UNIVERSITY OF OSLO
Department of Informatics

Lossless Data Compression Using Cellular Automata

Martin Haukeli

Network and System Administration
Oslo University College

May 23, 2012
Lossless Data Compression Using Cellular Automata

Martin Haukeli

Network and System Administration
Oslo University College

May 23, 2012
Abstract

This thesis presents an investigation into the idea of using Cellular Automata to compress digital data. The approach is based on the fact that many CA configurations have a previous configuration, but only one next generation. By going backwards and finding a smaller configuration we can store that configuration and how many steps to go forward, instead of the original.

In order to accomplish this an algorithm was developed that can backtrace a 2 dimensional CA configuration, listing its previous configurations. The algorithm is used to find a rule that often has previous configurations and also find an example where a matrix appears to become smaller by backtracing. The conclusion, however, is that when increasing the configuration size to trace the algorithm rapidly becomes too time consuming.
# Contents

1 Introduction

1.1 motivation ........................................ 4
1.2 objectives and methodology (high level overview) .......... 5

2 Background

2.1 Data Compression .................................... 6
2.2 Cellular Automata ..................................... 8
  2.2.1 Moore neighborhood ............................... 10
  2.2.2 Speed of Light .................................. 10
  2.2.3 The Game of Life ................................. 10
  2.2.4 Evolving Cellular Automata ........................ 12
2.3 The basic idea of compression using CA ....................... 12
2.4 Cellular Automata Transforms ................................ 14
2.5 Reversible Cellular Automata ................................ 14
2.6 Java .................................................. 14
2.7 Genetic Algorithm ...................................... 15
2.8 Previous algorithms for Reversing Cellular Automata ............ 15

3 Design

3.1 Reversible Cellular Automata .............................. 17
  3.1.1 Block Cellular Automata ............................. 17
  3.1.2 Second-Order Cellular Automata ....................... 18
3.2 Genetic Algorithm ...................................... 18
  3.2.1 Crossover ........................................ 18
  3.2.2 Mutation .......................................... 19
  3.2.3 Tournament Selection ............................... 19
3.3 Colouring Algorithm .................................... 21
  3.3.1 Color Map ....................................... 23
  3.3.2 Perm Map ......................................... 28

4 Results and Analysis

4.1 Reversible Cellular Automata .............................. 31
4.2 Genetic Algorithm ...................................... 34
  4.2.1 Measures to increase exploration ..................... 37
4.3 Colouring Algorithm .................................... 37
  4.3.1 Measure to decrease complexity on large matrices ....... 42
4.4 CA rules ............................................ 44
5 Discussion and Future Work
  5.1 Discussion ............................................. 48
      5.1.1 Reversible Cellular Automata ................. 48
      5.1.2 Genetic Algorithm ............................ 48
      5.1.3 Coloring Algorithm ............................ 49
  5.2 Future Work on Cellular Automaton based Data Compression 50

6 Conclusion .............................................. 51

7 Appendix ............................................... 54
  7.1 CA rules tested on 3x3 matrices, 1000 best, rules with B8 excluded 54
  7.2 ColourMap ............................................. 65
  7.3 PermMap ............................................... 75
  7.4 Map .................................................... 83
  7.5 ThreadedColourBacktracer ............................ 92
  7.6 Reversible Cellular Automata ........................ 94
      7.6.1 Second-Order CA .............................. 94
      7.6.2 Block Cellular Automata ....................... 94
      7.6.3 Tron Local Rule ............................... 95
      7.6.4 Critters Local Rule ........................... 95
  7.7 GACABacktracer ...................................... 96
      7.7.1 Tournament Selection ......................... 96
      7.7.2 Random Crossover ............................. 96
      7.7.3 Edge Flip Mutation ............................ 97

List of Figures

2.1 A huffman tree ........................................... 7
2.2 2d Cellular Automata (Game of Life, B3/S23) ................ 9
2.3 Example binary CA rules in the Moore neighborhood ........ 10
2.4 Moore neighbourhood .................................... 11
2.5 Evolution in Game of Life ................................ 13
2.6 Example Preimage Network ............................... 16
3.1 XOR crossover ............................................ 19
3.2 Random crossover with 2 parents producing 1 offspring .... 20
3.3 Edge Flip Mutate ........................................ 20
3.4 Coloring a 4x4 matrix ................................... 21
3.5 Symbols devised for backtracking ........................ 22
3.6 How counting the cells was done .......................... 22
3.7 Another configuration and corresponding colormap .......... 22
3.8 One permutation of a colormap ready to be tested .......... 23
LIST OF FIGURES

3.9 Solving Color Maps ................................................. 24
3.10 Color map for a 3x3 matrix with 4 cells .......................... 27
3.11 Venn diagram showing overlap between 2 subtractions ........ 27
3.12 Perm Map: Broken and solved ..................................... 28
3.13 Perm Map: Only one way to solve .................................. 29
3.14 Perm Map: Recursion needed ...................................... 29
3.15 Perm Map: Smallest Choice ........................................ 30

4.1 Reverse CA 1 ............................................................ 31
4.2 Reverse CA 2 ............................................................ 32
4.3 Reverse CA 3 ............................................................ 33
4.4 Colouring Algorithm: 2x2 matrix ................................. 38
4.5 Colouring Algorithm: Average times .............................. 38
4.6 Colouring Algorithm: 3x3 matrix ................................. 38
4.7 Colouring Algorithm: 4x4 matrix ................................. 39
4.8 Colouring Algorithm: 5x5 matrix ................................ 40
4.9 Colouring Algorithm: 6x6 matrix ................................. 40
4.10 Colouring Algorithm: Size vs Time Graph 1 ..................... 41
4.11 Colouring Algorithm: Size vs Time Graph 2 ..................... 41
4.12 Colouring Algorithm: Comparison with Brute Force .......... 41
4.13 Colouring Algorithm: Full backtrace of 4x4 matrix ............. 43
4.14 Colouring Algorithm: Histogram of live cell counts .......... 44
4.15 Colouring Algorithm: 6x6 matrix ................................. 45
4.16 Colouring Algorithm: Matrix split into four ..................... 46
4.17 Colouring Algorithm: 3x3 Matrix Corner ......................... 46
4.18 Colouring Algorithm: Solutions found by splitting matrix ...... 46
4.19 The 10 best rules selected by the test ............................ 47
Chapter 1

Introduction

What if you could compress a 3GB movie down to 50MB? How much faster could it be downloaded? Can a compressed file be compressed further and further?

Today’s Internet has grown very big and this is not only the number of nodes included, but also the data volume. In order to cope with the increasing amount of data everywhere in computer systems Data Compression is central. Data Compression helps users and system administrators lower the cost of storage and transfer by decreasing the volume required to represent the data.

A plentitude of algorithms have been developed to compress digital data. Most of which are specialized to a single type of data, such as text, image and sound. Note that several other categories exist for which special compression algorithms exist. Depending on the data type it may be allowed for some loss of the quality of the data. This allows the compression algorithm to make approximations to the original data(lossy), so that the Compression Ratio can be increased at the expense of the quality. Other algorithms however can be applied to any data type, these algorithms are named General Data Compression. In such algorithms it is generally not acceptable with loss of precision (lossless).

Cellular Automata, a mathematical model of interacting cells in which time and states are discrete, has been proven to be able to simulate many different fields. For example in seismic simulation Cellular Automata has been used to predict the effect of earthquakes, achieving comparable results to previous algorithms in the field.[1]. Cellular Automata may be evolved with different rules and in different number of dimensions, allowing for a vast number of different Cellular Automata to be created. Cellular automata can be simulated in both bounded and infinite space. Cellular Automata can however only be simulated in one direction of time: Forward.

1.1 motivation

Creating and maintaining backups is an important System Administrator job. The backups may be big and it may be required to keep them for an indefi-
nite amount of time. Because of this it is essential to compress backup data. This however could be done off-line, as nobody will usually want to see the backups unless some rare event has occurred.

Current compression algorithms can only compress general data somewhat, and cannot compress data further. What if it was possible to spend more time compressing data to achieve greater compression? This would not only affect Backups, files that are to be distributed to a large amount of peers could also gain benefit from more compression. It is therefore important to investigate new methods and ideas for compression.

Cellular Automata appears to be able to affect matrices in what seems unpredictable and chaotic ways. Yet, by altering the rules the Cellular Automaton can be tailored to change the data in varying ways. An interesting category of rules named Fredkin’s Replicators contain rules that will always result in the original pattern being copied an infinite number of times (Replicated).[2]

problem statement

This thesis will try to create a Cellular Automaton based compression algorithm for lossless general file compression with the hopes of being able to trade CPU time for increased compression ratios.

1.2 objectives and methodology (high level overview)

- Find a method to develop CA algorithms that can compress data.
- Review previous CA models to see if any of them can easily be used for lossless data compression.
- Try to develop a proof-of-concept CA compression algorithm.
Chapter 2

Background

2.1 Data Compression

General Data compression is a way of encoding data so as to eliminate redundancy and achieve smaller data sizes while still retaining the original data or the original meaning of the data. Redundancy can be seen as the amount of entropy in the data, more entropy equals less redundancy.

Lossless data compression, as opposed to lossy, guarantees that the original data can be restored from the compressed data. In lossy compression it may be that only an approximation of the original data can be restored from the compressed data. When storing data such as images, video or audio the use of lossy compression makes sense as the original meaning could be retained even if the actual bits are not the original. For other data types such as applications, lossy compression cannot be applied as it would change how the program operates.

In order to do this, we can search for repetitive parts of data, and replace them with short-hand representations. Random data cannot be compressed, however. [3]

The huffman coding works by assigning each symbol a new representation form. The most common symbols are given the shortest representation form. This can most easily be seen with a binary tree.[4]

For example the ASCII text "HELLO WORLD, THIS IS THE COMPUTER TALKING" gives us the probabilities:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>3/41</td>
</tr>
<tr>
<td>E</td>
<td>3/41</td>
</tr>
<tr>
<td>L</td>
<td>4/41</td>
</tr>
<tr>
<td>O</td>
<td>3/41</td>
</tr>
<tr>
<td>' '</td>
<td>6/41</td>
</tr>
<tr>
<td>W</td>
<td>1/41</td>
</tr>
<tr>
<td>R</td>
<td>2/41</td>
</tr>
<tr>
<td>D</td>
<td>1/41</td>
</tr>
<tr>
<td>,</td>
<td>3/41</td>
</tr>
<tr>
<td>T</td>
<td>4/41</td>
</tr>
<tr>
<td>I</td>
<td>3/41</td>
</tr>
<tr>
<td>S</td>
<td>2/41</td>
</tr>
<tr>
<td>C</td>
<td>1/41</td>
</tr>
<tr>
<td>M</td>
<td>1/41</td>
</tr>
<tr>
<td>P</td>
<td>1/41</td>
</tr>
<tr>
<td>U</td>
<td>1/41</td>
</tr>
<tr>
<td>K</td>
<td>1/41</td>
</tr>
<tr>
<td>N</td>
<td>1/41</td>
</tr>
<tr>
<td>G</td>
<td>1/41</td>
</tr>
<tr>
<td>A</td>
<td>1/41</td>
</tr>
</tbody>
</table>

Using these probabilities we can create a huffman tree. Note that several possible huffman trees exists. Figure 2.1 is one such tree.

Huffman coding is an example of statistical compression. It works well on data where we have a statistical model, like text from the English language.
2.1. DATA COMPRESSION

Figure 2.1: A huffman tree for the text "HELLO WORLD, THIS IS THE COMPUTER TALKING".

For other cases such as compressing never before seen data we have other methods.

The dictionary method’s main idea is to represent each string of symbols as
2.2. CELLULAR AUTOMATA

a token found in a dictionary. This dictionary can either be dynamic, allowing additions, or static.[5]

Information Theory

The beginnings of information theory was laid out by Claude Shannon in 1948 and describes information in a quantitative way.

The entropy of some data ’d’ is how much information this data contains. We can say that a data source with more entropy is more random. This way we can use the notion of entropy to describe how well a block of data is compressed.[6]

General data compression schemes

For most popular data formats exists specific compression tools, such as JPEG for pictures, lame for music and MPEG for movies.

For general data, when the content cannot be recognized, such algorithms prove useless and one must instead look at how the bit patterns look like. Most general data compression schemes are based on dictionaries; Finding what bit-strings appear the most and replacing those with shorter versions, thereby removing redundancy. When all redundancy is removed the file is said to be optimally compressed and contains only random data (except the header for decompressing).[7]

One of these methods, RLE run length encoding, achieves compression by replacing a string such as "aaabbcceaaaa" with "3a3b3c4a" which is shorter. Of course if the string to be compressed is more random, say "abjgaski" then this method will not be able to save anything.

On Linux with KDE, the available compression schemes in the default compression tool "ark" were: zip, 7-zip, rar, gzip, bzip, tar-z, xz and lzma. Comparing methods of lossless data compression has been done at several occasions. Generally the algorithms perform at a comparable level.[8][7]

2.2 Cellular Automata

Created in 1940’s by John Von Neumann, cellular automata is a mathematical model to simulate evolving cells in a lattice of infinite space where time is discrete and all rules are only applied locally at cell level. The original purpose of Cellular Automata was to study self-replication.[9][10] Cellular automata can be simulated in any dimension (1, 2, 3, etc.). But, in this thesis we will only consider 1 and 2 dimensional space.

The evolution of cellular automata springs out of an initial configuration, also known as seed, that along with its rules determines the rest of its history. Knowing the seed and rules for evolution, any following generation can be calculated.
2.2. CELLULAR AUTOMATA

The rules of a cellular automata describe how cells are to be evolved. The state of a cell at time $t$ is usually dependent on its neighboring cells at time $t$, and its state at time $t-1$.\[11][12] Which cells belong to its neighborhood is part of the rules, but is limited by its dimensionality.

The cells of a CA each have one of a finite number of states. The most common CAs have 2 states (binary CA) where 0 represents a dead cell and 1 represents a live cell. In this thesis we will focus on binary CA.

In a 1 dimensional cellular automata the field is usually represented as an array.

\[
\begin{matrix}
X & X & X & 0 & X & 0 & 0 & 0 & 0 & 0 & 0 & X & 0 & X
\end{matrix}
\]

Here the $X$’s represent live cells, $O$’s are dead cells. The neighborhood of 1 dimensional cellular automata can only be to the left and/or right, but any number of cells specified by the rules. Toroidal 1 dimensional CAs are not uncommon.

Two dimensional cellular automata can be represented by a matrix such as Figure 2.2, the white squares are live while the grey are dead.

![Figure 2.2: 2d Cellular Automata (Game of Life, B3/S23).](image)

Here the neighborhood can be either along the square axis and/or along the diagonals. Note that neighborhoods do not need to be symmetric.

In this thesis only Cellular Automata in finite space will be studied and in the form of a Null Boundary; Cells outside the boundary will be considered permanently dead.
2.2. CELLULAR AUTOMATA

The rules for binary CA is generally written as Bx/Sy (Golly format[13]), where x is the number of neighbors required for a dead cell to become alive(Born), and y is the number of neighbors a live cell needs to stay alive(Survive). Both x and y can contain several numbers and even the same numbers or be empty. Figure 2.3 shows several valid binary CA rules. Note that rules for CA with more than 2 states are written differently and are not considered in this thesis.

The x and y part of the rule may contain numbers from 0 to 8, that is up to 9 numbers in each part. Using this information we can calculate the amount of different binary CA rules. There are thus $2^{29} = 262144$ different rules for binary CA in the Moore neighborhood.

- **B1/S123** A rule where dead cells become alive with 1 neighbour and survive with 1, 2 or 3 neighbors. Will grow to infinity at the "Speed of Light".
- **B/S3** A rule where dead cells cannot become alive, but live cells with 3 neighbors stay alive. Note that with no births the CA can never grow.
- **B3/S012345678** A rule named "Life without Death" where cells never die and new cells are born if they have 4 neighbors.
- **B2/S** A rule named "seeds" where cells die every generation. Due to the low requirement for birth it will usually grow extremely fast.

Figure 2.3: Example binary CA rules in the Moore neighborhood

2.2.1 Moore neighborhood

This paper will only consider the Moore neighborhood for 2d Cellular Automata. In this neighborhood model all cells have 8 neighbors as can be seen in ??.[10] Note that in finite space cells that are in the corners or along the border have 3 and 5 neighbors respectively.

2.2.2 Speed of Light

In Cellular Automata movement speed is measured by how many cells the pattern moves per generation. The maximum speed is 1 cell per generation and the Speed of Light in Cellular Automata has thus been defined as 1 cell/generation.[10]

2.2.3 The Game of Life

One of most popular rules for cellular automata is John Horton Conway’s Game of Life(1970). It is a 2 dimensional cellular automata with the rule
2.2. CELLULAR AUTOMATA

Figure 2.4: The Moore neighbourhood for a cell X, this cell's neighbourhood is marked with o's.

B3/S23. This rule was chosen by Conway because it exhibited interesting features when simulating.[12]

Simulation:[10]

- If a live cell has less than 2 neighbors, it dies of loneliness.
- If a dead cell has exactly 3 neighbors it is born (becomes live).
- If a live cell has more than 3 neighbors, it dies of overcrowding.
- In all other cases, no change is made.

There has been a lot of research into different aspects of the Game of Life. It has been proven Turing Complete (allowing Universal Computability) and self-replicating patterns have been found, as well as frequent patterns and a huge collection of interesting patterns.[14][15]

There are many patterns in the Game of Life that cannot appear naturally because they contain some cell configuration (Orphan) that the rule B3/S23 does not generate. These patterns are named Gardens of Eden, they have no previous pattern, but as all other have a next pattern.[16]

As of writing, the currently known smallest Orphan was found by Marijn Heule, Christiaan Hartman, Kees Kwekkeboom, and Alain Noels. This Orphan fits a 10x10 area.[17] There are

$$2^{10^2} = 1267650600228229401496703205376$$

different 10x10 areas in the Game of Life.

By reading a 1MB file (1MB = 1024 * 1024 * 8 bits = 8388608 cells) into a square area (size $$\sqrt{1024 \times 1024 \times 8} \approx 2897$$), there is a chance that it will contain
2.3. THE BASIC IDEA OF COMPRESSION USING CA

at least one of this Orphans. If all the possible 10x10 areas where equally likely, then the chance of a single 10x10 area being the given Orphan would be

\[ \frac{1}{126765060228229401496703205376} \]

A square matrix of size 2897 can hold \((2897 - 10 + 1)^2 = 8340544\) 10x10 areas. Thus we can calculate the chance of a 2897 matrix containing a given 10x10 matrix as

\[ \frac{8340544}{126765060228229401496703205376} \]

If we also search for all the other known Orphans, this chance will increase further. It is therefore given that many files are Gardens of Eden (when using Game of Life).

2.2.4 Evolving Cellular Automata

Because time in Cellular Automata is discrete it is possible to iterate over time in steps of one generation at a time. To go from one time step to the next, the cellular automata rules must be applied once to every cell in the configuration. This can be seen as a function to evolve the cellular automata. Figure 2.5 shows how a configuration is evolved in Game of Life, it is important to note that the evolution function can only move time forward and that there is always exactly one next generation.

