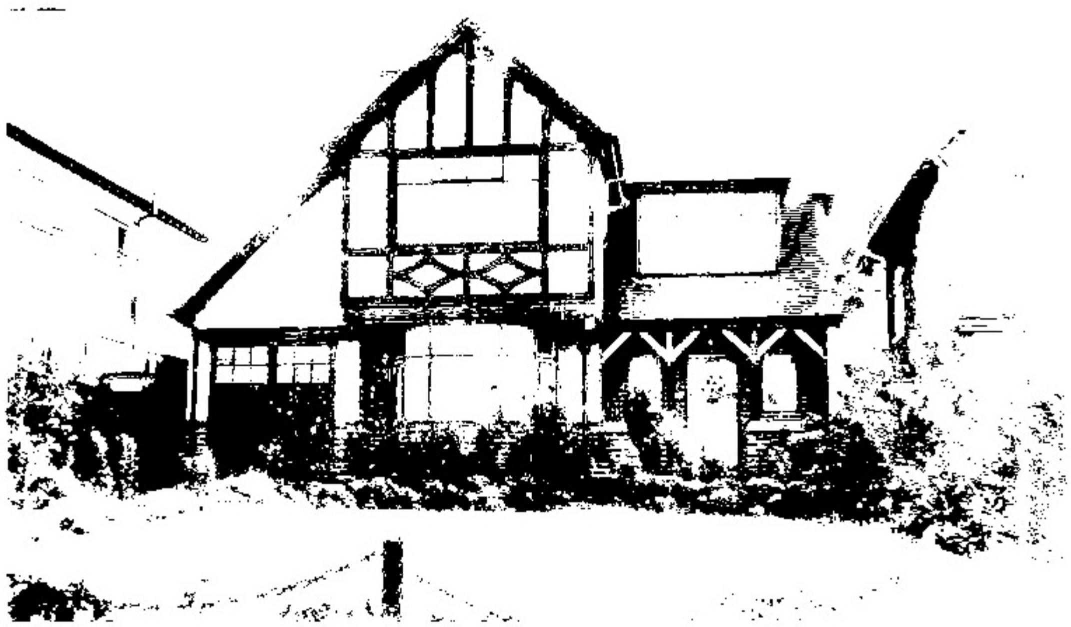

Sobel’s edge detection.

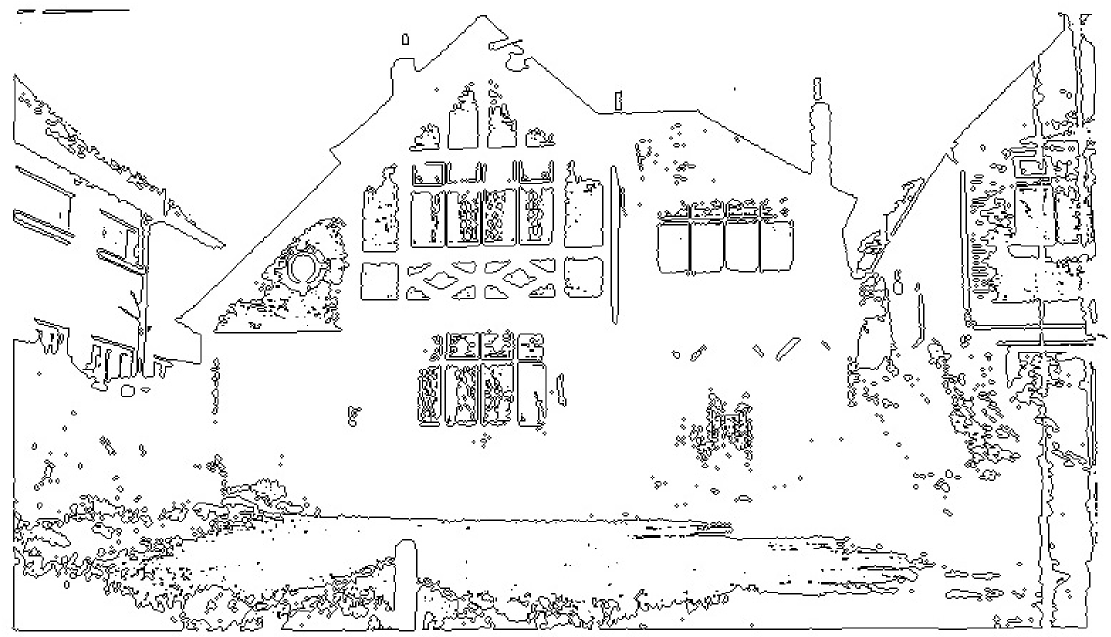

Prewitt’s edge detection.