Game of Life is defined with the rule B3/S23, this means that a dead cell becomes live when it has 3 live cells as neighbors and a live cell survives if it has 2 or 3 live neighbours. Not that this means that a live cell dies if it has 0,1,4,5,6,7 or 8 live neighbours. In Figure 2.5 the cells that will die are first marked with red, the 2 light red dies because it has 1 neighbour and the dark red (center cell) dies because it has 4 neighbours. Next the cells that are born is marked with green, 2 dead cells have 3 live neighbours each.

When all cells have been checked the actual cells are changed to give the next generation \( \tau + 1 \). Note that a cell can only be changed once each time step as all cells are updated at the same time.

Even though there is only one next generation for each configuration, several previous generations may exist.

2.3 The basic idea of compression using CA

In order to compress data, the idea is to lower the number of bits required to store it. That is representing information as effectively as possible. Compression is achieved by eliminating redundancy, thereby increasing the entropy of the data.[5]
2.3. THE BASIC IDEA OF COMPRESSION USING CA

Figure 2.5: How to evolve one configuration to the next generation in Game of Life (B3/S23)

Since any data can be generated from a function, although complex, it should be possible for any data to be represented as that function instead. Investigation into the generative properties of cellular automata has shown that Cellular Automata can be used to generate random data with high efficiency. [18]

It has further been shown that Cellular Automata can be used as a universal finite pattern generator. Meaning that any given finite pattern may be generated by a Cellular Automaton. It is however not given if this can be applied to binary CA or 2 dimensional CA space.[19]

The Idea
Since Cellular Automata can generate random data sequences and one can only evolve a CA forward it means one could in theory represent some data with a configuration that leads to the original data. An algorithm to recover the original data would be needed, and it would need to know which rule to use and how many generations to evolve. Now, given that one could find a configuration in some CA rule that leads to the data, if it is smaller than the original then the data has in fact been compressed.

Thus, the problem is to find a configuration that requires less bits to represent and evolves into the data to compress, possibly after many generations. If such a configuration is found then it can be stored, along with number of generations to go, instead of the original data. It has been shown that 2D Cellular Automata express interesting matrix properties, especially in the group of CA that are non-uniform.[20]
2.4. CELLULAR AUTOMATA TRANSFORMS

One important aspect of cellular automata in regards to matrices is the fact that no matter the size of the matrix, all cells are checked and updated at every step. This means that in only 1 or 2 steps a matrix, independent of the size, will change a lot. Of course this depends on the configuration and rule used.

In data compression, entropy is seen as the limiting factor. If the entropy is too high, the data cannot be compressed further. But in cellular automata compression, based on this idea, one can say that the ratio of Gardens of Eden is the limiting factor as it limits the movement backwards.

2.4 Cellular Automata Transforms

Cellular Automata Transforms (CAT) is a method for finding cellular automata that can be used to transform data into some other more desirable form. It has been able to both compress multimedia data and encrypt text.[21]

The way Cellular Automata Transforms has been used for compression is lossy. The original data was transformed by the CAT to a compressed form that is only an approximation of the original data, because of this the method cannot be directly applied to general data which requires lossless compression.

Because of this fact, and time constraints, Cellular Automata Transforms were not investigated further in this thesis.

2.5 Reversible Cellular Automata

A cellular automata is said to be reversible if given any configuration there is exactly one previous configuration. Finding this previous configuration follows the same procedure as finding the next.[11]

Trying to find the previous configuration of a non-reversible CA however is not as trivial, previous attempts usually includes brute force as the only way to find any of the previous configurations if they exist. It is also unclear if a previous configuration exists.

Since a previous configuration always exists for reversible cellular automata, it seems likely that they may be used to solve my generative problem.

2.6 Java

Java is an Object Oriented programming language with syntax much like c++. Java code is compiled into byte code which is read by a virtual machine (a JVM, java virtual machine). A compiled java file becomes a ‘.class’ (or ‘.jar’ for several files) which the JVM can read. Since code is not compiled to machine code any computer with a compatible JVM can run java code. Java is as such platform independent and easy to distribute.
In its early days, Java was quite a bit slower than code compiled with C. In recent years this has changed a lot however and Java has become almost as fast (and sometimes the same) as C.

### 2.7 Genetic Algorithm

The genetic algorithm (GA) is part of evolutionary algorithms, a field of algorithms inspired by nature and evolution. GA is a search and optimization algorithm that combines the strengths of brute force and local optimization. It is a good overall approach to many problems, but is usually beaten when specific algorithms written for the task exists.[22]

The algorithm simulates evolution of *candidate solutions* as genes fighting for survival. The more fit genes are allowed to create offspring, while the bad genes are weeded out. This process will take the fit genes and combine them, while discarding the bad genes. In this way the genetic algorithm will converge to a single solution. This solution is usually a good solution, but finding the absolute best cannot be guaranteed.

In order to simulate a real evolution the algorithm requires several operations to be defined:

- A crossover function to breed 2 (or more) genes to create offspring
- A mutation function to affect the offspring randomly
- A fitness function that rates the gene based on how good it is (in our case how close it is to the solution)
- A selection to decide which genes to breed, usually based on their fitness.

Although the Genetic Algorithm is quick to find locally good solution, the global optimum may never be found. It is also so that the Genetic Algorithm will investigate the same gene several times if it comes up.[23]

### 2.8 Previous algorithms for Reversing Cellular Automata

As of writing there are 4 known algorithms for “listing preimages”, that is finding the previous configurations (Ct-1) that lead to a specified configuration (Ct) in rule R. All of these were designed for 1 dimensional CA and can be solved in linear time depending on configuration size. All of them also make use of a model for permuting the total number of combinations of a diagram known as a De Bruijn Diagram.[24]

In this thesis we will consider only the two latest of these algorithms "Trace and backtrack" and "Count and List" as they are newer and contain the major
parts of the other two.[24]

The first 2 steps of these algorithms is to create "preimage diagrams" for each cell and then link these diagrams together to create what they call a "preimage network". After traversing the network we find all the previous configurations as successful paths. Figure 2.6 shows an example Preimage Network with one valid path.

![Figure 2.6: An example trace in a Preimage Network for 4 cells in a cyclic array. Grey boxes indicate a live cell value, white is dead. The only valid trace is given by the thick dashed line, as it is the only line reaching from the first cell to the last cell. Note that because this is a cyclic array, the first cell is also the last.](image)

Because these algorithms use 1 dimensional configurations it is possibly to trace simply from left to right (or right to left) as each cell can only have neighbours to the left or to the right that affect each other. If one were to consider 2 dimensional CA space one would have to iterate in all directions of the neighborhood. In the Moore neighborhood, where each cell has 8 neighbors, this would mean 4 directions of tracing.

The complexity can be described as the number of links between cells needed to be inspected to know if a solution is valid. When considering finite one dimensional CA the amount of links equal the size and the complexity is linear. In a square finite 2 dimensional CA with the Moore neighborhood and size \( s \) the number of links is given by

\[
L_s = \sum_{i=2}^{s} (8i - 10), \quad (s > 1, \text{ for } s = 1 \Rightarrow L_s = 0).
\]

Because of the abundant growth of links to inspect, and time constraints, the algorithms were not studied further in this thesis.
Chapter 3

Design

In this chapter the implementation of each algorithm is specified. Most of the actual code can be found in the appendix. Each algorithm tested has its own section in this chapter where design and implementation details are specified.

Test System

- Operating System: 32bit Ubuntu 11.10 with PAE kernel
- CPU: Intel Core i7 3820 (4 cores @ 3.6 GHz)
- Memory: 16GB
- Java version: OpenJDK 1.7 (1.7.0_147-icedtea)

3.1 Reversible Cellular Automata

In order to test the possibility of using Reversible Cellular Automata for compression an implementation was attempted in Java. The implementation was developed for both second order and block cellular automata. The algorithm will iterate over time step by step and attempt compressing the current configuration at every step.

3.1.1 Block Cellular Automata

For block cellular automata, the critters rule and the tron rule was used.

The implementation of the function to calculate the next generation, by applying the local rule to each block, can be found in appendix subsection 7.6.2.

Critters rule

For each block of 4 cells an operation is carried out depending on the number of live cells in the block.
3.2. GENETIC ALGORITHM

- If there are exactly 2 live cells, no operation is carried out.
- If there are 3 cells, the block is flipped (rotated 180 degrees).
- Unless the block had 2 cells it is inverted (all live cells become dead and dead cells become live).

The implementation of the Critters local rule can be found in appendix subsection 7.6.4.

Tron Rule

For each block of 4 cells, if all the cells in the block is the same state then the block is inverted.

The implementation of the Tron local rule can be found in appendix subsection 7.6.3.

3.1.2 Second-Order Cellular Automata

Because second order cellular automata requires 2 consecutive states to move in any direction, a randomly generated matrix was used to represent $τ - 1$ while the input data to be compressed was set as $τ$. The algorithm then iterates over time to get $τ + 1$, $τ + 2$ and so forth. As with block cellular automata, each configuration encountered is compressed.

Different rules were tested, among them ‘B234/S234’ because it makes the configurations look very different at each step. The implementation of the function to evolve the CA, $τ + 1 = τ - 1 \oplus τ'$, can be found in appendix subsection 7.6.1.

3.2 Genetic Algorithm

The genetic algorithm was implemented in Java to search for previous configurations of an input configuration and given rule. Two crossover and two mutation methods where implemented. Tournament selection was used to select which genes from the population to breed.

3.2.1 Crossover

The two crossover operations implemented are named XOR and random. The algorithm randomly chose which operation to use, with 50% chance for each operation.

XOR crossover

In XOR crossover the XOR operation is used on the 2 parents to give 1 offspring. The XOR ($\oplus$) operation is the exclusive 'OR' of the input. An example of how the operation works is shown in Figure 3.1.
3.2. GENETIC ALGORITHM

Random crossover

Random crossover was implemented as iterating over both parents choosing randomly either the first or second parent to copy from. At each step a new random number is drawn giving 50% chance to both. Hence the offspring will on average consist of 50% of each parent. Figure 3.2 shows how two parents may be combined by random crossover.

3.2.2 Mutation

After each crossover mutation is applied in 50% of the cases. The operations implemented is Flip Mutate and Edge Flip Mutate.

Flip Mutate

Flip mutate is a simple operation where each cell in the configuration is inverted by a configurable chance. On average the percentage of cells inverted is equal to the chance set. Although the operation is simple it will successfully randomize the configuration a configurable amount. This is good because we can then increase the mutation strength if we need more exploration.

Edge Flip Mutate

Edge flip mutate was derived from flip mutate to only iterate over those cells that have at least 1 live neighbor. Each of these cells are inverted by a configurable chance. Figure 3.3 shows which cells have a chance to be inverted in a given configuration. This mutation boosts exploration less than Flip Mutate as the randomization is only applied to some of the cells. It does however mimic the evolution of CA a little in that only the edges can be moved, new cells will not appear away from the other cells.

3.2.3 Tournament Selection

Tournament selection is a selection operation that favors the more fit genes. It operates by selecting a set amount of genes (contestants), compares them and selects the best. In order to get more than 1 gene the operation is run again from the start independent of the previous run.

A good suggestion for the number of combatants is 3 as this keeps the algorithm from converging too fast or too slow. [22]
3.2. GENETIC ALGORITHM

Figure 3.2: Random crossover with 2 parents producing 1 offspring

Figure 3.3: From the configuration above, the cells with a triangle have a chance to be inverted by edge flip mutate
3.3 COLOURING ALGORITHM

3.3 Colouring Algorithm

In order to find a new way to search for previous configurations, a test was carried out to find how one could accomplish this with only pen and paper. A deduction was made that the person solving it would look at each individual live cell and see how it can have appeared. To mark the possibilities for each cell, colour pencils were used. Figure 3.4 shows how one configuration was colored.

![Coloring a 4x4 matrix](image)

Figure 3.4: Coloring a 4x4 matrix

Then, by guessing permutations of the colors, each live cell was given neighbours to satisfy its requirements. If a requirement could not be met a new permutation was guessed and the old was discarded. Although a slow process requiring several guesses, the algorithm did work out.

In order to distinguish the cells that have been added from the original cells a convention for representation was devised. Table Figure 3.5 shows what each symbol means. It is important to note that a finally dead cell is not the same as a cell that dies now, the former is dead in the current generation and stays dead in the next, while the latter is alive in the current but dead in the next. New live cells that are added to satisfy another cell must be added as ‘dies now’, because the algorithm wants to get to the next generation it was given.

First a permutation of colors must be guessed, here the number of permutations depends on the number of live cells, the length of the rule and the cell positions. Note how in Figure 3.4 the corner cells can only be green because there is only 3 available neighbours for that cell, hence the only rule that fits is 2 which is green. Using the colormap permutations of colors can be guessed. To represent a guess the color chosen for each cell is listed in order, counting cells from the top left to bottom right. Figure 3.6 shows the order of counting cells, note that in this case the dead cells are skipped.

Figure 3.7 shows another configuration with a corresponding colormap. A sample guess could thus be OOXOXO, which would look like Figure 3.8. Here the symbols from Figure 3.5 is used together with the color. The next step is to try satisfying all the cells at the same time.
### 3.3. COLOURING ALGORITHM

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Symbol]</td>
<td>Finally alive; Alive in both the current and the next generation</td>
</tr>
<tr>
<td>![Symbol]</td>
<td>Undecided; Unknown in the current, dead in the next</td>
</tr>
<tr>
<td>![Symbol]</td>
<td>Dies now; Alive in the current generation, dead in the next</td>
</tr>
<tr>
<td>![Symbol]</td>
<td>Born; Dead in the current generation, alive in the next</td>
</tr>
<tr>
<td>![Symbol]</td>
<td>Finally dead; Dead in both the current and the next generation</td>
</tr>
</tbody>
</table>

Figure 3.5: Symbols devised for backtracking

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>
```

Figure 3.6: How counting the cells was done

Figure 3.7: Another configuration and corresponding colormap
3.3. COLOURING ALGORITHM

Figure 3.8: One permutation of a colormap ready to be tested

Figure 3.9 shows several guesses to satisfy a colormap, the rightmost holds a valid solution. Note how it is possible to discard an approach when a color cannot be satisfied, but before all cells have been tested. It was also found that several guesses could have been eliminated when looking at multiple neighbouring colors at the same time. For example in Figure 3.9 if the corner is orange then none of its neighbours may be orange, otherwise it cannot get enough neighbours.

The pen and paper approach was coded in Java and tentatively named the Colouring Algorithm, from the use of colour pencils. The algorithm was implemented in Java with an object oriented approach. The coloring part of the algorithm was implemented in a class named ColorMap while the algorithm for attempting to solve each permutation was implemented in a class named PermMap.

The code was written with thread support for multi-core systems. The threading is based on a monitor and worker model, where one thread creates worker threads that do the actual jobs and monitor their states. A separate thread works on pushing work onto an agenda, which the monitor reads and distributes to the workers. The system also does limited load-balancing by increasing the number of workers when there is a lot of work and decreasing the amount when there's little load per worker. Appendix ?? has the code used in conjunction with PermMap and ColorMap.

3.3.1 Color Map

In order to limit the number of iterations of the algorithm, it was decided to iterate only over the permutations of how each cell can have been formed using the colouring approach described above. For this the "Color Map" was created to represent such colored matrices as shown in Figure 3.4. The color map describes all possible ways that could have caused each live cell being alive, either by birth or survival. Using this information iteration is possible over the permutation of colors.

For example, a brute force approach to iterating over all 6x6 binary matrices would be \(2^{36}\) iterations. If 12 of the cells are live, and there are 3 colors,
Figure 3.9: Several attempts at solving colormaps, one attempt has a solution. The red squares indicate colors that don’t fit together, resulting in the permutation being discarded.
then the colormap would have $12^3$ permutations (roughly $2^{3.3^2}$ for comparison).

Note that a permutation of the color map is not a solution, but rather a possible explanation of how the cells can have entered their given state. In order to find a solution the permutation must be satisfied as in Figure 3.9.

In order to further eliminate iterations it was also looked at cells that exist in corners or next to borders. For these a check was made that each color given to them is possible given their maximum amount of neighbours. For example the corner cells can only have 3 neighbours, while cells at the borders can only have 5.

Another approach used to eliminate iterations was to look at cells that depend on each other. If one cell requires an amount of neighbours and of these cannot be in the required state, then that color can be eliminated.

The implementation of Color Map is based on an array containing the possible colors for each cell. Each color is a Byte that references one of the numbers in the rule used. The rule B3/S23 has 3 numbers = 3 color rule. Here 0 = B3, 1 = S2, and 2 = S3.

Listing 3.1: Iteration over the ColorMap is provided by the gotoNext() method

```java
public void gotoNext(){
    if(numColors <= 1)
        return; // Nothing to do.
    if(!hasNext())
        return;
    add(colors.length - 1);
    //this.gotoRandom(); an option to iterating..
    for(Elimination e : eliminations){
        if(e.isEliminated()){
            do {
                add(colors.length - 1);
            } while(e.isEliminated());
        }
    }
}
```

The gotoNext() method calls the add() method causing an increment in the current colors. It then checks all the eliminations that applies to this Color Map. If any of them trigger then add() is called again. This process repeats untill the elimination that triggered is satisfied.

Listing 3.2: The method add() is a recursive method that increments the color of the cell in the last position

```java
private void add(int pos){
    if(pos < 0 || pos >= colors.length){
        this.broken = true;
    }
}
```
3.3. COLOURING ALGORITHM