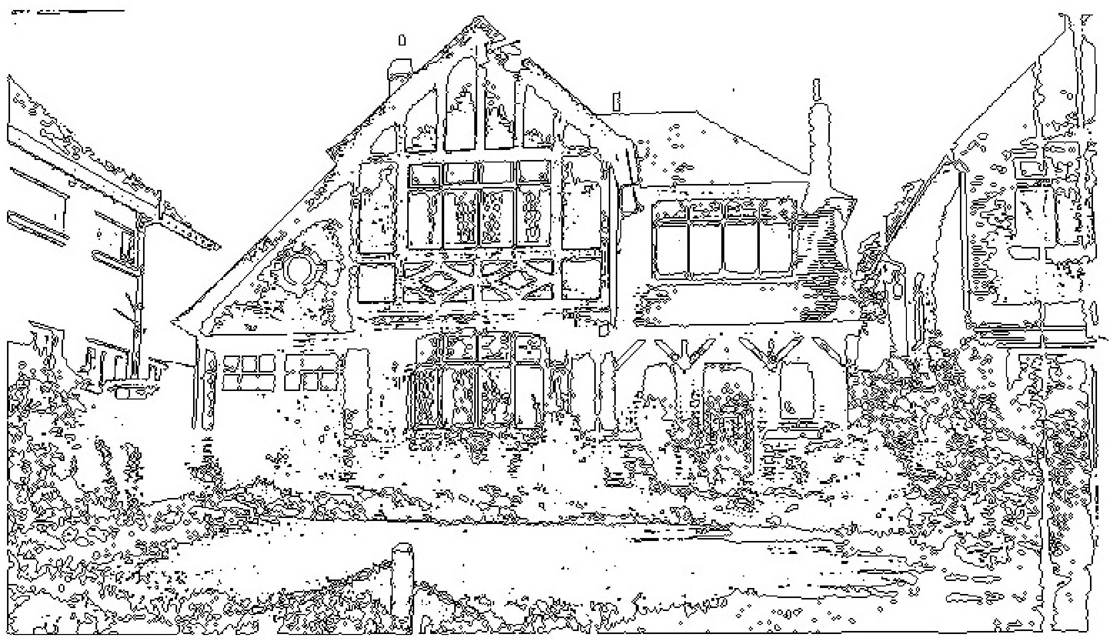

Kirsch’s edge detection.

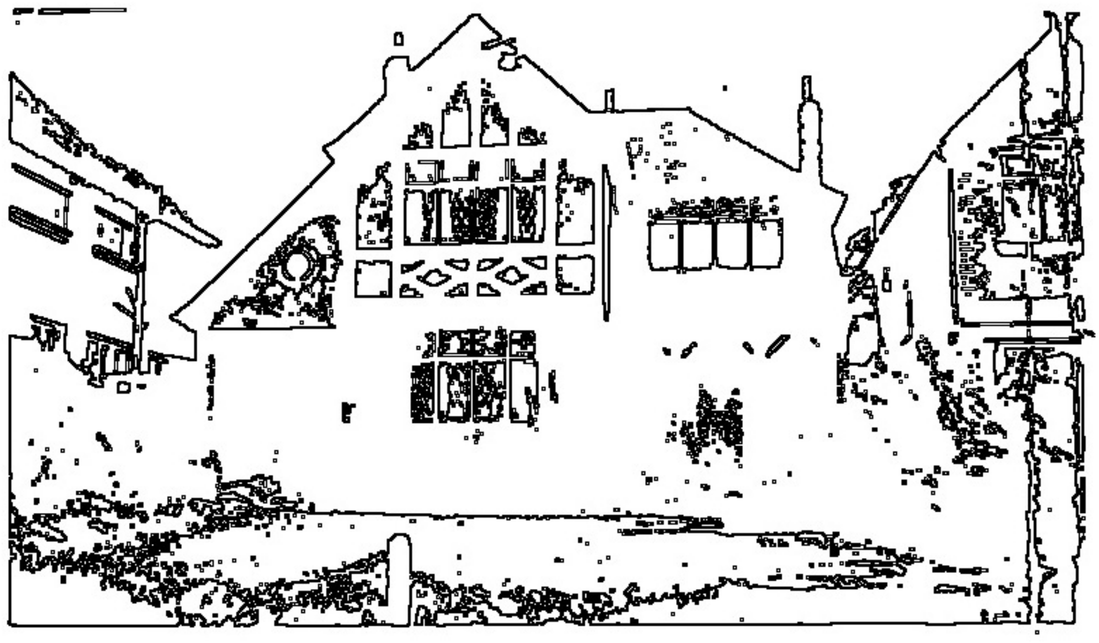

This is another inputted 256 level greyscale natural image.