```java
return;
}
if (aviColorsCounter[pos] < aviColors[pos].size() - 1) {
    colors[pos] = aviColors[pos].get(++aviColorsCounter[pos]);
    return;
} else {
    aviColorsCounter[pos] = 0;
    if (aviColors[pos].size() == 0)
        return;
    colors[pos] = aviColors[pos].getFirst().byteValue();
    add(pos - 1);
}
```

The add() method is a recursive permutation method that iterates the possible colors of each cell. By increasing the color, from right to left the method acts like a simple adder. Note however that since each cell may have different colors available the method needs to check which colors each cell may have. This information is stored in the array avi Colors.

In order to represent eliminated colors, a new class was created to support adding cells and which color was eliminated. This way an elimination can be made which includes several cells and colors. The implementation of the Elimination class can be found in appendix section 7.2 on line 105. The method optimize() that parses the Color Map and makes eliminations can be found in the same appendix on line 234.

Because the optimize() method may create duplicate eliminations another method was deviced to weed out the eliminations. The method optimizeEliminations(), in appendix section 7.2 on line 54, compares all the eliminations registered to see if any of them are equal or contain eachother.

**On complexity**

When making the colormap we can calculate how many different permutations of colors exist.

This is given by a simple combinatorial formula:

\[
\text{count} = \text{colors}_0 \times \text{colors}_1 \times \text{colors}_2 \times \ldots \times \text{colors}_n, \quad \text{colors}_n \text{ being the available colors of cell n, etc.}
\]

To calculate the number of permutations for a 3x3 matrix with 4 cells as shown in Figure 3.10:

1. cell$_0$ can have 3 different colors
2. cell$_1$ can have 4 different colors
3. cell$_2$ can have 1 color
4. \( \textit{cell}_3 \) can have 3 different colors

The calculation becomes

\[
\text{count} = \text{colors}_0 \times \text{colors}_1 \times \text{colors}_2 \times \text{colors}_3 = 3 \times 4 \times 1 \times 3 = 36 \text{ different ways to distribute the colors.}
\]

This number however is then affected by which eliminations that have been made. If for example \( \textit{cell}_0 \) cannot be blue while \( \textit{cell}_1 \) is orange, then all the permutations that contain \( \textit{cell}_0=\text{blue}, \textit{cell}_1=\text{orange} \) must be subtracted. In this case there are \( 1 \times 1 \times 1 \times 3 = 3 \) such permutations, which gives the new total permutations as 33.

If however it was found that in addition \( \textit{cell}_2 \) cannot be orange if \( \textit{cell}_1 \) is orange, then a new subtraction must be made. This time however it becomes much more difficult, as the overlap from the first elimination must not be subtracted again. This can be visualized by a venn diagram as seen in Figure 3.11. If one subtracts directly without considering the overlap, the overlap will have been subtracted twice. Calculating the overlap however becomes too difficult to consider for this thesis.

---

**Figure 3.10:** Color map for a 3x3 matrix with 4 cells

**Figure 3.11:** Venn diagram showing overlap between 2 subtractions
3.3. COLOURING ALGORITHM

3.3.2 Perm Map

In order to find out if a given distribution of colors from a colormap is the correct one, the PermMap was devised. Its function is to try to satisfy all cells given its color from the colormap while not adding any unwanted cells. This is accomplished by defining that any new live cells added must be dead in the next generation (symbol "Dies Now" from Figure 3.5), this way we can add live cells to satisfy other cells and make them disappear so the configuration will still match the original. If successful it means the color distribution leads to a goal and the PermMap will give us the first valid solution it found.

The PermMap iterates over all cells until it finds a cell that cannot be satisfied or all cells are satisfied, in either case we are done. Figure 3.12 shows both cases.

![PermMap Diagram](image)

Figure 3.12: Two permMaps from the same configuration and rule. The left cannot satisfy the highlighted cell, the right has satisfied all cells.

For each cell we check if it still has any hope of being satisfied, if not then we stop. If there is only one way to satisfy the cell, that ‘way’ is carried out. Figure 3.13 shows two such cases. In the first case the corner cell requires 3 cells, since there is exactly 3 cells available all of them must be live for it to be satisfied. The second case shows a cell that requires 4 neighbours while 5 exists, however one of them has previously been marked as finally dead meaning only 4/5 cells are available. With the requirement being 4 and the available being 4 there is only one way to satisfy it.

If no change happened in the last iteration (of all cells) and we didn’t satisfy all cells we need to recursively check all options for satisfying the cells remaining. Figure 3.14 shows two cases where recursion is needed as there is more than one way to satisfy the cells. The first case shows a cell requiring 3 neighbours when 8 is available. To find how many combinations exist, and hence
3.3. COLOURING ALGORITHM

Figure 3.13: The cells in the left PermMaps can only be satisfied one way, so they can be solved directly.

To determine how many recursions must be made, the calculation is an unordered combinatorial problem\[25\]. There are \( \binom{k}{n} \) \((k = 3, n = 8)\) \(\Rightarrow\) \( \binom{3}{8} = \frac{8!}{3!(8-3)!} = 56\) combinations for the first case. In the second case the cell requires 4 when there is 5 available, this gives \( \binom{4}{5} = 6\) combinations.

Figure 3.14: The cells can be satisfied in multiple ways, so recursion is needed.

To recurse as little as possible we find the \textit{smallest choice}, that is the cell that has the least possible ways to be satisfied. Figure 3.15 shows a 3x3 matrix where a choice must be made. The orange cell requires 1 from 3 \(\Rightarrow\) \( \binom{1}{3} = 3\), while the green cell requires 2 from 5 \(\Rightarrow\) \( \binom{2}{5} = 10\). This gives the \textit{smallest choice} as the orange cell where only 3 forks are needed. We iterate over its ways to
be satisfied and recurse. If any of the forks find the solution, we are done.

Figure 3.15: The orange cell requires 1 from 3, while the green cell requires 2 from 5.
Chapter 4

Results and Analysis

4.1 Reversible Cellular Automata

Using the code provided in the appendix we searched out configurations of various size, backtracked them and compressed them with the built in java deflate algorithm (zip) at each step. We found that we could easily move forward and backward through time while the configuration seemed to change drastically every step. We tried both Block cellular automata with the seeds and tron rules as well as second order CA with the B234S234 when trying to compress the data. The result of the compression however did not show much
4.1. REVERSIBLE CELLULAR AUTOMATA

Figure 4.2: The Reverse CA player evolving a 10x10 matrix 500 steps forward using a randomly generated previous 10x10 matrix. The rule used was B234/S234 with second-order Cellular Automata. The algorithm took less than a second to finish.
4.1. REVERSIBLE CELLULAR AUTOMATA

![Reverse CA Player](image)

Figure 4.3: The Reverse CA player evolving a 12x12 matrix 10000 steps forward using a randomly generated previous 12x12 matrix. The rule used was B135/S024 with second-order Cellular Automata. The algorithm took approximately 150ms to finish.
4.2. GENETIC ALGORITHM

promise.

For block cellular automata it was found that since no information is lost[14], it will not compress data (or very little). The reason for this comes from the ability to move in both directions, meaning it does not change itself so that the path is lost.

When looking at second order CA there exists another problem; To move in any direction, 2 configurations are needed because the algorithm needs the configuration at both time $t$ and $t-1$ to calculate $t+1$. This lead us to generate a random configuration as $t-1$ and set the data to be compress as $t$. We can then calculate $t+1$ and further.

However, this does not solve the problem, say that we found a configuration at $t+23$ that could be compressed to just 30% of the original. If we wanted to store this in place of the original we would also need to store the configuration at $t+24$ in order for the decompression algorithm to be able to get back to the original data (plus the number 23 to know how many steps to go). When we combine the size of $t+23$, which was small, with $t+24$ we see that the sum of these configurations must be less than the original. When testing this did never happen.

In order to combat the latter obstruction we investigated the possibility to store 2 consecutive configurations, but with a space between them. Although it would be possible to find the original, this would the require the decompression algorithm to work more as it first needs to find the missing part. In testing however, even this did not become smaller then the original.

Another problem we ran into with reversible CA was the fact that in finite space loops must appear. This can easily be observed with a thought experiment; Consider an algorithm that fills a 3x3 matrix with boolean values (true or false). There are $2^{3 \times 3} = 512$ possible 3x3 matrices created by this algorithm. If the algorithm is run $512 + 1$ times, the algorithm must have created atleast 1 matrix that has already been created.

This applies to Cellular Automata also, in finite space there is a finite number of possible configurations. If the CA is run more times than the number of configurations then the CA has entered a loop. It is not known however when it entered a loop or how man cycles the loop has. Note that a loop could in theory contain all possible configurations in the given space, or even just 1 (normally called still life)[14].

4.2 Genetic Algorithm

The GACABacktracer was tested several times with different rules and configurations. All tested configurations was known to have at least one previous
4.2. GENETIC ALGORITHM

configuration in the given rule (generated by running the CA forward one step on the input first). Here follows a sample run:

### Sample Run

<table>
<thead>
<tr>
<th>Points</th>
<th>G</th>
<th>GA complete</th>
<th>Entropy</th>
<th>Best</th>
<th>CurrentAverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>240</td>
<td>10</td>
<td>50 generations of CA has passed.</td>
<td>3198082052</td>
<td>199</td>
<td>82.916664%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The search took 8 seconds.</td>
<td></td>
<td>83.1192</td>
<td></td>
</tr>
</tbody>
</table>

Here follows the best solution we found (G=2):

<table>
<thead>
<tr>
<th>Points</th>
<th>G</th>
<th>GA complete</th>
<th>Entropy</th>
<th>Best</th>
<th>CurrentAverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>360</td>
<td>10</td>
<td>50 generations of CA has passed.</td>
<td>31688012</td>
<td>360</td>
<td>87.5%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The search took 2 minutes 13 seconds.</td>
<td></td>
<td>87.5%</td>
<td></td>
</tr>
</tbody>
</table>

The GA did not manage to find a solution, but instead found a configuration that looks 87.5% similar. In this test the GA was given an initial population of 10'000 genes and 50 generations to evolve.

In the following test the GA was given 5000 generations to evolve:

### Test 2

<table>
<thead>
<tr>
<th>Points</th>
<th>G</th>
<th>GA complete</th>
<th>Entropy</th>
<th>Best</th>
<th>CurrentAverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>39315</td>
<td>10</td>
<td>5000 generations of CA has passed.</td>
<td>31849362</td>
<td>304</td>
<td>93.61111%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The search took 1 minute 37 seconds.</td>
<td></td>
<td>93.61111%</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Points</th>
<th>G</th>
<th>GA complete</th>
<th>Entropy</th>
<th>Best</th>
<th>CurrentAverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>39315</td>
<td>20</td>
<td>5000 generations of CA has passed.</td>
<td>31742474</td>
<td>338</td>
<td>93.88889%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The search took 1 minute 2 seconds.</td>
<td></td>
<td>93.88889%</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Points</th>
<th>G</th>
<th>GA complete</th>
<th>Entropy</th>
<th>Best</th>
<th>CurrentAverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>39315</td>
<td>30</td>
<td>5000 generations of CA has passed.</td>
<td>31732002</td>
<td>338</td>
<td>93.88889%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The search took 1 minute 12 seconds.</td>
<td></td>
<td>93.88889%</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Points</th>
<th>G</th>
<th>GA complete</th>
<th>Entropy</th>
<th>Best</th>
<th>CurrentAverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>39315</td>
<td>40</td>
<td>5000 generations of CA has passed.</td>
<td>31723002</td>
<td>338</td>
<td>93.88889%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The search took 1 minute 22 seconds.</td>
<td></td>
<td>93.88889%</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Points</th>
<th>G</th>
<th>GA complete</th>
<th>Entropy</th>
<th>Best</th>
<th>CurrentAverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>39315</td>
<td>50</td>
<td>5000 generations of CA has passed.</td>
<td>31713002</td>
<td>338</td>
<td>93.88889%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The search took 1 minute 31 seconds.</td>
<td></td>
<td>93.88889%</td>
<td></td>
</tr>
</tbody>
</table>

GA complete. 5000 generations of CA has passed.

The search took 1 minute 29 seconds.
4.2. GENETIC ALGORITHM

Highest score was ‘338’ points 93.88889%.
Here follows the best solution we found (G: 2):

```
xxxxxox
xxoxxxx
ooxxxoox
oxoxoxxx
xxoxxxox
```

Because one needs 100% for lossless compression, the number of generations was increased. Here another sample run where the number of generations to go was increased to 50’000 and the number of genes lowered to 1000:

```
<table>
<thead>
<tr>
<th>Points</th>
<th>Size 4</th>
<th>Generations/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>310</td>
<td>89831</td>
<td>32.154340836012864</td>
</tr>
<tr>
<td>311</td>
<td>89831</td>
<td>32.154340836012864</td>
</tr>
<tr>
<td>310</td>
<td>89831</td>
<td>32.154340836012864</td>
</tr>
</tbody>
</table>
```

Note: the output was cut here.

The algorithm did not manage to attain 100%, but instead stuck to the 95% that was discovered in the beginning.

From the results described above we can see that so far the GA has not achieved the goal. Even though the algorithm can quickly search over large matrices, without a 100% match the result cannot be used as intended for compression.
4.3. COLOURING ALGORITHM

4.2.1 Measures to increase exploration

In order to increase the amount of exploration of the algorithm a couple of measures were implemented.

Reinitiation

After the algorithm has run for a given length of time the population was wiped out and recreated as if restarting the algorithm, but keeping the best solution.

Increased mutation

By increasing the mutation rate of the algorithm the offspring will be more random, leading to less exploitation and more exploration.

4.3 Colouring Algorithm

A 2x2 matrix was backtracked 1 generation in rule B13/S024 as shown in ??.

The algorithm was able to list several configurations that lead to it and finished in 16 milliseconds on the test system.

Figure 4.5 shows the average time(milliseconds) to backtrace random configurations in Game of Life(B3/S23).

Under all tests run the algorithm was able to find the previous configuration, of any configuration and rule, if it exists. The time it took however grows increasingly by the configuration size as shown by Figure 4.5. Figure 4.10 graphs the data from the table and clearly shows increasing growth of average time taken to solve configurations with the same rule (Game of Life ‘S2/B32’) of different sizes.

In order to judge what type of complexity growth the algorithm has the data was also plotted in a logarithmic graph as shown in Figure 4.11.

Figure 4.12 shows the difference between the growth of brute force and the colouring algorithm. Brute force was defined as testing all possible matrices of the given size, using 0.2 ms to test each candidate. That is, both algorithms searching for all possible previous configurations. The graph clearly shows that the colouring algorithm grows slower.

We have successfully backtracked configurations of sizes 2, 3, 4, 5 and 6. The algorithm can handle most configurations of size < 4 in milliseconds while size 6 took 2 hours. The algorithm can take any configuration (provided it can be placed in a square matrix) as input as well as any binary CA rule.
4.3. COLOURING ALGORITHM

Figure 4.4: Backtracing a 2x2 matrix in the B13/S024 rule. The center configuration has 8 previous configurations leading to it in 1 generation. The algorithm took 17ms to finish.

<table>
<thead>
<tr>
<th>size</th>
<th>2x2</th>
<th>3x3</th>
<th>4x4</th>
<th>5x5</th>
<th>6x6</th>
</tr>
</thead>
<tbody>
<tr>
<td>time (ms)</td>
<td>6.5</td>
<td>31.9</td>
<td>1455</td>
<td>109866</td>
<td>7095285</td>
</tr>
</tbody>
</table>

Figure 4.5: Average time(in milliseconds) to backtrace matrices of different sizes in Game of Life (B3/S23) using the Colouring Algorithm.

Figure 4.6: Backtracing a 3x3 matrix in the B3/S234 rule. The left configuration has exactly 1 configuration leading to it in 1 generation. The algorithm took 8ms to finish.
Figure 4.7: Backtracing a 4x4 matrix in the B24/S45 rule. The center configuration has 4 configuration leading to it in 1 generation. The algorithm took 255ms to finish.
4.3. COLOURING ALGORITHM

Figure 4.8: Backtracing a 5x5 matrix in the B248/S45 rule. The top configuration has 3 configurations leading to it in 1 generation. The algorithm took 36512ms to finish.

Figure 4.9: Backtracing a 6x6 matrix in the B248/S45 rule. This configuration has no parents, hence it is a Garden of Eden. The algorithm took 158660ms to finish.
4.3. COLOURING ALGORITHM

Figure 4.10: Graph of size vs time with the Colouring Algorithm, note the increasing growth.

Figure 4.11: Logarithmic graph of size vs time with the Colouring Algorithm.

Figure 4.12: Logarithmic graph of size vs time comparing growth of Bruteforce and the Colouring Algorithm.
4.3. COLOURING ALGORITHM

Figure 4.13 shows a full backtrack of a 4x4 configuration, not how most of the encountered configurations have at least one previous configuration. The initial configuration of Figure 4.13 has 22 configurations leading to it.

By inspection of the tree in Figure 4.13 the original matrix has 10 live of 16 cells. The configuration leading to it with the fewest live cells, highlighted in the figure, has only 5/16 live cells. The highest count of live cells is 11, which can also be seen as 5/16 dead cells.

The average number of cells in the predecessors in this tree is 9.181. Figure 4.14 shows a histogram of the number of different live cell counts observed in the tree from Figure 4.13. It is of importance to note that the histogram seems not to be a normal distribution. From the available data it seems that the number of live cells in the matrices do not obey any regulations, and therefore, is not predictable.

4.3.1 Measure to decrease complexity on large matrices

Because the goal is to compress data, the algorithm needs to be able to handle bigger matrices. As has been seen, a binary matrix of size 6 (the largest backtraced so far) contains only 6 * 6 = 36 bits or 36 / 8 = 4 + 1/2 byte. In order to accomplish handling bigger matrix sizes the matrix must be split into smaller parts that can be solved directly in order to reduce the overall complexity.

In order to test if the theory of splitting bigger matrices into smaller for faster solution holds a proof-of-concept was developed. First a 6x6 matrix was selected and backtraced directly. The matrix is given in Figure 4.15 and took 114 minutes to backtrace.

The matrix was then split into four 3x3 areas as shown in Figure 4.16. Each of these matrices where altered so that the borders that face each other was set as "don't care" cells, Figure 4.17 shows which cells of the North-West corner became "don't cares".

The "Don't care" cells affect other cells as normal, but their requirements are ignored by the algorithm. They are also set as dead if they were alive.

By then solving all the four 3x3 areas directly, all their solutions were recorded. By permutating the recombination of all the solutions the solutions for the original 6x6 matrix was sought.

Unfortunately, because the South-West corner was completely empty (no live cells) the algorithm did not return any solutions. This is because there is no Colors to iterate over, hence no Perm Maps being created or solved. It was therefore decided to increase the size of the South-West corner to 4x4.

By permutation and recombination of the solutions of the 4x4 and the three 3x3 matrices, 3/4 of the previous configurations for the original 6x6 matrix
Figure 4.13: A full backtrace of a 4x4 configuration with rule B1358/S02467.
was found. Figure 4.18 shows the solutions that were found, along with the last solution that could not be found.

The time to solve each 3x3 matrix was less than a second, while the 4x4 took 3 seconds. The recombination process took 30 minutes.

Looking at the total time of 30 minutes compared with the original 2 hours the new timing is a positive result. The approach did however not yield all the previous configurations.

4.4 CA rules

In order to find out which rule is optimal for compressing data we have tested all the rules on matrixes on size 3 to find out which rule most often have a previous configuration (have the least gardens of eden). To test this we used to colour algorithm to test each rule on 1000 random matrices and check how many of the matrices had a solution for the given rule. This is an indication of how many gardens of Eden exist for the rule in 3x3 matrices, the result for each rule is given in a percentage of how many had a solution.

The rule that was found to be the best for size 3 is ‘B7531/S76420’ with 91%. 4.19 Shows the 10 best rules in this test. A longer list can be seen in appendix section 7.1.
Figure 4.15: 6x6 matrix backtraced in Game of Life (B3/S23) using the Colouring algorithm. The center cell is the origin and has 4 previous configurations leading to it in 1 generation.
Figure 4.16: The 6x6 matrix split into four 3x3 areas

Figure 4.17: 3x3 North-West corner showing "don’t cares" as D

Figure 4.18: The solutions for the 6x6 matrix that were found by the split approach. The grayed out (lower right) solution could not be found
### 4.4. CA RULES

<table>
<thead>
<tr>
<th>Rule</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>B531/S420 =&gt; 85%</td>
<td></td>
</tr>
<tr>
<td>B7531/S420 =&gt; 85%</td>
<td></td>
</tr>
<tr>
<td>B531/S7420 =&gt; 87%</td>
<td></td>
</tr>
<tr>
<td>B6531/S6420 =&gt; 87%</td>
<td></td>
</tr>
<tr>
<td>B7531/S7420 =&gt; 87%</td>
<td></td>
</tr>
<tr>
<td>B531/S6420 =&gt; 89%</td>
<td></td>
</tr>
<tr>
<td>B6531/S76420 =&gt; 89%</td>
<td></td>
</tr>
<tr>
<td>B7531/S6420 =&gt; 89%</td>
<td></td>
</tr>
<tr>
<td>B531/S76420 =&gt; 91%</td>
<td></td>
</tr>
<tr>
<td>B7531/S76420 =&gt; 91%</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 4.19**: The 10 best rules selected by the test
Chapter 5

Discussion and Future Work

5.1 Discussion

5.1.1 Reversible Cellular Automata

Reversible Cellular Automata was implemented as both Second-Order and Block Cellular Automata with success. The algorithms were able to run in both directions taking input of varying size. Both algorithms were tested by attempting to compress the new configuration at each step.

Although the reversible Cellular Automata implementation was able to go both forth and back in time it did not compress the data. A measure was implemented to lower the threshold by allowing a triplet, storing two configurations of second-order ca with a 1 generation gap, may have worked partially. But ultimately did not allow for compression either.

The implementation of the Reversible Cellular Automata only considered Block Cellular Automata and Second-Order Cellular Automata in infinite space. In infinite space the result could be different. The thesis has however proven that Cellular Automata must always enter a loop in finite space.

As a closing note on Reversible CA for Data Compression it is not clear how running such a model on some data affects its entropy. This is suggested for further study.

5.1.2 Genetic Algorithm

The Genetic Algorithm was implemented to search the CA space for configurations that lead to some given configuration. Two crossover functions and two mutation functions were implemented. The CA was able to perform a search in the CA space, leading to results of up to 95% similar to the original.

The genetic algorithm was able to find an answer for some 2x2 and 3x3 matrices, but in most cases it found a local optimum where the best answer is about 90% the same as the global maximum (a correct answer). Several
5.1. DISCUSSION

tweaks of the population size, the mutation rate and the mutation function were attempted but it only helped marginally.

The problem with the GA was that it would in most cases get stuck with exploiting the current best answer instead of exploring for the global optimum. Even when changing the balance greatly toward exploration it would finally get stuck on exploiting something that didn’t lead to a real answer. Some of the reason for this could be that there is not enough relation between CA configurations or that we could not produce mutation functions and recombination functions that found the relation.

The GA may however have application in lossy compression using CA, which could be studied further.

5.1.3 Coloring Algorithm

The colouring algorithm was designed, implemented and tested. The algorithm was able to identify the previous configurations of any configuration and rule it was given, when one existed. Sizes tested range from 2x2 to 6x6. On sizes smaller than 5x5 the algorithm will generally finish in milliseconds, while 6x6 matrices may take several hours.

Using the algorithm all binary CA rules in the Moore neighbourhood was tested on 3x3 matrices to rank them in order of how often Garden of Edens appear. Rules containing B8, B0 and rules with no birth were excluded as they seldom appear. The rule found to have the least amount of Gardens of Eden in Null Boundary 3x3 space was B1357/S02467, a list of the 1000 best can be found in appendix section 7.1.

Compressing actual file become very difficult however, this is mainly because of the rapid growth nature of the algorithm. So far the biggest matrices that have backtracked directly is 6x6, which takes from 160 seconds to 5 hours. Since each bit in a file would be treated as a cell, a small 20 KiB file would amass to a 405x405 matrix ($\sqrt{20 * 1024 * 8} = 405$). If the algorithm started backtracking such a matrix directly it wouldn’t finish before our sun goes cold (about 5 billion years).

In order to compress data, the algorithm needs to work on matrices bigger than what the it can handle in a timely manner. By splitting the matrix into smaller parts that can be solved quickly, then recombining all their possibilities to find the solutions the problem may become smaller.

As was shown in subsection 4.3.1 this may hold merit. But is not implemented by the algorithm.
5.2. FUTURE WORK ON CELLULAR AUTOMATON BASED DATA COMPRESSION

5.2 Future Work on Cellular Automaton based Data Compression

Optimizations and a real implementation of the idea of splitting the CA configuration into smaller parts and solving them independently could be considered for future study.

An important aspect of Cellular Automata that has not been considered in this thesis is multiple state CA, that is Cellular Automata with more than 2 states. Such CA could in theory pack more information per cell and thus more information in small matrices. The application of multiple state CA to data compression is therefore suggested as future work.
Chapter 6

Conclusion

The Genetic algorithm was tried and discarded as it could not find any previous configurations. Second-order cellular automata was discarded because storing a way back to the original would require storing 2 matrices by the same size as the original which did not result in smaller files in any of the tests. Block cellular Automata did not in any test make the matrix smaller.

A method to systematically search for previous configurations in 2 dimensional binary Cellular Automata was defined as the Colouring Algorithm and a proof-of-concept based on this definition was implemented and tested.

The algorithm works and it is better than a brute force approach, in all tests where a previous configuration was known to exist the algorithm found it. The algorithm cannot however directly achieve the goal of data compression because of its complexity growth. Although the original goal of data compression could not be achieved the colouring algorithm in itself is an achievement.

The bottom line about using Cellular Automata for lossless Data Compression is, however, that it seems very difficult.
Bibliography


Chapter 7

Appendix

7.1 CA rules tested on 3x3 matrices, 1000 best, rules with B8 excluded

<table>
<thead>
<tr>
<th>Rule</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>B6431/S7420 =&gt; 59%</td>
<td></td>
</tr>
<tr>
<td>B6431/S7620 =&gt; 59%</td>
<td></td>
</tr>
<tr>
<td>B6432/S =&gt; 59%</td>
<td></td>
</tr>
<tr>
<td>B6432/S741 =&gt; 59%</td>
<td></td>
</tr>
<tr>
<td>B6521/S430 =&gt; 59%</td>
<td></td>
</tr>
<tr>
<td>B6521/S5430 =&gt; 59%</td>
<td></td>
</tr>
<tr>
<td>B6521/S65430 =&gt; 59%</td>
<td></td>
</tr>
<tr>
<td>B6531/S2 =&gt; 59%</td>
<td></td>
</tr>
<tr>
<td>B6531/S431 =&gt; 59%</td>
<td></td>
</tr>
<tr>
<td>B6531/S6421 =&gt; 59%</td>
<td></td>
</tr>
<tr>
<td>B6531/S72 =&gt; 59%</td>
<td></td>
</tr>
<tr>
<td>B6531/S730 =&gt; 59%</td>
<td></td>
</tr>
<tr>
<td>B6531/S75421 =&gt; 59%</td>
<td></td>
</tr>
<tr>
<td>B6532/S43 =&gt; 59%</td>
<td></td>
</tr>
<tr>
<td>B6532/S740 =&gt; 59%</td>
<td></td>
</tr>
<tr>
<td>B6532/S764 =&gt; 59%</td>
<td></td>
</tr>
<tr>
<td>B6532/S76420 =&gt; 59%</td>
<td></td>
</tr>
<tr>
<td>B6542/S5310 =&gt; 59%</td>
<td></td>
</tr>
<tr>
<td>B6542/S6532 =&gt; 59%</td>
<td></td>
</tr>
<tr>
<td>B6542/S76410 =&gt; 59%</td>
<td></td>
</tr>
<tr>
<td>B65432/S =&gt; 59%</td>
<td></td>
</tr>
<tr>
<td>B654321/S74 =&gt; 59%</td>
<td></td>
</tr>
<tr>
<td>B721/S43 =&gt; 59%</td>
<td></td>
</tr>
<tr>
<td>B721/S543 =&gt; 59%</td>
<td></td>
</tr>
<tr>
<td>B721/S65430 =&gt; 59%</td>
<td></td>
</tr>
<tr>
<td>B721/S75430 =&gt; 59%</td>
<td></td>
</tr>
<tr>
<td>B742/S310 =&gt; 59%</td>
<td></td>
</tr>
<tr>
<td>B742/S531 =&gt; 59%</td>
<td></td>
</tr>
<tr>
<td>B742/S55430 =&gt; 59%</td>
<td></td>
</tr>
<tr>
<td>B742/S75430 =&gt; 59%</td>
<td></td>
</tr>
<tr>
<td>B742/S7543 =&gt; 59%</td>
<td></td>
</tr>
</tbody>
</table>
### 7.1. CA RULES TESTED ON 3X3 MATRICES, 1000 BEST, RULES WITH B8 EXCLUDED

<table>
<thead>
<tr>
<th>Rule</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>B74321/S 60</td>
<td>59%</td>
</tr>
<tr>
<td>B74321/S6530</td>
<td>59%</td>
</tr>
<tr>
<td>B74321/S654</td>
<td>59%</td>
</tr>
<tr>
<td>B74321/S6410</td>
<td>59%</td>
</tr>
<tr>
<td>B74321/S67410</td>
<td>59%</td>
</tr>
<tr>
<td>B7521/S642</td>
<td>59%</td>
</tr>
<tr>
<td>B7521/S7430</td>
<td>59%</td>
</tr>
<tr>
<td>B7521/S7542</td>
<td>59%</td>
</tr>
<tr>
<td>B7521/S7642</td>
<td>59%</td>
</tr>
<tr>
<td>B7531/S24</td>
<td>59%</td>
</tr>
<tr>
<td>B7531/S72</td>
<td>59%</td>
</tr>
<tr>
<td>B7531/S4</td>
<td>59%</td>
</tr>
<tr>
<td>B7531/S64</td>
<td>59%</td>
</tr>
<tr>
<td>B7531/S66</td>
<td>59%</td>
</tr>
<tr>
<td>B7531/S740</td>
<td>59%</td>
</tr>
<tr>
<td>B7531/S7542</td>
<td>59%</td>
</tr>
<tr>
<td>B7531/S7642</td>
<td>59%</td>
</tr>
<tr>
<td>B7532/S6410</td>
<td>59%</td>
</tr>
<tr>
<td>B7532/S67410</td>
<td>59%</td>
</tr>
<tr>
<td>B7532/S7430</td>
<td>59%</td>
</tr>
<tr>
<td>B7532/S75430</td>
<td>59%</td>
</tr>
<tr>
<td>B7532/S76430</td>
<td>59%</td>
</tr>
<tr>
<td>B7532/S765430</td>
<td>59%</td>
</tr>
<tr>
<td>B7532/S765431</td>
<td>59%</td>
</tr>
<tr>
<td>B7533/S6410</td>
<td>59%</td>
</tr>
<tr>
<td>B7533/S67410</td>
<td>59%</td>
</tr>
<tr>
<td>B7533/S7430</td>
<td>59%</td>
</tr>
<tr>
<td>B7533/S75430</td>
<td>59%</td>
</tr>
<tr>
<td>B7533/S76430</td>
<td>59%</td>
</tr>
<tr>
<td>B7533/S765430</td>
<td>59%</td>
</tr>
<tr>
<td>B7533/S765431</td>
<td>59%</td>
</tr>
<tr>
<td>B75342/S6532</td>
<td>60%</td>
</tr>
<tr>
<td>B75342/S6542</td>
<td>60%</td>
</tr>
<tr>
<td>B75342/S75430</td>
<td>60%</td>
</tr>
<tr>
<td>B75342/S76430</td>
<td>60%</td>
</tr>
<tr>
<td>B75342/S765430</td>
<td>60%</td>
</tr>
<tr>
<td>B75342/S765431</td>
<td>60%</td>
</tr>
<tr>
<td>B75351/S6432</td>
<td>60%</td>
</tr>
<tr>
<td>B75351/S65432</td>
<td>60%</td>
</tr>
<tr>
<td>B75351/S75432</td>
<td>60%</td>
</tr>
<tr>
<td>B75351/S76432</td>
<td>60%</td>
</tr>
<tr>
<td>B75352/S643</td>
<td>60%</td>
</tr>
<tr>
<td>B75352/S6543</td>
<td>60%</td>
</tr>
<tr>
<td>B75352/S7543</td>
<td>60%</td>
</tr>
<tr>
<td>B75352/S7643</td>
<td>60%</td>
</tr>
<tr>
<td>B75362/S653</td>
<td>60%</td>
</tr>
<tr>
<td>B75362/S6543</td>
<td>60%</td>
</tr>
<tr>
<td>B75362/S7543</td>
<td>60%</td>
</tr>
<tr>
<td>B75362/S7643</td>
<td>60%</td>
</tr>
<tr>
<td>B75362/S76543</td>
<td>60%</td>
</tr>
<tr>
<td>B75362/S765431</td>
<td>60%</td>
</tr>
<tr>
<td>B75371/S654</td>
<td>60%</td>
</tr>
<tr>
<td>B75371/S6543</td>
<td>60%</td>
</tr>
<tr>
<td>B75371/S7543</td>
<td>60%</td>
</tr>
<tr>
<td>B75371/S7643</td>
<td>60%</td>
</tr>
<tr>
<td>B75372/S654</td>
<td>60%</td>
</tr>
<tr>
<td>B75372/S6543</td>
<td>60%</td>
</tr>
<tr>
<td>B75372/S7543</td>
<td>60%</td>
</tr>
<tr>
<td>B75372/S7643</td>
<td>60%</td>
</tr>
</tbody>
</table>
### 7.1. CA RULES TESTED ON 3X3 MATRICES, 1000 BEST, RULES WITH B8 EXCLUDED

<table>
<thead>
<tr>
<th>Rule</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>B3521/S6530 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B5321/S65420 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B35/S765420 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B3521/S764 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B3521/S765320 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B3521/S76532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B531/S76532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B3521/S765320 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B5321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B631/S642 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B631/S6420 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B631/S6532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B631/S65320 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S653 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S6530 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S6532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65320 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S6543 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65431 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B63/S65321/S65430 =&gt; 60%</td>
<td></td>
</tr>
</tbody>
</table>
7.1. CA RULES TESTED ON 3X3 MATRICES, 1000 BEST, RULES WITH B8 EXCLUDED

<table>
<thead>
<tr>
<th>Rule</th>
<th>Success Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>B7532/S65420 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B753/S65420 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S42 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S731 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S742 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S75320 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S6532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S765420 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S75320 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S6532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S765430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S75320 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S6532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S765430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S75320 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S6532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S765430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S75320 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S6532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S765430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S75320 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S6532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S765430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S75320 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S6532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S765430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S75320 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S6532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S765430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S75320 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S6532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S765430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S75320 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S6532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S765430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S75320 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S6532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S765430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S75320 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S6532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S765430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S75320 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S6532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S765430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S75320 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S6532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S765430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S75320 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S6532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S765430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S75320 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S6532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S765430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S75320 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S6532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S765430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S75320 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S6532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S765430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S75320 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S6532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S765430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S75320 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S6532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S765430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S75320 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S6532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S765430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S75320 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S6532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S765430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S75320 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S6532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S765430 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S75320 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S7532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S6532 =&gt; 60%</td>
<td></td>
</tr>
<tr>
<td>B7542/S765430 =&gt; 60%</td>
<td></td>
</tr>
</tbody>
</table>
7.1. CA RULES TESTED ON 3X3 MATRICES, 1000 BEST, RULES WITH B8 EXCLUDED

<table>
<thead>
<tr>
<th>Rule</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>B6521/S643</td>
<td>61%</td>
</tr>
<tr>
<td>B521/S65430</td>
<td>61%</td>
</tr>
<tr>
<td>B6521/S7640</td>
<td>61%</td>
</tr>
<tr>
<td>B65321/S30</td>
<td>61%</td>
</tr>
<tr>
<td>B65321/S5630</td>
<td>61%</td>
</tr>
<tr>
<td>B65321/S730</td>
<td>61%</td>
</tr>
<tr>
<td>B65321/S7640</td>
<td>61%</td>
</tr>
<tr>
<td>B65321/S76530</td>
<td>61%</td>
</tr>
<tr>
<td>B65421/S43</td>
<td>61%</td>
</tr>
<tr>
<td>B65421/S5531</td>
<td>61%</td>
</tr>
<tr>
<td>B65421/S55320</td>
<td>61%</td>
</tr>
<tr>
<td>B6542/S7631</td>
<td>61%</td>
</tr>
<tr>
<td>B6542/S76350</td>
<td>61%</td>
</tr>
<tr>
<td>B65432/S530</td>
<td>61%</td>
</tr>
<tr>
<td>B65432/S740</td>
<td>61%</td>
</tr>
<tr>
<td>B65432/S7450</td>
<td>61%</td>
</tr>