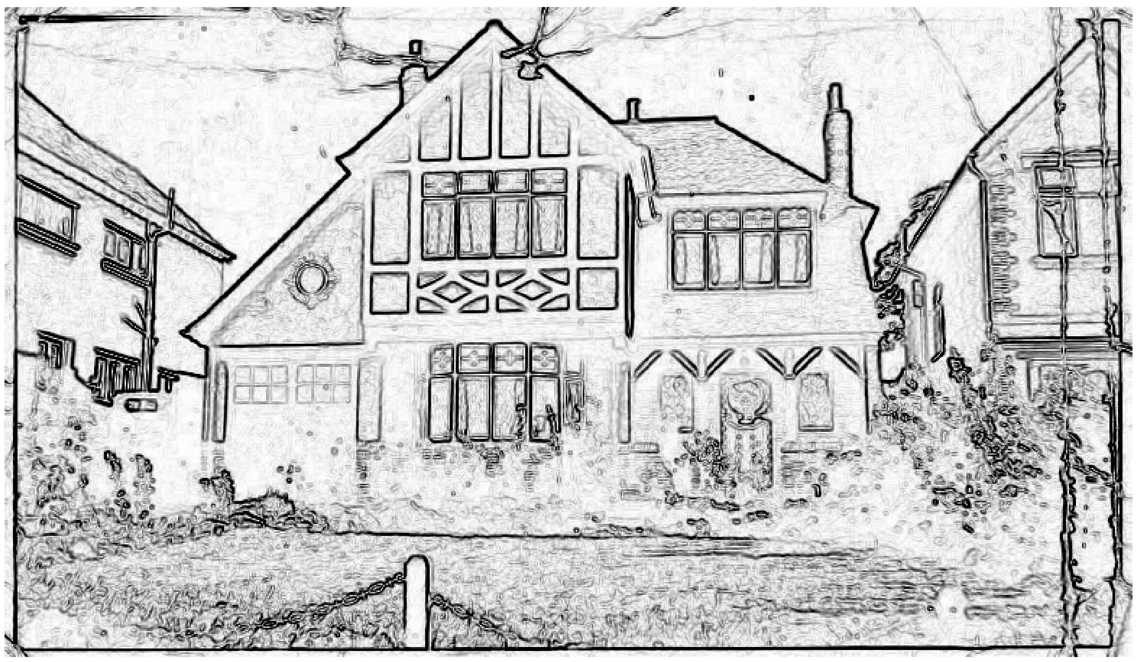
Following are examples of changing the threshold value of greyscale levels of what becomes black and what becomes white.


This is what is actually inputted to the cellular automata with a threshold of 128.


This is the edge detection my method produces with a threshold of 128.

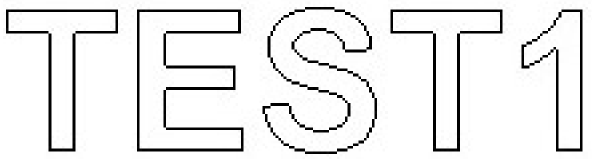

This is the edge detection my method produces with a threshold of 196.


This is the output of a conventional edge detection algorithm on the binary image.

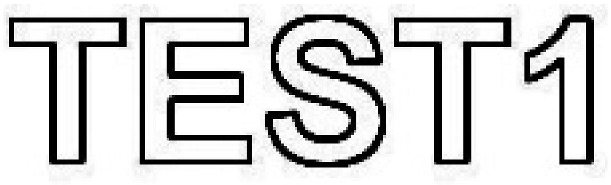

This is the output of a conventional edge detection algorithm on the 256 level greyscale image.

2.3 Extended examples of results for text images

The inputted text image.

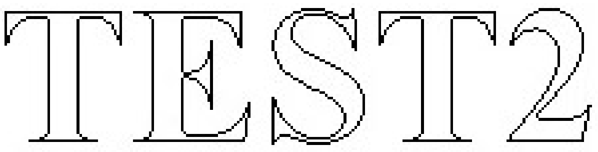


My edge detection result.

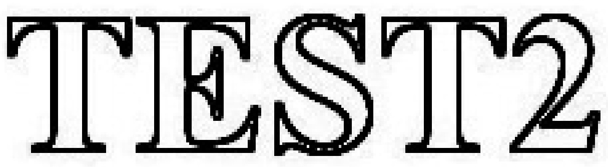


A conventional algorithm.

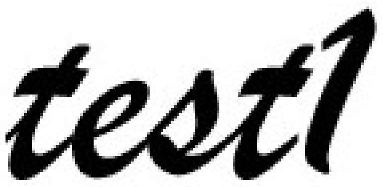
The inputted text image.



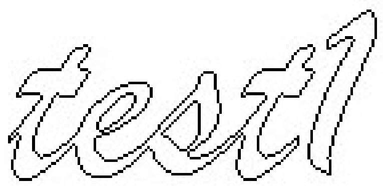
My edge detection result.



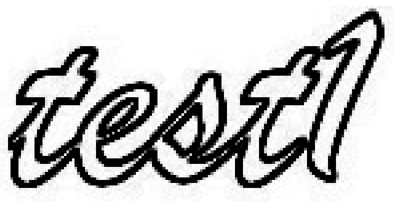
A conventional algorithm.



The inputted text image.

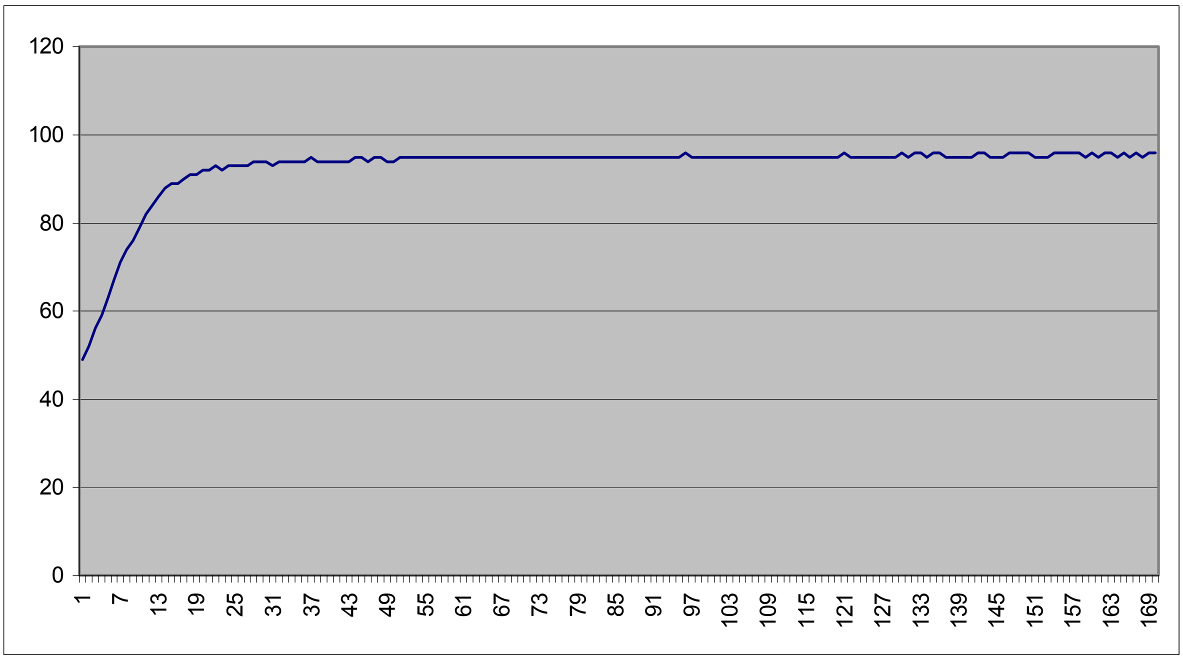


My edge detection result.

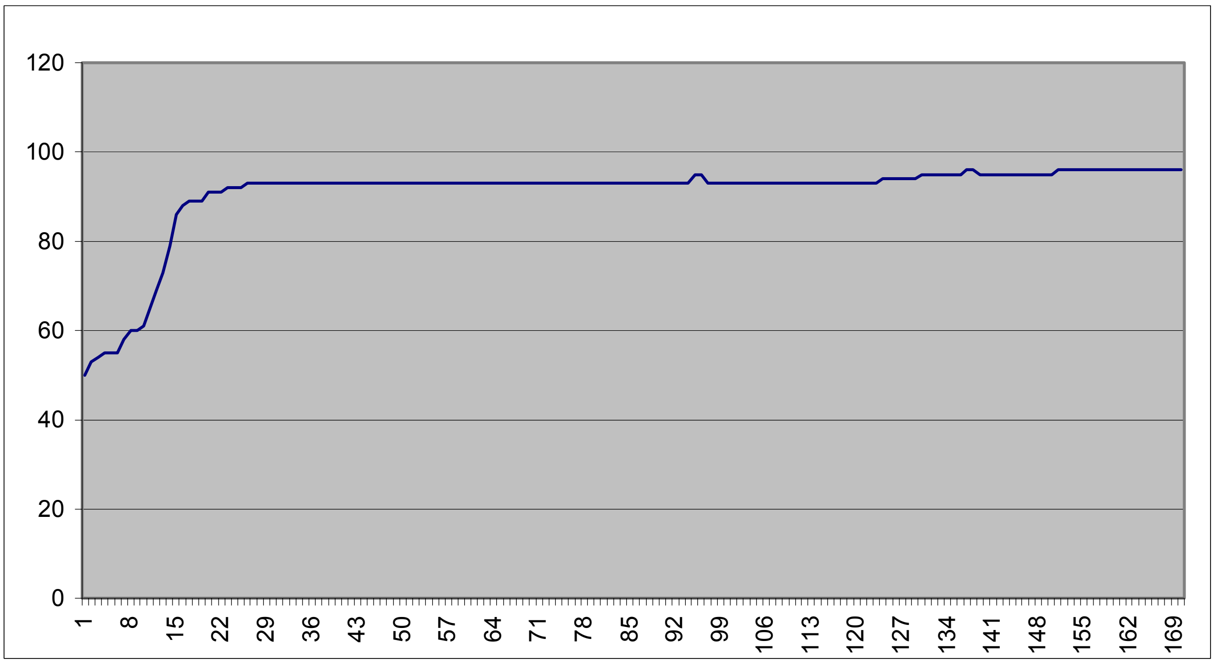


A conventional algorithm.