<tr>
<td>B65432/S75430</td>
<td>61%</td>
</tr>
<tr>
<td>B65432/S76430</td>
<td>61%</td>
</tr>
<tr>
<td>B65432/S7640</td>
<td>61%</td>
</tr>
<tr>
<td>B65432/S76530</td>
<td>61%</td>
</tr>
<tr>
<td>B731/S7642</td>
<td>61%</td>
</tr>
<tr>
<td>B7321/S763</td>
<td>61%</td>
</tr>
<tr>
<td>B7421/S64310</td>
<td>61%</td>
</tr>
<tr>
<td>B7421/S6543</td>
<td>61%</td>
</tr>
<tr>
<td>B7421/S764310</td>
<td>61%</td>
</tr>
<tr>
<td>B7421/S76543</td>
<td>61%</td>
</tr>
<tr>
<td>B75321/S7620</td>
<td>61%</td>
</tr>
<tr>
<td>B75321/S76430</td>
<td>61%</td>
</tr>
<tr>
<td>B75321/S7640</td>
<td>61%</td>
</tr>
<tr>
<td>B75321/S76530</td>
<td>61%</td>
</tr>
<tr>
<td>B75421/S31</td>
<td>61%</td>
</tr>
<tr>
<td>B75421/S6531</td>
<td>61%</td>
</tr>
<tr>
<td>B75421/S76531</td>
<td>61%</td>
</tr>
<tr>
<td>B7542/S5320</td>
<td>61%</td>
</tr>
<tr>
<td>B75431/S52</td>
<td>61%</td>
</tr>
<tr>
<td>B75431/S62</td>
<td>61%</td>
</tr>
<tr>
<td>B75431/S752</td>
<td>61%</td>
</tr>
<tr>
<td>B75431/S7652</td>
<td>61%</td>
</tr>
<tr>
<td>B754321/S4</td>
<td>61%</td>
</tr>
<tr>
<td>B754321/S730</td>
<td>61%</td>
</tr>
<tr>
<td>B754321/S7410</td>
<td>61%</td>
</tr>
<tr>
<td>B754321/S7540</td>
<td>61%</td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S7540</td>
<td>61%</td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S754 =&gt; 61%</td>
<td></td>
</tr>
<tr>
<td>B754321/S75 =&gt; 61%</td>
<td></td>
</tr>
</tbody>
</table>
### 7.1. CA RULES TESTED ON 3x3 MATRICES, 1000 BEST, RULES WITH B8 EXCLUDED

<table>
<thead>
<tr>
<th>Rule</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>B42/S75320 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B431/S620 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B431/S720 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B431/S7530 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B431/S7640 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B432/S6430 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B432/S7421 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B432/S76421 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B5321/S530 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B5321/S563 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B5321/S7530 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B532/S756420 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B5421/S43 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B542/S5520 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B542/S5932 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B542/S731 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B542/S7532 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B5431/S652 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B54321/S55 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B54321/S764 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B5432/S65410 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B5432/S7410 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B5432/S765410 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B621/S7643 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B631/S55020 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B631/S75320 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B641/S7442 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B6421/S4340 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B6421/S7430 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B642/S3510 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B642/S65310 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B642/S65430 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B642/S75310 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B642/S75320 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B642/S76532 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B64321/S66 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B64321/S77 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B64321/S786 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B6432/S55410 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B6432/S6410 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B6432/S65410 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B6432/S76410 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B6521/S5542 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B6521/S6430 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B6521/S75320 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B6531/S57520 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B65321/S6640 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B65321/S6643 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B65321/S66543 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B65321/S76543 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B65421/S65410 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B65421/S7430 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B6542/S3510 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B6542/S55320 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B6542/S56430 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B6542/S65320 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B6542/S65430 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B65431/S75420 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B654321/S752 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B721/S76430 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B7421/S65430 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B742/S55320 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B742/S65320 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B742/S665430 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B742/S76430 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B74321/S66 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B7432/S4510 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B7531/S56421 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B7531/S7421 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B7531/S75720 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B7531/S76520 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B75321/S6654 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B75321/S7654 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B75321/S76542 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B75321/S765430 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B75421/S5731 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B754321/S5731 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B75432/S4510 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B75432/S665410 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B75432/S76410 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B75432/S765410 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B7631/S55320 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B7631/S55420 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B7631/S75320 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B7631/S76422 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B76321/S653 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B76421/S531 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B76421/S7531 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B7642/S3510 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B7642/S45410 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B7642/S665320 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B7642/S665330 ⇒ 62%</td>
<td></td>
</tr>
<tr>
<td>B7642/S75310 ⇒ 62%</td>
<td></td>
</tr>
</tbody>
</table>
### 7.1. CA RULES TESTED ON 3X3 MATRICES, 1000 BEST, RULES WITH B8 EXCLUDED

<table>
<thead>
<tr>
<th>Rule</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>B7642/S65320 =&gt; 62%</td>
<td>510</td>
</tr>
<tr>
<td>B7642/S65322 =&gt; 62%</td>
<td>512</td>
</tr>
<tr>
<td>B7642/S7430 =&gt; 62%</td>
<td>514</td>
</tr>
<tr>
<td>B7642/S75520 =&gt; 62%</td>
<td>516</td>
</tr>
<tr>
<td>B7642/S75532 =&gt; 62%</td>
<td>518</td>
</tr>
<tr>
<td>B76432/S510 =&gt; 62%</td>
<td>520</td>
</tr>
<tr>
<td>B76432/S5320 =&gt; 62%</td>
<td>522</td>
</tr>
<tr>
<td>B76432/S5520 =&gt; 62%</td>
<td>524</td>
</tr>
<tr>
<td>B76432/S631 =&gt; 63%</td>
<td>526</td>
</tr>
<tr>
<td>B76432/S65320 =&gt; 63%</td>
<td>528</td>
</tr>
<tr>
<td>B76432/S7430 =&gt; 63%</td>
<td>530</td>
</tr>
<tr>
<td>B76432/S75532 =&gt; 63%</td>
<td>532</td>
</tr>
<tr>
<td>B76432/S755430 =&gt; 63%</td>
<td>534</td>
</tr>
<tr>
<td>B76432/S765320 =&gt; 63%</td>
<td>536</td>
</tr>
<tr>
<td>B76432/S765430 =&gt; 63%</td>
<td>538</td>
</tr>
<tr>
<td>B76432/S7654320 =&gt; 63%</td>
<td>540</td>
</tr>
<tr>
<td>B76432/S765432 =&gt; 63%</td>
<td>542</td>
</tr>
<tr>
<td>B76432/S7654320 =&gt; 63%</td>
<td>544</td>
</tr>
<tr>
<td>B76432/S765432 =&gt; 63%</td>
<td>546</td>
</tr>
<tr>
<td>B76432/S7654320 =&gt; 63%</td>
<td>548</td>
</tr>
<tr>
<td>B76432/S765432 =&gt; 63%</td>
<td>550</td>
</tr>
<tr>
<td>B76432/S8 =&gt; 63%</td>
<td>552</td>
</tr>
<tr>
<td>B76432/S9 =&gt; 63%</td>
<td>554</td>
</tr>
<tr>
<td>B76432/S10 =&gt; 63%</td>
<td>556</td>
</tr>
<tr>
<td>B76432/S11 =&gt; 63%</td>
<td>558</td>
</tr>
<tr>
<td>B76432/S12 =&gt; 63%</td>
<td>560</td>
</tr>
<tr>
<td>B76432/S13 =&gt; 63%</td>
<td>562</td>
</tr>
<tr>
<td>B76432/S14 =&gt; 63%</td>
<td>564</td>
</tr>
<tr>
<td>B76432/S15 =&gt; 63%</td>
<td>566</td>
</tr>
<tr>
<td>B76432/S16 =&gt; 63%</td>
<td>568</td>
</tr>
<tr>
<td>B76432/S17 =&gt; 63%</td>
<td>570</td>
</tr>
<tr>
<td>B76432/S18 =&gt; 63%</td>
<td>572</td>
</tr>
<tr>
<td>B76432/S19 =&gt; 63%</td>
<td>574</td>
</tr>
<tr>
<td>B76432/S20 =&gt; 63%</td>
<td>576</td>
</tr>
<tr>
<td>B76432/S21 =&gt; 63%</td>
<td>578</td>
</tr>
<tr>
<td>B76432/S22 =&gt; 63%</td>
<td>580</td>
</tr>
<tr>
<td>B76432/S23 =&gt; 63%</td>
<td>582</td>
</tr>
<tr>
<td>B76432/S24 =&gt; 63%</td>
<td>584</td>
</tr>
<tr>
<td>B76432/S25 =&gt; 63%</td>
<td>586</td>
</tr>
<tr>
<td>B76432/S26 =&gt; 63%</td>
<td>588</td>
</tr>
<tr>
<td>B76432/S27 =&gt; 63%</td>
<td>590</td>
</tr>
<tr>
<td>B76432/S28 =&gt; 63%</td>
<td>592</td>
</tr>
<tr>
<td>B76432/S29 =&gt; 63%</td>
<td>594</td>
</tr>
<tr>
<td>B76432/S30 =&gt; 63%</td>
<td>596</td>
</tr>
<tr>
<td>B76432/S31 =&gt; 63%</td>
<td>598</td>
</tr>
</tbody>
</table>
7.1. CA RULES TESTED ON 3X3 MATRICES, 1000 BEST, RULES WITH B8 EXCLUDED

<table>
<thead>
<tr>
<th>Rule</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>B7432/S5410</td>
<td>63%</td>
</tr>
<tr>
<td>B4321/S40</td>
<td>64%</td>
</tr>
<tr>
<td>B4321/S6540</td>
<td>64%</td>
</tr>
<tr>
<td>B4321/S570</td>
<td>64%</td>
</tr>
<tr>
<td>B4321/S76540</td>
<td>64%</td>
</tr>
<tr>
<td>B531/S5520</td>
<td>64%</td>
</tr>
<tr>
<td>B531/S5650</td>
<td>64%</td>
</tr>
<tr>
<td>B531/S7520</td>
<td>64%</td>
</tr>
<tr>
<td>B531/S7620</td>
<td>64%</td>
</tr>
<tr>
<td>B5321/S6534</td>
<td>64%</td>
</tr>
<tr>
<td>B5321/S73</td>
<td>64%</td>
</tr>
<tr>
<td>B5321/S753</td>
<td>64%</td>
</tr>
<tr>
<td>B5321/S7643</td>
<td>64%</td>
</tr>
<tr>
<td>B5321/S8765</td>
<td>64%</td>
</tr>
<tr>
<td>B5321/S76540</td>
<td>64%</td>
</tr>
<tr>
<td>B5321/S76543</td>
<td>64%</td>
</tr>
<tr>
<td>B542/S31</td>
<td>64%</td>
</tr>
<tr>
<td>B5431/S420</td>
<td>64%</td>
</tr>
<tr>
<td>B5431/S65420</td>
<td>64%</td>
</tr>
<tr>
<td>B5431/S5731</td>
<td>64%</td>
</tr>
<tr>
<td>B5431/S58420</td>
<td>64%</td>
</tr>
<tr>
<td>B54321/S6540</td>
<td>64%</td>
</tr>
<tr>
<td>B54321/S7540</td>
<td>64%</td>
</tr>
<tr>
<td>B54321/S7640</td>
<td>64%</td>
</tr>
<tr>
<td>B5521/S643</td>
<td>64%</td>
</tr>
<tr>
<td>B561/S643</td>
<td>64%</td>
</tr>
<tr>
<td>B561/S65420</td>
<td>64%</td>
</tr>
<tr>
<td>B561/S75420</td>
<td>64%</td>
</tr>
<tr>
<td>B6321/S73</td>
<td>64%</td>
</tr>
<tr>
<td>B6321/S753</td>
<td>64%</td>
</tr>
<tr>
<td>B6321/S7653</td>
<td>64%</td>
</tr>
<tr>
<td>B6321/S76540</td>
<td>64%</td>
</tr>
<tr>
<td>B6321/S76543</td>
<td>64%</td>
</tr>
<tr>
<td>B642/S6430</td>
<td>64%</td>
</tr>
<tr>
<td>B642/S65320</td>
<td>64%</td>
</tr>
<tr>
<td>B642/S65332</td>
<td>64%</td>
</tr>
<tr>
<td>B6432/S410</td>
<td>64%</td>
</tr>
<tr>
<td>B6432/S7410</td>
<td>64%</td>
</tr>
<tr>
<td>B6531/S643</td>
<td>64%</td>
</tr>
<tr>
<td>B6531/S6531</td>
<td>64%</td>
</tr>
<tr>
<td>B6531/S7543</td>
<td>64%</td>
</tr>
<tr>
<td>B65321/S640</td>
<td>64%</td>
</tr>
<tr>
<td>B65321/S7640</td>
<td>64%</td>
</tr>
<tr>
<td>B65321/S7643</td>
<td>64%</td>
</tr>
<tr>
<td>B65321/S7653</td>
<td>64%</td>
</tr>
<tr>
<td>B65321/S76540</td>
<td>64%</td>
</tr>
<tr>
<td>B65321/S76543</td>
<td>64%</td>
</tr>
<tr>
<td>B6621/S643</td>
<td>64%</td>
</tr>
<tr>
<td>B6621/S6543</td>
<td>64%</td>
</tr>
<tr>
<td>B6621/S7543</td>
<td>64%</td>
</tr>
<tr>
<td>B6621/S7654</td>
<td>64%</td>
</tr>
<tr>
<td>B6721/S643</td>
<td>64%</td>
</tr>
<tr>
<td>B6721/S6543</td>
<td>64%</td>
</tr>
<tr>
<td>B7421/S643</td>
<td>64%</td>
</tr>
<tr>
<td>B7421/S7543</td>
<td>64%</td>
</tr>
<tr>
<td>B74321/S643</td>
<td>64%</td>
</tr>
<tr>
<td>B74321/S6540</td>
<td>64%</td>
</tr>
<tr>
<td>B74321/S750</td>
<td>64%</td>
</tr>
<tr>
<td>B74321/S76540</td>
<td>64%</td>
</tr>
<tr>
<td>B75321/S553</td>
<td>64%</td>
</tr>
<tr>
<td>B75321/S563</td>
<td>64%</td>
</tr>
<tr>
<td>B75321/S643</td>
<td>64%</td>
</tr>
<tr>
<td>B75321/S653</td>
<td>64%</td>
</tr>
<tr>
<td>B75321/S753</td>
<td>64%</td>
</tr>
<tr>
<td>B75321/S7653</td>
<td>64%</td>
</tr>
<tr>
<td>B75321/S76540</td>
<td>64%</td>
</tr>
<tr>
<td>B75421/S7531</td>
<td>64%</td>
</tr>
<tr>
<td>B7542/S531</td>
<td>64%</td>
</tr>
<tr>
<td>B75431/S642</td>
<td>64%</td>
</tr>
<tr>
<td>B75431/S7642</td>
<td>64%</td>
</tr>
<tr>
<td>B75431/S76420</td>
<td>64%</td>
</tr>
<tr>
<td>B754321/S530</td>
<td>64%</td>
</tr>
<tr>
<td>B754321/S55</td>
<td>64%</td>
</tr>
<tr>
<td>B754321/S755</td>
<td>64%</td>
</tr>
<tr>
<td>B754321/S757</td>
<td>64%</td>
</tr>
<tr>
<td>B754321/S7643</td>
<td>64%</td>
</tr>
<tr>
<td>B7621/S7643</td>
<td>64%</td>
</tr>
<tr>
<td>B7621/S7653</td>
<td>64%</td>
</tr>
<tr>
<td>B7621/S76540</td>
<td>64%</td>
</tr>
<tr>
<td>B7621/S76543</td>
<td>64%</td>
</tr>
<tr>
<td>B76421/S7410</td>
<td>64%</td>
</tr>
<tr>
<td>B76421/S743</td>
<td>64%</td>
</tr>
<tr>
<td>B76421/S7543</td>
<td>64%</td>
</tr>
<tr>
<td>B76421/S7630</td>
<td>64%</td>
</tr>
<tr>
<td>B76421/S7640</td>
<td>64%</td>
</tr>
<tr>
<td>B76421/S7643</td>
<td>64%</td>
</tr>
<tr>
<td>B76421/S7653</td>
<td>64%</td>
</tr>
<tr>
<td>B76421/S76540</td>
<td>64%</td>
</tr>
<tr>
<td>B76421/S76543</td>
<td>64%</td>
</tr>
<tr>
<td>B76421/S76543</td>
<td>64%</td>
</tr>
<tr>
<td>B76421/S76543</td>
<td>64%</td>
</tr>
<tr>
<td>B76421/S76543</td>
<td>64%</td>
</tr>
<tr>
<td>B76421/S76543</td>
<td>64%</td>
</tr>
<tr>
<td>B76421/S76543</td>
<td>64%</td>
</tr>
<tr>
<td>B76421/S76543</td>
<td>64%</td>
</tr>
<tr>
<td>B76421/S76543</td>
<td>64%</td>
</tr>
<tr>
<td>B76421/S76543</td>
<td>64%</td>
</tr>
<tr>
<td>B76421/S76543</td>
<td>64%</td>
</tr>
<tr>
<td>B76421/S76543</td>
<td>64%</td>
</tr>
<tr>
<td>B76421/S76543</td>
<td>64%</td>
</tr>
</tbody>
</table>
### 7.1. CA RULES TESTED ON 3X3 MATRICES, 1000 BEST, RULES WITH B8 EXCLUDED

<table>
<thead>
<tr>
<th>Rule</th>
<th>Effectiveness</th>
</tr>
</thead>
<tbody>
<tr>
<td>B5/321/S7653</td>
<td>≈ 65%</td>
</tr>
<tr>
<td>B5431/S65420</td>
<td>≈ 65%</td>
</tr>
<tr>
<td>B5431/S7420</td>
<td>≈ 65%</td>
</tr>
<tr>
<td>B731/S5420</td>
<td>≈ 65%</td>
</tr>
<tr>
<td>B7321/S753</td>
<td>≈ 65%</td>
</tr>
<tr>
<td>B7421/S6431</td>
<td>≈ 65%</td>
</tr>
<tr>
<td>B7421/S7543</td>
<td>≈ 65%</td>
</tr>
<tr>
<td>B7421/S76431</td>
<td>≈ 65%</td>
</tr>
<tr>
<td>B742/S6430</td>
<td>≈ 65%</td>
</tr>
<tr>
<td>B7531/S7620</td>
<td>≈ 65%</td>
</tr>
<tr>
<td>B75321/S543</td>
<td>≈ 65%</td>
</tr>
<tr>
<td>B75321/S653</td>
<td>≈ 65%</td>
</tr>
<tr>
<td>B75321/S6543</td>
<td>≈ 65%</td>
</tr>
<tr>
<td>B75321/S7534</td>
<td>≈ 65%</td>
</tr>
<tr>
<td>B75321/S7563</td>
<td>≈ 65%</td>
</tr>
<tr>
<td>B75421/S5531</td>
<td>≈ 65%</td>
</tr>
<tr>
<td>B7542/S6531</td>
<td>≈ 65%</td>
</tr>
<tr>
<td>B7543/S7420</td>
<td>≈ 65%</td>
</tr>
<tr>
<td>B7621/S6430</td>
<td>≈ 65%</td>
</tr>
<tr>
<td>B763/S65420</td>
<td>≈ 65%</td>
</tr>
<tr>
<td>B7631/S75420</td>
<td>≈ 65%</td>
</tr>
<tr>
<td>B7631/S75631</td>
<td>≈ 65%</td>
</tr>
<tr>
<td>B7642/S5430</td>
<td>≈ 65%</td>
</tr>
<tr>
<td>B7642/S543</td>
<td>≈ 65%</td>
</tr>
<tr>
<td>B7642/S76430</td>
<td>≈ 65%</td>
</tr>
<tr>
<td>B76431/S6540</td>
<td>≈ 65%</td>
</tr>
<tr>
<td>B76431/S7640</td>
<td>≈ 65%</td>
</tr>
<tr>
<td>B31/S5420</td>
<td>≈ 66%</td>
</tr>
<tr>
<td>B4321/S740</td>
<td>≈ 66%</td>
</tr>
<tr>
<td>B531/S520</td>
<td>≈ 66%</td>
</tr>
<tr>
<td>B531/S642</td>
<td>≈ 66%</td>
</tr>
<tr>
<td>B531/S6620</td>
<td>≈ 66%</td>
</tr>
<tr>
<td>B531/S7520</td>
<td>≈ 66%</td>
</tr>
<tr>
<td>B5421/S643</td>
<td>≈ 66%</td>
</tr>
<tr>
<td>B5421/S7643</td>
<td>≈ 66%</td>
</tr>
<tr>
<td>B6531/S65420</td>
<td>≈ 66%</td>
</tr>
<tr>
<td>B642/S7631</td>
<td>≈ 66%</td>
</tr>
<tr>
<td>B642/S76430</td>
<td>≈ 66%</td>
</tr>
<tr>
<td>B64321/S6540</td>
<td>≈ 66%</td>
</tr>
<tr>
<td>B64321/S7640</td>
<td>≈ 66%</td>
</tr>
<tr>
<td>B7421/S5430</td>
<td>≈ 66%</td>
</tr>
<tr>
<td>B7421/S76430</td>
<td>≈ 66%</td>
</tr>
<tr>
<td>B74321/S560</td>
<td>≈ 66%</td>
</tr>
<tr>
<td>B74321/S740</td>
<td>≈ 66%</td>
</tr>
<tr>
<td>B7531/S542</td>
<td>≈ 66%</td>
</tr>
<tr>
<td>B75421/S643</td>
<td>≈ 66%</td>
</tr>
<tr>
<td>B75421/S7643</td>
<td>≈ 66%</td>
</tr>
<tr>
<td>B7542/S76531</td>
<td>≈ 66%</td>
</tr>
<tr>
<td>B75431/S55420</td>
<td>≈ 66%</td>
</tr>
<tr>
<td>B75431/S65420</td>
<td>≈ 66%</td>
</tr>
<tr>
<td>B7621/S6435</td>
<td>≈ 66%</td>
</tr>
<tr>
<td>B7642/S6531</td>
<td>≈ 66%</td>
</tr>
<tr>
<td>B7642/S76531</td>
<td>≈ 66%</td>
</tr>
<tr>
<td>B7642/S76430</td>
<td>≈ 66%</td>
</tr>
<tr>
<td>B31/S75420</td>
<td>≈ 67%</td>
</tr>
<tr>
<td>B4321/S5540</td>
<td>≈ 67%</td>
</tr>
<tr>
<td>B531/S6542</td>
<td>≈ 67%</td>
</tr>
<tr>
<td>B531/S6620</td>
<td>≈ 67%</td>
</tr>
<tr>
<td>B531/S7542</td>
<td>≈ 67%</td>
</tr>
<tr>
<td>B6521/S5531</td>
<td>≈ 67%</td>
</tr>
<tr>
<td>B6521/S7531</td>
<td>≈ 67%</td>
</tr>
<tr>
<td>B6531/S65420</td>
<td>≈ 67%</td>
</tr>
<tr>
<td>B683/S6531</td>
<td>≈ 67%</td>
</tr>
<tr>
<td>B65431/S65420</td>
<td>≈ 67%</td>
</tr>
<tr>
<td>B7531/S65420</td>
<td>≈ 67%</td>
</tr>
<tr>
<td>B75321/S6531</td>
<td>≈ 67%</td>
</tr>
<tr>
<td>B75321/S76420</td>
<td>≈ 67%</td>
</tr>
<tr>
<td>B75321/S76531</td>
<td>≈ 67%</td>
</tr>
<tr>
<td>B75321/S76430</td>
<td>≈ 67%</td>
</tr>
<tr>
<td>B75321/S765420</td>
<td>≈ 67%</td>
</tr>
<tr>
<td>B75321/S76420</td>
<td>≈ 67%</td>
</tr>
<tr>
<td>B75321/S765420</td>
<td>≈ 67%</td>
</tr>
<tr>
<td>B75321/S76430</td>
<td>≈ 67%</td>
</tr>
</tbody>
</table>
### 7.1. CA RULES TESTED ON 3X3 MATRICES, 1000 BEST, RULES WITH B8 EXCLUDED

<table>
<thead>
<tr>
<th>Rule Set</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>B7531/S642</td>
<td>67%</td>
</tr>
<tr>
<td>B7531/S7542</td>
<td>67%</td>
</tr>
<tr>
<td>B7531/S3</td>
<td>67%</td>
</tr>
<tr>
<td>B7542/S531</td>
<td>67%</td>
</tr>
<tr>
<td>B7542/S7531</td>
<td>67%</td>
</tr>
<tr>
<td>B7542/S640</td>
<td>67%</td>
</tr>
<tr>
<td>B7542/S5420</td>
<td>67%</td>
</tr>
<tr>
<td>B76321/S73</td>
<td>67%</td>
</tr>
<tr>
<td>B764321/S0</td>
<td>67%</td>
</tr>
<tr>
<td>B764321/S70</td>
<td>67%</td>
</tr>
<tr>
<td>B764321/S470</td>
<td>67%</td>
</tr>
<tr>
<td>B764321/S740</td>
<td>67%</td>
</tr>
<tr>
<td>B764321/S7540</td>
<td>67%</td>
</tr>
<tr>
<td>B764321/S76540</td>
<td>67%</td>
</tr>
<tr>
<td>B31/S65420</td>
<td>68%</td>
</tr>
<tr>
<td>B421/S6430</td>
<td>68%</td>
</tr>
<tr>
<td>B42/S31</td>
<td>68%</td>
</tr>
<tr>
<td>B42/S731</td>
<td>68%</td>
</tr>
<tr>
<td>B4321/S35</td>
<td>68%</td>
</tr>
<tr>
<td>B4321/S760</td>
<td>68%</td>
</tr>
<tr>
<td>B531/S742</td>
<td>68%</td>
</tr>
<tr>
<td>B54321/S540</td>
<td>68%</td>
</tr>
<tr>
<td>B54321/S640</td>
<td>68%</td>
</tr>
<tr>
<td>B54321/S650</td>
<td>68%</td>
</tr>
<tr>
<td>B54321/S6540</td>
<td>68%</td>
</tr>
<tr>
<td>B54321/S7650</td>
<td>68%</td>
</tr>
<tr>
<td>B54321/S76540</td>
<td>68%</td>
</tr>
<tr>
<td>B6421/S76430</td>
<td>68%</td>
</tr>
<tr>
<td>B64321/S50</td>
<td>68%</td>
</tr>
<tr>
<td>B64321/S6540</td>
<td>68%</td>
</tr>
<tr>
<td>B64321/S750</td>
<td>68%</td>
</tr>
<tr>
<td>B64321/S740</td>
<td>68%</td>
</tr>
<tr>
<td>B64321/S7650</td>
<td>68%</td>
</tr>
<tr>
<td>B6542/S76531</td>
<td>68%</td>
</tr>
<tr>
<td>B65421/S720</td>
<td>68%</td>
</tr>
<tr>
<td>B654321/S740</td>
<td>68%</td>
</tr>
<tr>
<td>B742/S731</td>
<td>68%</td>
</tr>
<tr>
<td>B74321/S35</td>
<td>68%</td>
</tr>
<tr>
<td>B7531/S520</td>
<td>68%</td>
</tr>
<tr>
<td>B7531/S720</td>
<td>68%</td>
</tr>
<tr>
<td>B7531/S73</td>
<td>68%</td>
</tr>
<tr>
<td>B764321/S40</td>
<td>68%</td>
</tr>
<tr>
<td>B764321/S55</td>
<td>68%</td>
</tr>
<tr>
<td>B764321/S560</td>
<td>68%</td>
</tr>
<tr>
<td>B764321/S6540</td>
<td>68%</td>
</tr>
<tr>
<td>B764321/S7650</td>
<td>68%</td>
</tr>
<tr>
<td>B764321/S76540</td>
<td>68%</td>
</tr>
<tr>
<td>B4321/S76420</td>
<td>69%</td>
</tr>
<tr>
<td>B54321/S7500</td>
<td>69%</td>
</tr>
<tr>
<td>B54321/S7600</td>
<td>69%</td>
</tr>
<tr>
<td>B54321/S7400</td>
<td>69%</td>
</tr>
<tr>
<td>B54321/S7640</td>
<td>69%</td>
</tr>
<tr>
<td>B54321/S7650</td>
<td>69%</td>
</tr>
<tr>
<td>B64321/S6430</td>
<td>69%</td>
</tr>
<tr>
<td>B64321/S7643</td>
<td>69%</td>
</tr>
<tr>
<td>B64321/S650</td>
<td>69%</td>
</tr>
<tr>
<td>B6531/S6420</td>
<td>69%</td>
</tr>
<tr>
<td>B6531/S7642</td>
<td>69%</td>
</tr>
<tr>
<td>B6531/S742</td>
<td>69%</td>
</tr>
<tr>
<td>B6531/S7642</td>
<td>69%</td>
</tr>
<tr>
<td>B6531/S76542</td>
<td>69%</td>
</tr>
<tr>
<td>B731/S765420</td>
<td>69%</td>
</tr>
<tr>
<td>B7421/S76430</td>
<td>69%</td>
</tr>
<tr>
<td>B7531/S755</td>
<td>69%</td>
</tr>
<tr>
<td>B7531/S6542</td>
<td>69%</td>
</tr>
<tr>
<td>B7531/S742</td>
<td>69%</td>
</tr>
<tr>
<td>B75431/S6420</td>
<td>69%</td>
</tr>
<tr>
<td>B75431/S76420</td>
<td>69%</td>
</tr>
<tr>
<td>B754321/S7640</td>
<td>69%</td>
</tr>
<tr>
<td>B754321/S6540</td>
<td>69%</td>
</tr>
<tr>
<td>B754321/S7540</td>
<td>69%</td>
</tr>
<tr>
<td>B754321/S7650</td>
<td>69%</td>
</tr>
<tr>
<td>B754321/S76540</td>
<td>69%</td>
</tr>
<tr>
<td>B76421/S6430</td>
<td>69%</td>
</tr>
<tr>
<td>B76421/S643</td>
<td>69%</td>
</tr>
<tr>
<td>B76421/S76430</td>
<td>69%</td>
</tr>
<tr>
<td>B76421/S7643</td>
<td>69%</td>
</tr>
<tr>
<td>B8321/S0</td>
<td>70%</td>
</tr>
<tr>
<td>B8321/S70</td>
<td>70%</td>
</tr>
<tr>
<td>B8321/S7500</td>
<td>70%</td>
</tr>
<tr>
<td>B8531/S6542</td>
<td>70%</td>
</tr>
<tr>
<td>B8531/S7642</td>
<td>70%</td>
</tr>
<tr>
<td>B8531/S76542</td>
<td>70%</td>
</tr>
<tr>
<td>B8631/S420</td>
<td>70%</td>
</tr>
</tbody>
</table>
7.1. CA RULES TESTED ON 3X3 MATRICES, 1000 BEST, RULES WITH B8 EXCLUDED

<table>
<thead>
<tr>
<th>Rule</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>B6421/S643</td>
<td>70%</td>
</tr>
<tr>
<td>B64321/S66</td>
<td>70%</td>
</tr>
<tr>
<td>B64321/S7540</td>
<td>70%</td>
</tr>
<tr>
<td>B7421/S6430</td>
<td>70%</td>
</tr>
<tr>
<td>B7421/S643</td>
<td>70%</td>
</tr>
<tr>
<td>B7421/S7643</td>
<td>70%</td>
</tr>
<tr>
<td>B74321/S765</td>
<td>70%</td>
</tr>
<tr>
<td>B7531/S7642</td>
<td>70%</td>
</tr>
<tr>
<td>B7531/S76542</td>
<td>70%</td>
</tr>
<tr>
<td>B754321/S540</td>
<td>70%</td>
</tr>
<tr>
<td>B754321/S760</td>
<td>70%</td>
</tr>
<tr>
<td>B7631/S6420</td>
<td>70%</td>
</tr>
<tr>
<td>B7631/S66200</td>
<td>70%</td>
</tr>
<tr>
<td>B7631/S6655</td>
<td>70%</td>
</tr>
<tr>
<td>B764321/S7540</td>
<td>70%</td>
</tr>
<tr>
<td>B421/S643</td>
<td>71%</td>
</tr>
<tr>
<td>B4321/S655</td>
<td>71%</td>
</tr>
<tr>
<td>B4321/S765</td>
<td>71%</td>
</tr>
<tr>
<td>B5431/S520</td>
<td>71%</td>
</tr>
<tr>
<td>B54321/S40</td>
<td>71%</td>
</tr>
<tr>
<td>B54321/S750</td>
<td>71%</td>
</tr>
<tr>
<td>B54321/S865</td>
<td>71%</td>
</tr>
<tr>
<td>B631/S6420</td>
<td>71%</td>
</tr>
<tr>
<td>B642/S31</td>
<td>71%</td>
</tr>
<tr>
<td>B64321/S5540</td>
<td>71%</td>
</tr>
<tr>
<td>B74321/S550</td>
<td>71%</td>
</tr>
<tr>
<td>B74321/S665</td>
<td>71%</td>
</tr>
<tr>
<td>B74321/S750</td>
<td>71%</td>
</tr>
<tr>
<td>B74321/S765</td>
<td>71%</td>
</tr>
<tr>
<td>B75431/S720</td>
<td>71%</td>
</tr>
<tr>
<td>B754321/S750</td>
<td>71%</td>
</tr>
<tr>
<td>B754321/S76520</td>
<td>71%</td>
</tr>
<tr>
<td>B654321/S50</td>
<td>72%</td>
</tr>
<tr>
<td>B654321/S650</td>
<td>72%</td>
</tr>
<tr>
<td>B654321/S750</td>
<td>72%</td>
</tr>
<tr>
<td>B731/S420</td>
<td>72%</td>
</tr>
<tr>
<td>B742/S7520</td>
<td>72%</td>
</tr>
<tr>
<td>B742/S76531</td>
<td>72%</td>
</tr>
<tr>
<td>B754321/S750</td>
<td>72%</td>
</tr>
<tr>
<td>B7631/S6420</td>
<td>72%</td>
</tr>
<tr>
<td>B7631/S66200</td>
<td>72%</td>
</tr>
<tr>
<td>B7631/S6655</td>
<td>72%</td>
</tr>
<tr>
<td>B764321/S5540</td>
<td>72%</td>
</tr>
<tr>
<td>B764321/S7540</td>
<td>72%</td>
</tr>
<tr>
<td>B5431/S750</td>
<td>73%</td>
</tr>
<tr>
<td>B54321/S650</td>
<td>73%</td>
</tr>
<tr>
<td>B54321/S750</td>
<td>73%</td>
</tr>
<tr>
<td>B631/S6420</td>
<td>73%</td>
</tr>
<tr>
<td>B642/S31</td>
<td>73%</td>
</tr>
<tr>
<td>B64321/S5540</td>
<td>73%</td>
</tr>
<tr>
<td>B74321/S550</td>
<td>73%</td>
</tr>
<tr>
<td>B74321/S665</td>
<td>73%</td>
</tr>
<tr>
<td>B74321/S750</td>
<td>73%</td>
</tr>
<tr>
<td>B74321/S765</td>
<td>73%</td>
</tr>
<tr>
<td>B75431/S720</td>
<td>73%</td>
</tr>
<tr>
<td>B754321/S750</td>
<td>73%</td>
</tr>
<tr>
<td>B754321/S76520</td>
<td>73%</td>
</tr>
<tr>
<td>B754321/S40</td>
<td>74%</td>
</tr>
<tr>
<td>B754321/S650</td>
<td>74%</td>
</tr>
<tr>
<td>B754321/S750</td>
<td>74%</td>
</tr>
<tr>
<td>B764321/S540</td>
<td>74%</td>
</tr>
<tr>
<td>B764321/S7540</td>
<td>74%</td>
</tr>
<tr>
<td>B31/S520</td>
<td>74%</td>
</tr>
<tr>
<td>B31/S650</td>
<td>74%</td>
</tr>
<tr>
<td>B31/S750</td>
<td>74%</td>
</tr>
<tr>
<td>B5431/S550</td>
<td>74%</td>
</tr>
<tr>
<td>B54321/S650</td>
<td>74%</td>
</tr>
<tr>
<td>B54321/S750</td>
<td>74%</td>
</tr>
<tr>
<td>B631/S6420</td>
<td>74%</td>
</tr>
<tr>
<td>B642/S31</td>
<td>74%</td>
</tr>
<tr>
<td>B64321/S5540</td>
<td>74%</td>
</tr>
<tr>
<td>B74321/S550</td>
<td>74%</td>
</tr>
<tr>
<td>B74321/S665</td>
<td>74%</td>
</tr>
<tr>
<td>B74321/S750</td>
<td>74%</td>
</tr>
<tr>
<td>B74321/S765</td>
<td>74%</td>
</tr>
<tr>
<td>B75431/S720</td>
<td>74%</td>
</tr>
<tr>
<td>B754321/S750</td>
<td>74%</td>
</tr>
<tr>
<td>B754321/S76520</td>
<td>74%</td>
</tr>
<tr>
<td>B31/S6420</td>
<td>75%</td>
</tr>
<tr>
<td>B42/S531</td>
<td>75%</td>
</tr>
</tbody>
</table>
7.2. COLOURMAP

B5431/S6520 => 75%
B642/S76531 => 75%
B64321/S650 => 75%
B7321/S7650 => 75%
B75431/S620 => 75%
B7642/S76531 => 75%
B764321/S650 => 75%
B4321/S650 => 76%
B4321/S7650 => 76%
B5431/S620 => 76%
B642/S7531 => 76%
B74321/S531 => 76%
B74321/S650 => 76%
B7531/S5420 => 77%
B531/S5420 => 77%
B6531/S65420 => 77%
B6531/S765420 => 77%
B7531/S5420 => 77%
B7642/S6551 => 77%
B7642/S7531 => 77%
B42/S531 => 78%
B7642/S531 => 78%
B531/S65420 => 81%
B531/S765420 => 81%
B7531/S65420 => 81%
B6531/S65420 => 83%
B6531/S7420 => 84%
B531/S420 => 85%
B7531/S65420 => 85%
B7531/S7420 => 87%
B6531/S6420 => 89%
B7531/S7420 => 89%
B531/S6420 => 89%
B7531/S76420 => 89%
B531/S6420 => 91%
B7531/S76420 => 91%
B6531/S76420 => 91%

7.2 ColourMap