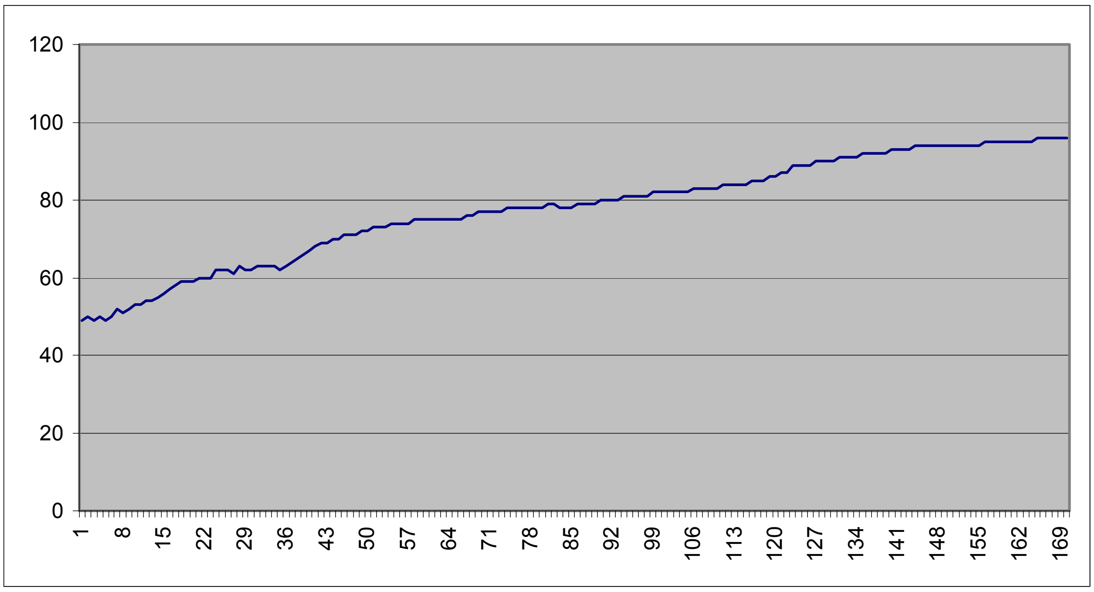
Appendix 3: Various graphs of results



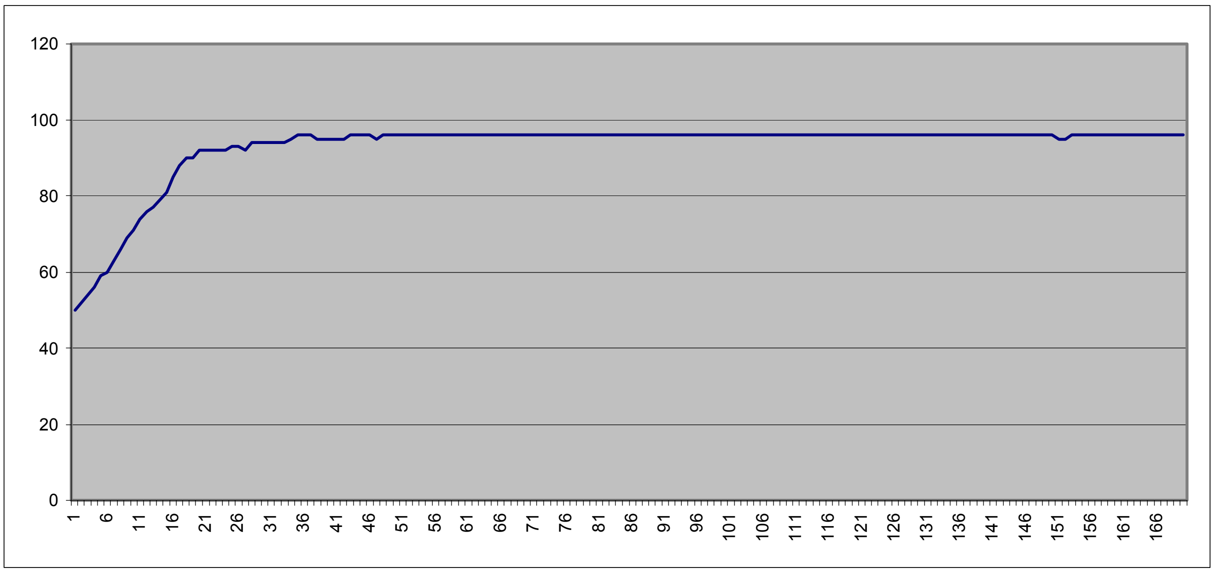
This graph shows fitness against time for $M = 100$ in section 6.6.5



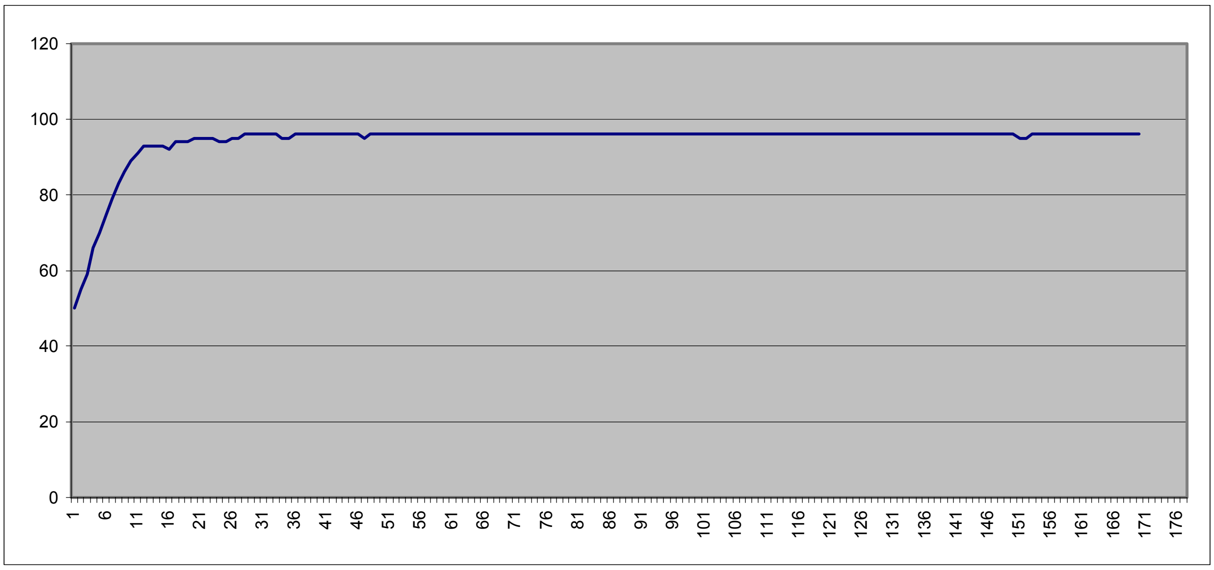
This graph shows fitness against time for $M = 300$ in section 6.6.5



This graph shows fitness against time for $M = 10$ in section 6.6.5



This graph shows fitness against time for the non-optimised algorithm in section 6.12 – the non-optimised sub-iterative algorithm.



This graph shows fitness against time for the optimised algorithm in section 6.12 – the optimised sub-iterative algorithm.

Appendix 4: How to run my program

The code for this project is located in the directory: "~cdt/project/". Note that a lot of this code was experimental and the graphic output can be a little spurious.

The best binary edge detection developed in this project:

~cdt/project/best/

This program takes pictures in RAW format, the images must be symmetrical and in greyscale. In the Picread3.java file, the following variable contains the filename:

```
   String inputt = "filename.raw";
```

The following variable must both have the same value and are the x and y dimensions of the image in pixels:

```
   private final int sizeX = 100;
   private final int sizeY = 100;
```

To run this program to perform edge detection
– type "java_g Picread3"

To run this program to evolve the cellular automata
– type "java_g Picread"

Uniform cellular automata:

~cdt/project/uniform/

To run this program to evolve the uniform cellular automata
– type "java Picread4"

The attempt at greyscale edge detection:

~cdt/project/grey/

To run this program to evolve the cellular automata
– type "java_g Picread"

The density task:

~cdt/project/density/

To run this program to evolve the cellular automata
– type "java_g Picread"

The game of life:

~cdt/project/life/

To run this program to simulate the game of life
– type “java_g Picread”

Appendix 5: Code fragments

5.5 Rule-run code-fragment

This is the part of the program that implements the effect of the rule within each cell to cause the cells state to change in the next time step. After profiling this was the procedure that almost all the processing time was spent in – 97%. That is why it was important this piece of code was efficient.

```
private void ruleRun()
{
   int i,j,k;
   for (i=1; i<4; i++)
      {
         for (j=1; j<4; j++)
         {
            int ir = i + 1;
            int ju = j + 1;
            int bit1 = 0;
            if (stateArray2[i][ju] == 1)
            {bit1 = bit1 + 1;};
            if (stateArray2[ir][ju] == 1)
            {bit1 = bit1 + 2;};
            if (stateArray2[ir][j] == 1)
            {bit1 = bit1 + 4;};
            if (stateArray2[i][j] == 1)
            {bit1 = bit1 + 8;};
            int r=cellArray[i-1][j-1].rule2;
            int p=32768;
            for (k=15-bit1; k>0; k--)
            {
               p = p >> 1;
            }
            if ((p & r) == 0)
            {
               stateArray[i][j] = 0;
            }
            else
            {
               stateArray[i][j] = 1;
            };

         };
      };

      for (i=1; i<4; i++)
      {
         for (j=1; j<4; j++)
         {
            stateArray2[i][j] = stateArray[i][j];
         };
      };

}
```

5.6 Cross-over code fragment

This was a very important piece of code as it was what actually caused the rules to evolve. This utilises bit shift operators to dramatically increase the speed of the codes execution.

```
private int GA(int rule1, int rule2)
{
   // because you cant crossover at position 0 or 16
   int XO = ((int)(Math.random() * 14) +1);
   int mask=0;
   int p=32768;
   int i;

   for(int j=16; j>=XO; j--)
   {
      mask += (rule1 & p);
      p = p >> 1;
   }
   int chunk = rule1 ^ mask;
   mask = 0;
   p = 32768;
   for(int k=16; k>=XO;  k--)
   {
      mask += (rule2 & p);
      p = p >> 1;
   }
   int child1 = (mask | chunk);

   return child1;
}
```

5.7 Mutation code-fragment

This is the code fragment for the mutation operator. It mutates one in every thousand bits passed to it. This utilises bit shift operators to dramatically increase the speed of the codes execution.

```
private int mutate(int rule)
{
   int p=32768;

   for(int j=16; j>0; j--)
   {
      if ((Math.random() * 10000) < 10)
      {
         rule = (rule ^ p);
      }
      p = p >> 1;
   }
   return rule;
}
```

5.4 Cell class code-fragment

The cell class is very basic, containing the current fitness of the cell, and two copies of its rule – to preserve parallelism. The main program uses a 2-dimensional array of these cell objects to form the cellular automata.

```
public class Cell
{
   int fitness = 0;
   int rule = 0;
   int rule2 = 0;

   public Cell()
   {
      rule = ((int)(Math.random() * 65535));
      rule2 = rule;
   }
}
```

Appendix 6: Specification

6.1 Original specification

SUPERVISOR

My supervisor for this project is Riccardo Poli.