```java
import java.math.BigInteger;
import java.util.Iterator;
import java.util.LinkedList;

/**
 * @author talas
 */
public class ColorMap {
    private static final byte MAX_ELIMINATION_SIZE = 2;
    private final byte[] colors;
    public final byte[] aviColors;
    protected byte[] aviColorsCounter;
    public final long startComplexity;
    private boolean broken = false;
    private boolean crashed = false;
    private boolean[] notBorn;
    private boolean[] notSurvived;
    private LinkedList<Elimination> eliminations;
    private LinkedList<Integer> colorRelations;

    public void gotoRandom() {
        boolean needRecurse = false;
        byte[] randColors = new byte[colors.length];
        for (int i = 0; i < randColors.length; i++) {
            randColors[i] = aviColors[i].get((byte) (Math.random() * aviColors[i].size()));
        }
        if (!this.tryGoto(randColors)) {
            needRecurse = true;
        }
    }
```
```java
private void optimizeEliminations()
{
    // find duplicate and overlapping eliminations and remove them...
    System.out.println("E: "+eliminations.size());
    int timeOut = Math.max eliminations.size() / 10, 6000;
    LinkedList&lt;Elimination&gt; eliminated = new LinkedList&lt;Elimination&gt;();
    for(Elimination e : eliminations)
    {
        if(e.checked) continue;
        for(Elimination s : eliminations)
        {
            if(e.containedIn(s))
                eliminated.add(s);
            break;
        }
        e.checked = true;
        if(--timeOut == 0) break;
    }
    for(Elimination i : eliminated) eliminations.remove(i);  
    if(timeOut == 0)
        System.out.println("restart!");
    optimizeEliminations();
}

public boolean tryGoto(byte[] givenColors) {
    if(colors.length != givenColors.length)
        return false;
    for(int i = 0; i < givenColors.length; i++) {
        colors[i] = givenColors[i];
        aviColorsCounter[i] = 0;
        boolean found = false;
        for(byte j = 0; j < aviColors[i].size(); j++) {
            if(aviColors[i].get(j) == givenColors[i])
            {
                aviColorsCounter[i] = j;
                found = true;
                break;
            }
        }
        if(!found)
            return false;
    }  
    return true;
}

private final class Elimination {
    /*
    * count : how many number/color pairs in this elimination
    */
    final byte count;
    final int[] elimNumbers;
    final byte[] elimColors;
    boolean checked;
    //public boolean discarded = false;

    @Override
    public String toString()
    {
        StringBuilder sb = new StringBuilder();
        for(int i = 0; i < elimNumbers.length; i++){
            if(i &gt; 0)
                sb.append(";");
            sb.append("num: ");
            sb.append(\elimNumbers[i]);
            sb.append("color: ");
            sb.append(\elimColors[i]);
        }
        return sb.toString();
    }