1. INTRODUCTION

This project aims to find out whether it is feasible and useful to use Cellular Automata to carry out the pre-defined task of finding, recording and then displaying the edges in an image. Cellular Automata will be evolved to generate an algorithm that when any image, supplied to it in the correct format, will be able to locate the edges and show the user the edges in that image. With this project the final piece of software is not actually that important – but the understanding of what Cellular Automata can do is important and also why they can.

2. OBJECTIVES

In this project I hope to construct the building blocks for further research in this area.

As this project is using what I regard as “state of the art” concepts in Artificial Intelligence, namely Cellular Automata and Evolutionary Computation, the project will have a large research element to it. As you can see from my work-plan I have allocated a large proportion of my time to research. This is why the specification is a little vague in some areas – I will have to have done the research before I can fully understand some of the concepts of the project. It may come to the point that the specification (and the work-plan) may have to be re-written as I better understand the task that makes up my project.

Edge detection has many uses especially in graphical packages when manipulating images – there is a large amount of edge detection software either imbedded in graphics packages or available as “plug-ins” for them. Although this project when finished will most likely not be a commercially viable product “as is” but could well in the future be extended into a product that can be used in graphical packages and suchlike.

In doing this project I will prove whether Cellular Automata are a feasible means of carrying out edge detection in images. It may turn out that at the end of the project that I have concluded that Cellular Automata are not appropriate for this form of image processing – in this scenario the project will not have failed in its objectives as I will have proved categorically that Cellular Automata should not be used for this purpose. This still means that the project has had purpose as it could well save other researchers a lot of time in the future.

By “feasible means” I mean that the resultant evolved Cellular Automata will be able to display the edges of an image in a fast and efficient manner – there would be little point in having developed something that is worse than whatever is now available and that has no potential to be improved. This I would not class as “feasible”.

I plan to place my findings on my web site with interactive facilities.

3. IMPLEMENTATION

To describe the implementation of this project, I am going to break it down into several key sections:

- Research into simulator
- Research into Cellular Automata
- Coding the Cellular Automata
- Simulator for the Cellular Automata
- Coding the breakdown and reconstruction of images to be processed
- State transition functions
- Testing/Evaluating the resulting Cellular Automata from the state transition functions

I will use binary images for the edge detection – images consisting of just black and white pixels (1 ands 0’s).

One of the extensions that I will aim to achieve is edge detection on greyscale images consisting of up to 256 shades. This would allow a reasonable quality photo to be processed. I would like to be able to extend the detection to include colour images – but this will be much more complicated as the colours are no longer on a linear scale from white to black. Colour scales have multiple dimensions and values that how close they are to each other depends on which colour model is being used. Obviously it is not impossible (as several good edge detection programs exist) but probably beyond the scope of this project – it is something that could be done in the future.

As I see it the finished project, if Cellular Automata can be used effectively here, will take in an image then split it up into sections of X by X pixels (where X is a value I will determine for optimal performance). Then each of those sections will have the algorithm (resultant from the evolution of Cellular Automata) used on them and this will tell the program where the edge is. This is repeated until the whole image has been analyzed. The edges for each section of the image are then reconstructed and then displayed to the user.

The program should hopefully be able to process any size (in pixels) of image – as the image is broken down to fixed sized blocks by the program. Problems may occur when working with images that do not completely divide up into the correct block sizes – I am confident that this can be overcome quite easily

I aim to be able to display the edges to the user along with the original image for comparison. What I would like to do is to display the image with the edges overlaid on top of it (probably enlarged slightly so that the edges can be discerned from the original image) as a transparent GIF image.

One very important part of the project is the simulator. The simulator is where the evolution of the Cellular Automata will take place. I have decided to write my own simulator for the cellular automata. The reason for this is that I believe that this will allow me to be able to customise the simulator for to work more effectively for my project.

As I plan to place my findings on my web site to make it generally available – to do this I will use Java Applets imbedded in Cold Fusion pages. This makes it much easier to take feedback and provide information and restriction to the pages if necessary.

4. KEY STAGES

This section describes what events in the project are counted as major breakthroughs in the project. They are summarized as a list below:

- Simulator Constructed
- First Cellular Automata evolving
- Splitting image and reforming newly generated images
- Algorithm formed from Cellular Automata used to process an image
- Using the Algorithm on the binary split images and reforming them into a single image with just the edges.
- Evolving Cellular Automata to generate an "effective" algorithm
- A graphical interface for the images to be displayed and to allow the user to process them.
- Extending the processing to include greyscale images – 32 shades then 64 then 128 and finally 256.

5. FALL BACK POSITIONS

Listed here are things that if the key stages for the project do not work out as planned these are alternatives:

Simulator – If I do not manage to construct and design my own simulator then there are several available simulators available for more general use.

Greyscale Images – If I can't get the Evolved Cellular Automata to successfully process the edges of 256 shade greyscale images, I will settle for 128, or 64, or 32. Failing this just binary images will be used – if this is feasible.