    // Single elimination
    public Elimination(int num, byte color){
        count = 1;
        elimNumbers = new int[num];
        elimColors = new byte[\color];
    }
```

// Double elimination
public Elimination(int num1, int num2, byte color1, byte color2) {
    count = 2;
    elimNumbers = new int[count];
    elimNumbers[0] = num1;
    elimNumbers[1] = num2;
    elimColors = new byte[count];
    elimColors[0] = color1;
    elimColors[1] = color2;
}

// X elimination
public Elimination(int[] numbers, byte[] colors) {
    count = (byte)Math.min(numbers.length, colors.length);
    if (count < 1) {
        System.err.println("Bugged Elimination created, count<1!");
        this.elimNumbers = numbers;
        this.elimColors = colors;
    } else if (count < 2) {
        // this is a bugged elimination...
        return false;
    }
    // the current colors are only eliminated by this elimination if all colors match.
    for (byte i = 0; i < elimNumbers.length; i++) {
        if (colors[elimNumbers[i]] != elimColors[i]) {
            return false;
        }
    }
    return true;
}

public final boolean isEliminated() {
    if (count == 1) {
        if (colors[elimNumbers[0]] != elimColors[0]) {
            return false;
        }
    } else if (count < 2) {
        // this is a bugged elimination...
        return false;
    }
    // Check if container has at least all the same num/color combinations...
    for (byte i = 0; i < this.elimNumbers.length; i++) {
        int num1 = this.elimNumbers[i];
        byte color1 = this.elimColors[i];
        boolean found = false;
        for (byte j = 0; j < container.elimNumbers.length; j++) {
            if (container.elimNumbers[j] == num1) {
                if (container.elimColors[j] != color1) {
                    // Not the same number/color pair
                    return false;
                }
            }
        }
        found = true;
        break;
    }
    if (!found) // eliminations doesn't have the same numbers...
        return false;
    return (container.count > this.count);
}

public final boolean contains(Elimination other) {
    // those that have the same colors can be swapped...
    for (byte i = 0; i < this.elimNumbers.length; i++) {
        int num1 = this.elimNumbers[i];
        byte color1 = this.elimColors[i];
        boolean found = false;
        for (byte j = 0; j < other.elimNumbers.length; j++) {
            if (other.elimNumbers[j] == num1) {
                if (other.elimColors[j] != color1) {
                    return false;
                }
            }
        }
        found = true;
        break;
    }
    if (!found) // eliminations doesn't have the same numbers...
        return false;
}
7.2. COLOURMAP

    return false;
    return true;
}

public final void optimize(byte[] ruleB, byte[] ruleS, Map m)
{
    for (int i = 0; i < aviColors.length; i++)
    {
        LinkedList<Byte> current = aviColors[i];
        if (current.size() == 0)
        {
            this.broken = true;
            return;
        }
        Iterator<Byte> iterator = current.iterator();
        boolean canBorn = false;
        boolean canSurvive = false;
        while (iterator.hasNext())
        {
            byte avi = iterator.next().byteValue();
            if (avi >= ruleB.length)
            {
                canSurvive = true;
            }
            else
            {
                canBorn = true;
            }
        }
        if (!canBorn)
        {
            this.notBorn[i] = true;
        }
        else if (!canSurvive)
        {
            this.notSurvived[i] = true;
        }
    }
}

PopulationIterator pi = new PopulationIterator(m);
LinkedList<Integer> NWCornerDwellers = new LinkedList<Integer>();

int current = 0;
while (pi.hasNext())
{
    // for each color, make eliminations based on available and relations
    pi.next();
    int[] xy = pi.getXY();
    int x = xy[1];
    int y = xy[0];
    if ((x == 0 || x == 1) && (y == 0 || y == 1))
    {  
        NWCornerDwellers.add(current);
    }
    int available = Map AVAILABLE Neighbours(x, y, m.getMap().length);
    LinkedList<Byte> avi = this.aviColors[current];
    Iterator<Byte> colorIterator = avi.iterator();
    int numRels = this.colorRelations[current].size();
    int numBorn = 0;
    int numSurvive = 0;
    Iterator<Integer> iterator1 = this.colorRelations[current].iterator();
    while (iterator1.hasNext())
    {
        // find out, for each relation if it must be born, survived or either.
        int i = iterator1.next();
        if (this.notBorn[i])
        {
            numSurvive++;
        }
        else if (this.notSurvived[i])
        {
            numBorn++;
        }
    }
    boolean[] eliminatedColors = new boolean[18];
    while (colorIterator.hasNext())
    {
        // check if enough available even when considering relations
        byte color = colorIterator.next();
        int countDoubles = 0;
        byte currentWanted = 0;
        if (color >= ruleB.length)
        {
            currentWanted = ruleS[color - ruleB.length];
        }
7.2. COLOURMAP

```java
else
    currentWanted = ruleB[color];

if(numSurvive > currentWanted) {
    // impossible
    // there are too many neighbours that MUST be true for this color.
    eliminatedColors[color] = true;
    continue;
} else if (available - numBorn < currentWanted) {
    // impossible
    // there is NOT enough neighbours that can be true for this color.
    eliminatedColors[color] = true;
    continue;
} else if (available - numRels < currentWanted) {
    // some eliminations could be had?
    // because we depend on some rels to satisfy this color.
    int xElim = (available - currentWanted)+1;// how many required to break this color
    if(xElim == 1) {
        // double elimination
        for(Integer other : this.colorRelations[current])
            for(Byte otherColor : this.aviColors[other])
                if(otherColor < ruleB.length)
                    if(countDoubles < current) {
                        Iterator<Elimination> iterator =
                            this.eliminations.iterator();
                        Elimination n = new
                            Elimination(current,other,color,otherColor);
                        boolean found = false;
                        while(iterator.hasNext()){
                            Elimination t = iterator.next();
                            if(t.sameAs(n)){
                                found = true;
                                break;
                            }
                            if((found)) {
                                countDoubles--;
                            }
                            if((found)) {
                                //Discarded = true;
                                //countDoubles++;
                                //continue;
                            }
                            // check if it ACTUALLY should be discarded...
                        }
                        Elimination n = new Elimination(current,other,color,otherColor);
                        //n.discarded = discarded;
                        this.eliminations.add(n);
                        countDoubles++;
                    }
                } else {
                    //pick x and x colors to eliminate
                    if(xElim > numRels)
                        continue; // cant satisfy this x????
                    if(xElim > MAX_ELIMINATION_SIZE)
                        continue;
                    Combinations c = new Combinations(numRels, xElim);
                    while(c.hasNext()) {
                        int[] elim = c.next();
                        byte[] mins = new byte[elim.length];
                        byte[] maxs = new byte[elim.length];
                        for(int i = 0; i < elim.length; i++)
```
7.2. COLOURMAP

```java
mins[i] = 99;
maxs[i] = -99;
}

System.out.println("<");
for(int i = 0; i < elim.length; i++) {
    // for each i in elim, also permute its colors that can kill...
    int other = this.colorRelations[current].get(elim[i]);
    for(byte j = 0; j < this.aviColors[other].size(); j++)
        Byte avicolor = this.aviColors[other].get(j);
        if(avicolor < ruleB.length){
            if(j < mins[i])
                mins[i] = j;
            if(j > maxs[i])
                maxs[i] = j;
        }
}

Permutations p = new Permutations(mins, maxs);
byte[] pElim = null;
do {
    pElim = p.next();
    if(pElim != null)
        int[] numbers = new int[pElim.length+1];
        numbers[0] = current;
        byte[] currentColors = new byte[pElim.length+1];
        currentColors[0] = color;
        for(int i = 0; i < pElim.length; i++)
            numbers[i+1] = elim[i];
            currentColors[i+1] = this.aviColors[other].get(pElim[i]);
        }
}

Elimination n = new Elimination(numbers, currentColors);
//System.out.println("E+"+n);
this.eliminations.add(n);
```

```java
else if(numRelis > currentWanted){
    // in some cases we could get too many, so can eliminate some here too...
    int xElim = currentWanted+1; // how many required to break this color
    if(xElim == 1){
        // double elimination
        for(Integer other : this.colorRelations[current])
            for(Byte otherColor : this.aviColors[other])
                // if otherColor is inruleS, we can eliminate...
                if(otherColor >= ruleB.length)
                    Elimination n = new Elimination(current, other, color, otherColor);
                    //System.out.println("E-"+n);
                    this.eliminations.add(n);
    }
}
```

```java
else {
    if(xElim > numRelis)
        continue; // cant satisfy this x....
    if(xElim > MAX_ELIMINATION_SIZE)
        continue;

    Combinations c = new Combinations(numRelis, xElim);
    while(c.hasNext())
        int[] elim = c.next();
        byte[] mins = new byte[elim.length];
        byte[] maxs = new byte[elim.length];
        for(int i = 0; i < elim.length; i++)
            mins[i] = 99;
            maxs[i] = -99;
            //System.out.append(" + this.colorRelations[current].get(elim[i])
            System.out.print(<");
```
7.2. COLOURMAP

```java
mins[i] = 99;
mass[i] = -99;
//System.out.append("+this.colorRelations[current].get elim[i]);
} //System.out.println("<");
for(int i = 0; i < elim.length; i++)
    // for each i in elim, also permute its colors that can kill...
    int other = this.colorRelations[current].get elim[i];
    for(byte j = 0; j < this.aviColors[other].size(); j++){
        byte aviColor = this.aviColors[other].get(j);
        if(aviColor >= ruleB.length) // is inside ruleS!
            if(j < mins[i])
                mins[i] = j;
            if(j > maxs[i])
                maxs[i] = j;
    }

Permutations p = new Permutations(mins, mass);
byte[] pElim = null;
do {
    pElim = p.next();
    if(pElim != null){
        int[] numbers = new int[pElim.length+1];
        numbers[0] = current;
        byte[] currentColors = new byte[pElim.length+1];
        currentColors[0] = color;
        for(int i = 0; i < pElim.length; i++)
            int other = this.colorRelations[current].get elim[i];
            numbers[i+1] = other;
        currentColors[i+1] = this.aviColors[other].get(pElim[i]);
    }
    Elimination n = new Elimination(numbers, currentColors);
    //System.out.println("E+"+n);
    this.eliminations.add(n);
} while(pElim != null);
}

// finally...
current++;}
```

// TODO: Greetings gentlemen, this is corner control.
// go thru each corner and check its color stuffs.
// eliminate combinations that create 'prisoners with no hope'.
// start with northwest corner...
// check if it has at least 2 dots adjacent to it, otherwise no elimination??
// North-West
if(NWCornerDwellers.size() == 3){
    // must be room for some eliminations here.
    // if all of them are born AND at least one requires the corner -> corner will live and have zero
    // if all of them survive AND at least one requires the corner -> corner will live and have 3
    // if all of them survive AND none of them requires the corner -> corner is dead and have 3
    boolean allBorn = true;
    boolean allSurvive = true;
    for(Integer cur : NWCornerDwellers)
        if(this.notBorn[cur])
            allBorn = false;
        if(this.notSurvived[cur])
            allSurvive = false;

```
7.2. COLOURMAP

```java
74 | if ( allBorn )
75 | // check if at least 1 requires the corner...
76 | // boolean reqFound = false;
77 | for ( Integer cur : NWCornerDwellers )
78 | }
79 | /\'*
80 | if ( allSurvive )
81 | // check if at least 1 requires the corner...
82 | |
83 | /\'*
84 | System.out.println( "Optimizing ..." );
85 | this.optimizeEliminations();
86 | System.out.println( "Done optimizing eliminations." );
88 | public ColorMap( ColorMap copyMap ) {
89 |   this.aviColors = copyMap.aviColors;
90 |   this.broken = copyMap.broken;
91 |   this.eliminations = copyMap.eliminations;
92 |   this.numColors = copyMap.numColors;
93 |   this.startComplexity = copyMap.startComplexity;
94 |   this.aviColorsCounter = new byte[copyMap.aviColorsCounter.length];
95 |   this.colors = new byte[copyMap.colors.length];
96 |   for ( int i = 0; i < colors.length; i++ )
97 |     colors[i] = aviColors[i].getFirst().byteValue();
98 | }
100 | /* *
101 | * "Blank" ColorMap
102 | * max size <= Integer.MAX_VALUE
103 | * param size
104 | *
105 | @SuppressWarnings("unchecked")
106 | public ColorMap( int size, byte[] ruleB, byte[] ruleS, Map map ) {
107 |   this.colors = new byte[size];
108 |   this.numColors = (byte)[(byte)ruleB.length+ruleS.length];
109 |   this.aviColors = new LinkedList[size];
110 |   this.eliminations = new LinkedList<Elimination>();
111 |   this.colorRelations = new LinkedList[size];
112 |   this.notBorn = new boolean[size];
113 |   this.notSurvived = new boolean[size];
114 |   for ( int i = 0; i < aviColors.length; i++ )
115 |     aviColors[i] = new LinkedList<Byte>();
116 | PopulationIterator pi = new PopulationIterator(map);
118 |   Position[] positions = new Position[size];
119 |   int currentCellNumber = 0;
120 |   StringBuilder complexityString = new StringBuilder();
121 |   complexityString.append( "CM Perms: ");
122 |   long complexity = 1;
123 |   { // Initial cycle to find all the positions...
124 |     int clc = 0;
125 |     PopulationIterator pi_tmp = new PopulationIterator(map);
126 |     while ( pi_tmp.hasNext() )
127 |       pi_tmp.next();
128 |     int[] yx = pi_tmp.getXY();
129 |     int currentCellNumber = new Position(yx[1], yx[0]);
130 |   }
132 |   while ( pi.hasNext() )
133 |     pi.next();
134 |     int[] yx = pi.getXY();
135 |     int x = yx[1];
136 |     int y = yx[0];
137 |     int available = Map.getAvailableNeighbours(x, y, map.getMap().length);
138 |     this.colorRelations[currentCellNumber] = new LinkedList<Integer>();
139 |     for ( int p = 0; p < positions.length; p++ )
140 |       if ( positions[p].isNextTo(positions[currentCellNumber]) )
141 |         this.colorRelations[currentCellNumber].add(p);
142 |   }
72
```
7.2. COLOURMAP

```java
for (byte i = 0; i < ruleB.length; i++)
    if (ruleB[i] <= available)
        aviColors[currentCellNumber].add((Byte)i);

byte l = (byte)ruleB.length;
for (byte i = 0; i < ruleS.length; i++)
    if (ruleS[i] <= available)
        byte n = (byte)(i+l);
    aviColors[currentCellNumber].add(n);

if (aviColors[currentCellNumber].size() <= 0)
    broken = true;
    startComplexity = 0;
    return;

System.out.append(currentCellNumber + " : ");
for (int i = 0; i < aviColors[currentCellNumber].size(); i++)
    System.out.append(aviColors[currentCellNumber].get(i));
System.out.println();
if (currentCellNumber != 0)
    complexityString.append("*");
complexity *= aviColors[currentCellNumber].size();
complexityString.append(aviColors[currentCellNumber].size());
colors[currentCellNumber] = aviColors[currentCellNumber].getFirst().byteValue();
currentCellNumber ++;
```

```java
            this.optimize(ruleB, ruleS, map);

            StringBuilder cp2 = new StringBuilder();
            cp2.append("NeoComplexity :");
            BigInteger n = BigInteger.ONE;
            for (int i = 0; i < this.aviColors.length; i++)
                n = n.multiply(BigInteger.valueOf(this.aviColors[i].size()));
            System.out.println(cp2);
            System.out.println(" Decimal places : "+(n.toString().length()-1));
```

```java
public boolean isBroken()
    return broken;
```

```java
public void print()
    for (int i = 0; i < colors.length; i++)
        System.out.append("" + colors[i]);
```
7.2. COLOURMAP

```java
754     System.out.println(".");
755 }
756 public int size()
757     return colors.length;
758 }
759 public byte getColor(int pos){ // throws ArrayOutOfBoundsException
760     return colors[pos];
761 }
762 public Byte[] getColors()
763     Byte[] b = new Byte[colors.length];
764     int i = 0;
765     for (byte c : colors)
766         b[i++] = c;
767     return b;
768 }
769 public void gotoNext()
770     if(numColors <= 1)
771         return; // Nothing to do.
772     add(colors.length-1);
773     //this.gotoRandom(), an option to iterating...
774     crashed = false;
775     for (Elimination e : eliminations)
776         if(e.isEliminated())
777             try
778                 do {
779                     add(colors.length-1);
780                     } while(e.isEliminated());
781             catch(java.lang.StackOverflowError soe){
782                 crashed = true;
783                 //System.out.println("*");
784                 break;
785             }
786             break;
787 }
788 }
789 public boolean hasCrashed()
790     return crashed;
791 }
792 public boolean hasNext()
793     if(numColors <= 1)
794         return false;
795     for(int i = 0; i < colors.length; i++)
796         if(colors[i] != aviColors[i].getLast().byteValue())
797             if(aviColorsCounter[i] != aviColors[i].size()-1)
798                 return true;
799         }
800     return false;
801 }
802 }
803 private void add(int pos)
804     if(pos < 0 || pos >= colors.length)
805         this.broken = true;
806     return;
807     if(aviColorsCounter[pos] < aviColors[pos].size()-1)
808         colors[pos] = aviColors[pos].get(++aviColorsCounter[pos]);
809     return;
810     else
811         aviColorsCounter[pos] = 0;
812     if(aviColors[pos].size() == 0)
813         return;
814     colors[pos] = aviColors[pos].getFirst().byteValue();
815     add(pos-1);
816 }
817
818```
import java.util.Iterator;
import java.util.LinkedList;

/**
 * @author talas
 */
public final class PermMap implements Runnable {
    private final byte[][] map;
    private final byte[][] neighbourMap;
    private final boolean[][] doneMap;
    private final boolean[][] dontCareMap;
    public static final byte FINAL_DEAD = 3;
    public static final byte FINAL_ALIVE = 4;
    public static final byte DIES_NOW = 5;
    public static final byte CAN = 1;
    public static final byte BORN = 2;
    public static final byte ruleB = Global.ruleB;
    public static final byte ruleS = Global.ruleS;
    public boolean hasSolution = false;
    public boolean hasForks = false;
    private static final byte[] ruleB = Global.ruleB;
    private static final byte[] ruleS = Global.ruleS;
    private static long superIt = 0;
    private long myIt = superIt++;
    public void setStaticMap(boolean[][] smap) {
        for(byte i = 0; i < smap.length; i++)
            for(byte j = 0; j < smap.length; j++)
                if(smap[i][j] == true) {
                    if(map[i][j] != FINAL_ALIVE) {
                        map[i][j] = DIES_NOW;
                    }
                }
    }
    public static final boolean debug = false;
    private Map solution;
    public long getScore() {
        long n = 0;
        for(int i = 0; i < doneMap.length; i++)
            for(int j = 0; j < doneMap.length; j++)
                if(doneMap[i][j]) {
                    byte code = map[i][j];
                    if(code == FINAL_ALIVE) n += 10;
                    else n += 1;
                }
        return n;
    }
    public byte[][] getTempMap() {
        return map;
    }
    public byte[][] getNeighbourMap() {
        return neighbourMap;
    }
    public Map buildSolution() {
        if(!hasSolution) return null;
        if(solution != null)
            return solution;
        Map m = new Map(map.length);
        return m;
    }
}
7.3. PERMMAP

```java
for (int i = 0; i < map.length; i++) {
    for (int j = 0; j < map.length; j++) {
        m.setState(i, j, (map[i][j] == FINAL_ALIVE || map[i][j] == DIES_NOW));
    }
}
```