Splitting Images – If when splitting images to smaller blocks with all sizes of image I will use images of a fixed multiple of my block size. If still there are problems with this then I will use fixed sizes of images as a last resort.

6. OTHER WORK ALREADY DONE IN THE FIELD

This describes work carried out by other people in areas similar what I am trying to do. I will research this data more thoroughly during the early stages of my project for possible pitfalls and for inspiration.

General:
Cellular Neural Networks have been used in the past to do several kinds of image processing.

Papers:

- Genetic Programming for Image Analysis – *Riccardo Poli*

This centres on the idea that image processing can be seem as image processing problems.

- Cellular Associative Neural Networks for Image Interpretation – *C Orovas & J Austin*

This presents the idea to use associative neural networks for image and symbolic processing. This makes use of Cellular Automata theory quite heavily.

- Cellular Associative Symbolic Processing for Pattern Recognition - *C Orovas & J Austin*

- A Cellular System for Pattern Recognition using Associate Neural Networks - *C Orovas & J Austin*

7. HARDWARE / SOFTWARE TO BE USED

I plan to design and code my project using the Microsoft Windows platform. The machines that I will be using are:

Home PC
Intel Celeron 450 Processor
128Mb 100Mhz dual inline memory
19Gb hard disk drive
Windows 98
Possibly Redhat Linux 6

University Machine
Windows NT Machine – G13

Software:
Sun Java Development Kit
Microsoft Visual J++
Allaire Cold Fusion (for demonstration web pages)

All of my coding I plan to do in Java unless I find it absolutely necessary to use other languages. I may use C or C++ for some parts of the software – especially if I have to use any existing code in my own code as most of the work done so far in this field has been done in C or C++. The reason I would like to do most of my work in Java (or possibly C / C++) is that it can then easily be ported between different operating systems/platforms. This is very important as the project is meant to be providing a basis for other people in the future.

6.2 Original work-plan

Day	**Month**	**Work to be Done**
		Specification due
4	October	Research into Cellular Automata in general
11	October	Research into current image processing Cellular Automata
18	October	Research into CA simulator & start designing simulator
25	October	Code Simulator
1	November	Code Simulator
8	November	Design & Code Image Splitter
15	November	Design & Code Image Reconstructor
22	November	Research into Cellular Automata
29	November	Code Cellular Automata
6	December	Code Cellular Automata
13	December	Code Cellular Automata
		First Inspection
20	December	Graphical Output
27	December	Start to Plan Dissertation
3	January	Start to Plan Dissertation
10	January	State Transistion functions - Research / Code
17	January	State Transistion functions - Research / Code
24	January	Evaluate Functions
31	January	Evaluate Functions
7	February	Evaluate Functions
14	February	Greyscale Image Extension
21	February	Greyscale Image Extension
28	February	Greyscale Image Extension
6	March	GUI
13	March	Web Page Interface
20	March	Final debugging and testing
27	March	Dissertation writing
		Final Inspection
3	April	Dissertation writing
10	April	Dissertation writing
17	April	***Dissertation Deadline***

Appendix 7: A revised work-plan

This is a revised version of my work-plan. A large proportion of the sections that state "code" also involved a lot of research to get that code to operate to perform the task it was intended to do.

<u>Day</u>	<u>Month</u>	<u>Work to be Done</u>
		Specification due
4	October	Research into Cellular Automata in general
11	October	Research into uniform Cellular Automata
18	October	Research into non-uniform Cellular Automata
25	October	Research into non-uniform Cellular Automata
1	November	Code Simulator
8	November	Code Cellular Automata
15	November	Research into current image processing Cellular Automata
22	November	Research into current image processing in general
29	November	State Transistion functions - Research / Code
6	December	Code Cellular Automata game of life
13	December	Research & code Cellular Automata density task
		First Inspection
20	December	Research Cellular Automata thinning
27	December	Code Cellular Automata edge detection tests - uniform
3	January	More Cellular Automata edge detection research
10	January	Code Cellular Automata edge detection primliminary tests
17	January	Code Cellular Automata edge detection tests
24	January	Research Cellular Automata thinning
31	January	Code Cellular Automata thinning
7	February	Code Cellular Automata thinning
14	February	Code Cellular Automata diagonal thinning
21	February	Code Cellular Automata edge detection diagonal
28	February	Refine Cellular Automata edge detection diagonal
6	March	Greyscale Image Extension
13	March	Greyscale Image Extension
20	March	Dissertation writing
27	March	Dissertation writing
		Final Inspection
3	April	Dissertation writing
10	April	Dissertation writing
17	April	***Dissertation Deadline***

Appendix 8: The outline of my edge detection algorithm

The following is a brief procedure of how my sub-iterative edge detection algorithm operates.

- Read image into array.
- Use the 9 cell (3 by 3) cellular automata on each 5 x 5 chunk of the image, with slight over-lapping.
- As each cells state is calculated in the next time step, the three neighbouring cells: left, below and left below are used with the 16-bit rule.
- The image chunk is now rotated 180 degrees, and the last operation is repeated.
- The two results are then compared using an "or" operator and the result stored in a temporary array.
- Then the next chunk of the image is processed.