```
solution = m.clone();
return m;
```
7.3. PERMMAP

```java
byte color = def[i];
boolean born = true;
if (color >= ruleB.length) {
    born = false;
    neighbourMap[current.x][current.y] = ruleS[color-ruleB.length];
} else {
    neighbourMap[current.x][current.y] = ruleB[color];
}
if (born == true) {
    map[current.x][current.y] = BORN;
} else {
    map[current.x][current.y] = FINAL_ALIVE;
}
Global.ins++;
}

public PermMap(ColorMap def, Position[] dots, byte size) {
    map = new byte[size][size];
    neighbourMap = new byte[size][size];
    doneMap = new boolean[size][size];
    this.dontCareMap = null;
    for (int i = 0; i < neighbourMap.length; i++) {
        for (int j = 0; j < neighbourMap.length; j++) {
            neighbourMap[i][j] = -1;
        }
    }
    for (int i = 0; i < dots.length; i++) {
        Position current = dots[i];
        byte color = def.getColor(i);
        boolean born = true;
        if (color >= ruleB.length) {
            born = false;
            neighbours[i] = ruleS[color-ruleB.length];
        } else {
            neighbours[i] = ruleB[color];
        }
        if (born == true) {
            perms[i] = BORN;
        } else {
            perms[i] = FINAL_ALIVE;
        }
        Global.ins++;
    }
}

public PermMap(ColorMap definition, Map realMap) {
    byte[] neighbours = new byte[definition.size()];
    for (int i = 0; i < definition.size(); i++) {
        byte color = definition.getColor(i);
        boolean born = true;
        if (color >= ruleB.length) {
            born = false;
            neighbours[i] = ruleS[color-ruleB.length];
        } else {
            neighbours[i] = ruleB[color];
        }
        if (born == true) {
            perms[i] = BORN;
        } else {
            perms[i] = FINAL_ALIVE;
        }
    }
    PopulationIterator pi = new PopulationIterator(realMap);
    while (pi.hasNext()) {
        pi.next();
        int[] xy = pi.getXY();
        if (xy != null) {
            //System.out.println("x:" + xy[0] + ", y:" + xy[1] + ", c:" + count);
        }
    }
```
7.3. PERMMAP

```java
7.3. PERMMAP

map[xy[1]][xy[0]] = perms[(int)pi.getCurrent()];
Global.ins++;
neighbourMap[xy[1]][xy[0]] = neighbours[(int)pi.getCurrent()];

if(debug) {
    System.out.println("b-");
    Global.printMap(map);
    Global.printMap(neighbourMap);
    System.out.println("b-");
}

public boolean solveDunno(int x, int y) {
    for(int i = -1; i <= 1; i++) {
        for(int j = -1; j <= 1; j++) {
            if(x+i < 0 || x+i >= map.length || y+j < 0 || y+j >= map.length || (i == 0 && j == 0)) {
                continue;
            }
            byte nWants = neighbourMap[x+i][y+j];
            if(nWants != -1) {
                map[x+i][y+j] = CAN;
                Global.ins++;
                neighbourMap[x+i][y+j] = -1;
                return true;
            }
        }
    }
    return false;
}

public byte countNeighbours(int x, int y) {
    byte count = 0;
    for(int i = -1; i <= 1; i++) {
        for(int j = -1; j <= 1; j++) {
            if(x+i < 0 || x+i >= map.length || y+j < 0 || y+j >= map.length || (i == 0 && j == 0)) {
                continue;
            }
            byte code = map[x+i][y+j];
            if(code == FINAL_ALIVE || code == DIES_NOW) {
                count ++;
            }
        }
    }
    return count;
}

public void killCAN(int x, int y) {
    map[x][y] = FINAL_DEAD;
    Global.ins++;
    keepDead(x,y);
}

public boolean keepDead(int x, int y) {
    if(this.dontCareMap != null) {
        if(this.dontCareMap[x][y]) {
            doneMap[x][y] = true;
            return true; // dont fix 'dont cares'
        }
    }
    boolean dies = (map[x][y] == DIES_NOW);
    byte countNeighbours = 0;
    byte canDunnoCount = 0;
    LinkedList<Position> cans = new LinkedList<Position>();
    for(byte i = -1; i <= 1; i++) {
        for(byte j = -1; j <= 1; j++) {
            int xi = x+i;
            int yj = y+j;
            if(xi < 0 || xi >= map.length || yj < 0 || yj >= map.length || (i == 0 && j == 0)) {
                continue;
            }
            byte code = map[xi][yj];
            if(code == FINAL_ALIVE || code == DIES_NOW) {
                countNeighbours++;
                canDunnoCount = 0;
                cans.add(new Position(xi,yj));
            }
            if(code == CAN || code == DUNNO) {
                canDunnoCount++;
            }
        }
    }
    // End for loops
```
boolean fail = false;
if (!dies) {// "dies" = trying to stay dead
for (byte i = 0; i < ruleB.length; i++)
    if (ruleB[i] == countNeighbours)
        fail = true;
    else
        for (byte i = 0; i < ruleS.length; i++)
            if (ruleS[i] == countNeighbours)
                fail = true;
if (!fail) {
    // solved for the moment...
    neighbourMap[x][y] = countNeighbours;
    return true;
}
byte first = -1;
for (byte i = countNeighbours; i < 9; i++)
    boolean has = false;
    for (byte j = 0; j < (!dies ? ruleB.length : ruleS.length); j++)
        if (!dies && ruleB[j] == i) {
            has = true;
        } else if (dies && ruleS[j] == i) {
            has = true;
        }
    if (!has) {
        first = i;
        break; // no need to continue...
    }
if (first == -1) {
    // Cant make this one stay dead...
    return false;
}
if (first == countNeighbours + canDunnoCount) {
    // Only one way to do it... so... do it...
    Iterator<Position> iterator = cans.iterator();
    while (iterator.hasNext()) {
        Position p = iterator.next();
        map[p.x][p.y] = DIES_NOW;
        Global.ins++;
        neighbourMap[x][y] = -1;
        doneMap[x][y] = true;
    } else {
        // Act how many we think we need...
        neighbourMap[x][y] = first;
    }
    return true;
}
public byte runValue = -99;
@Override
public void run() {
    runValue = go();
}
private byte go() {
    boolean allDone = false;
    int timeOut = 1080;
    while (!allDone) {
        allDone = true;
        timeOut = -1;
        if (timeOut == 0)
            break;
        boolean noChange = true;
        for (int x = 0; x < map.length; x++)
            for (int y = 0; y < map.length; y++)
                if (neighbourMap[x][y] != -1)
7.3. PERMMAP

```java
if (this.countNeighbours(x, y) == neighbourMap[x][y]) {
    if (this.map[x][y] == BORN || this.map[x][y] == FINAL_ALIVE) {
        // kill off the remaining CANs and DUNNOs.
        for (int i = -1; i <= 1; i++)
            for (int j = -1; j <= 1; j++)
                if ((x + i < 0 || x + i >= map.length || y + j < 0 || y + j >= map.length ||
                    (i == 0 && j == 0)))
                    continue;
    }
    byte code = map[x+i][y+j];
    if (code == CAN || code == DUNNO) {
        killCAN(x+i, y+j);
    } else {
        doneMap[x][y] = true;
    }
    if (doneMap[x][y])
        continue;
    byte reqNeighbours = neighbourMap[x][y];
    byte availableNeighbours = 8;
    byte relNeighbours = reqNeighbours;

    if (map[x][y] == CAN) {
        continue; // Nothing to do.
    }
    boolean iDunno = (map[x][y] == DUNNO);
    if (iDunno) {
        boolean tmp = solveDunno(x, y);
        if (allDone)
            allDone = tmp;
        continue;
    }
    allDone = false;
    availableNeighbours = Map.getAvailableNeighbours(x, y, map.length);

    //System.out.println("req : " + reqNeighbours + " / " + availableNeighbours);
    if (reqNeighbours > availableNeighbours) {
        return 0; // ColorMap is wrong.. corner case
    }
    // 1 check for 'zeros' and nullify neighbours, if possible -> discard colormap
    // 2 check for cells that have too many neighbours, if found -> discard colormap
    // 3 check for cells that have been satisfied and nullify neighbours, if possible
    // discard colormap
    // 4 check if cell has same amount of avi and req, if so, solve req..
    if (map[x][y] == FINAL_DEAD || map[x][y] == DIES_NOW) {
        boolean test = this.keepDead(x, y);
        if (!test) {
            return 7;
        }
    }
    byte canCount = 0;
    for (int i = 1; i <= 1; i++)
        for (int j = -1; j <= 1; j++)
            if ((x + i < 0 || x + i >= map.length || y + j < 0 || y + j >= map.length ||
                    (i == 0 && j == 0)))
                    continue;
    byte code = map[x+i][y+j];
    if (code == FINAL_ALIVE || code == DIES_NOW) {
        if (relNeighbours <= 0)
            return 1; // ColorMap is wrong.. too many neighbours for this cell..
    }
}
```

80
7.3. PERMMAP

```java
734 relNeighbours--; 
735
736 if (code == CAN || code == DUNNO) {
737     if (relNeighbours == 0) // set it to dead...
738         map[x+i][y+j] = FINAL_DEAD; // zero case nullify neighbours
739         Global.ins++;
740         noChange = false;
741     } else 
742         canCount++;
743 }
744
745 if (relNeighbours == 0) 
746     doneMap[x][y] = true;
747 else {
748     if (canCount < relNeighbours)
749         return 2;
750     else if (canCount == relNeighbours) {
751         // Solve
752         for (int i = -1; i <= 1; i++)
753             for (int j = -1; j <= 1; j++) {
754                 if (x+i < 0 || x+i >= map.length || y+j < 0 || y+j >= map.length ||
755                     (i == 0 && j == 0))
756                     continue;
757                 if (map[x+i][y+j] == CAN || map[x+i][y+j] == DUNNO)
758                     map[x+i][y+j] = DIES_NOW;
759                 Global.ins++;
760             }
761         doneMap[x][y] = true;
762         noChange = false;
763     }
764     // end for y
765     // end for x
766     if (noChange)
767         break; // if nothing changes we're done for now..
768 } // end while
769
770 if (debug) {
771     System.out.println("f+");
772     Global.printMap(map);
773     Global.printMap(neighbourMap);
774     Global.printMap(doneMap, 'd', 'n');
775     System.out.println("f-");
776 }
777
778 boolean solved = true;
779 for (int i = 0; i < map.length; i++)
780     for (int j = 0; j < map.length; j++) {
781         if (!doneMap[i][j] || !solved)
782             return -1;
783     }
784     //System.out.println("PermMap solved!");
785 }
786 else {
787     hasForks = true;
788     //System.out.println("/////// FORKING ///////////");
789     int canCount = 0;
790     for (int i = 0; i < map.length; i++)
791         for (int j = 0; j < map.length; j++)
792             if (map[i][j] == CAN)
793                 canCount++;
794     
795 }
```

81
// look for 1/2's. "either/or's"
// that is: dots that require 1 and have 2 cans near it.
// else "smallest choice"
double miniPoints = 0;
byte smallestDiv = 99;
LinkedList<Position> bestChoices = null;

if(true)
{
    for(int x = 0; x < map.length; x++)
    {
        for(int y = 0; y < map.length; y++)
        {
            if(map[x][y] == CAN && map[x][y] != DLNN)
            {
                // check if we want more...
                byte current = 0;
                LinkedList<Position> thisOnesChoices = new LinkedList<Position>();
                for(int i = -1; i <= 1; i++)
                {
                    for(int j = -1; j <= 1; j++)
                    {
                        if((x+i < 0 || x+i >= map.length || y+j < 0 || y+j >= map.length)
                        {
                            continue;
                        }
                        byte code = map[x+i][y+j];
                        if(code == CAN || code == DLNN)
                        {
                            thisOnesChoices.add(new Position(x+i, y+j));
                        }
                        else if((code != BORN && code != FINAL_DEAD)
                            current++;)
                    }
                }
                byte req = (byte)((this.neighbourMap[x][y]−current);
                if(req >= 1 && req <= smallestDiv)
                {
                    boolean reallySmaller = (req < smallestDiv);
                    double myMinipoints = Math.abs(thisOnesChoices.size()−(req/2.0));
                    if(thisOnesChoices.size() >= 1 && (myMinipoints > miniPoints ||
                        reallySmaller))
                    {
                        smallestDiv = req;
                        miniPoints = myMinipoints;
                        bestChoices = thisOnesChoices;
                        //System.out.println("n:"+myIt);
                    }
                }
            }
        }
    }
}

if(bestChoices != null)
{
    //System.out.println("bestChoices.size():"+bestChoices.size());
    for(int current = 0; current < bestChoices.size(); current++)
    {
        // set number 'current' as DBS_NOW and 'go' at it.
        byte[][] copyMap = Global.cloneMap(this.map);
        byte[][] copyNeighbour = Global.cloneMap(this.neighbourMap);
        boolean[][] copyDone = Global.cloneMap(doneMap);
        Position currentPos = bestChoices.get(current);
        copyMap[currentPos.x][currentPos.y] = DBS_NOW;
        Global.ins++;

        PermMap newMap = null;
        if(this.dontCareMap == null)
        {
            newMap = new PermMap(copyMap, copyNeighbour, copyDone);
            else
            {
                newMap = new PermMap(copyMap, copyNeighbour, copyDone, dontCareMap);
            }
        }
        byte value = newMap.go();
        if(debug)
        {
            System.out.println(""+value);
        }
        if(value == -1)
        {
            // check if found
            if(newMap.hasSolution)
            {
                // ok, set solution.. and return
                if(this.hasSolution)
                {
                    System.out.println("new??");
                    solution.printMap();
                }
            }
        }
    }
}
synchronized(Global.solutions) {
    Global.solutions.add(solution);
    //System.out.println("go back.");
    this.solution = newMap.buildSolution();
    this.hasSolution = true;
    //return -1;
}
if (this.hasSolution) return -1;
تكون(ونص) | // Older code, brute force to insert dots.
for (int current = 1; current <= canCount; current++) {
    // set number 'current' as DIES_NOW and 'go' at it.
    byte[][] copyMap = Global.cloneMap(this.map);
    byte[][] copyNeighbour = Global.cloneMap(this.neighbourMap);
    boolean[][] copyDone = Global.cloneMap(doneMap);
    int count = 0;
    boolean found = false;
    for (int i = 0; i < copyMap.length; i++)
        for (int j = 0; j < copyMap.length; j++)
            if (copyMap[i][j] == CAN) count++;
    if (count == current) {
        copyMap[i][j] = DIES_NOW;
        found = true;
    }
    if (found) break;
    if (found) break;
    PermMap newMap = new PermMap(copyMap, copyNeighbour, copyDone);
    byte value = newMap.go();
    if (debug) {
        System.out.println("<--");
        System.out.println("t*" + value);
    }
    if (value == -1) {
        // check if found...
        if (newMap.hasSolution) {
            // ok, set solution... and return
            this.solution = newMap.buildSolution();
            this.hasSolution = true;
            if (debug) {
                solution.printMap();
                System.out.println("go back.");
            }
            return -1;
        }
    }
    */
    // Forks... need to permute...
    return -2;
}

7.4 Map

import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.util.Iterator;
import java.util.Random;
import java.util.logging.Level;
import java.util.logging.Logger;
import java.util.zip.*;

/* *
* @author talas
*/

public final class Map implements Iterable<Boolean> {
    private int population = 0;

    public void printMap() {
        if(real_map.length > 200) {
            System.out.println("Too big, not printing to console.");
            return;
        }
        for(int x = 0; x < real_map.length; x++) {
            StringBuilder sb = new StringBuilder();
            for(int y = 0; y < real_map.length; y++) {
                if(_getState(x,y))
                    sb.append("x");
                else
                    sb.append("o");
            }
            System.out.println(sb.toString());
        }
    }

    public String getOneliner() {
        StringBuilder sb = new StringBuilder();
        for(int x = 0; x < real_map.length; x++) {
            for(int y = 0; y < real_map.length; y++) {
                if(_getState(x,y))
                    sb.append("x");
                else
                    sb.append("o");
        }
        return sb.toString();
    }

    public static boolean[][] fromByte(byte[] bytes) {
        int size = (int)Math.ceil(Math.sqrt(bytes.length*8)) ;
        boolean[][] map = new boolean[size][size];
        int count = 0;
        for(int i = 0; i < bytes.length; i++) {
            int remainder = bytes[i] == 128;
            boolean current = false;
            for(int j = 128; j >= 1; j = (int)Math.floor(j / (0.0+2))){
                if(remainder >= j) {
                    remainder -= j;
                    current = true;
                }
            }
            if(current) {
                int y = (int)Math.floor(count/map.length+0.0);
                map[count/map.length*y][y] = current;
                count++;
            }
        }
        return map;
    }

    public byte[] toByte() {
        int[] res = new int[(real_map.length*real_map.length)/8];
        int currentbyte = 0;
        int myByte = 0;
        int bitNumber = 0;
        for(int i = 0; i < real_map.length; i++) {
            for(int j = 0; j < real_map.length; j++){
                switch(bitNumber){
                case 0:
                    myByte = ((real_map[j][i]) ? 128 : 0);
                    break;
                case 1:
                    myByte = ((real_map[j][i]) ? 64 : 0);
                    break;
                case 2:
                    myByte = ((real_map[j][i]) ? 32 : 0);
                    break;
                case 3:
                    myByte = ((real_map[j][i]) ? 16 : 0);
                    break;
                case 4:
                    myByte = ((real_map[j][i]) ? 8 : 0);
                    break;
                case 5:
                    myByte = ((real_map[j][i]) ? 4 : 0);
                    break;
                case 6:
                    myByte = ((real_map[j][i]) ? 2 : 0);
                    break;
                case 7:
                    myByte = ((real_map[j][i]) ? 1 : 0);
                    break;
                }
                currentbyte = (currentbyte << 1) + myByte;
                if(currentbyte >= 128) {
                    res[bitNumber/8] = (currentbyte & 127);
                    currentbyte >>= 7;
                }
            }
        }
        return res;
    }
}
```java
7.4. MAP

case 1:
    myByte += ((real_map[j][i]) ? 64 : 0);
    break;
case 2:
    myByte += ((real_map[j][i]) ? 32 : 0);
    break;
case 3:
    myByte += ((real_map[j][i]) ? 16 : 0);
    break;
case 4:
    myByte += ((real_map[j][i]) ? 8 : 0);
    break;
case 5:
    myByte += ((real_map[j][i]) ? 4 : 0);
    break;
case 6:
    myByte += ((real_map[j][i]) ? 2 : 0);
    break;
case 7:
    myByte += ((real_map[j][i]) ? 1 : 0);
    res[currentByte++] = (int)Math.abs(myByte);
    myByte = 0;
    bitNumber = -1;
    break;
}
bitNumber ++;
}
byte[] b = new byte[res.length];
for (int i = 0; i < res.length; i++) {
    b[i] = (byte)(res[i] - 128);
}
return b;

public int compressMap()
{
    try {
        int l = 0;
        int l2 = 0;
        byte[] sendBuf = toByte();
        Deflater d = new Deflater();
        d.setInput(sendBuf);
        d.finish();
        byte[] output = new byte[sendBuf.length];
        l = d.deflate(output);
        int n = (int)d.getBytesWritten();
        n = d.getTotalOut();
        d.finish();
        n = d.getTotalOut();
    }
    byte[] sendBuf = this.RLEncode().getBytes("ASCII");
    Deflater d = new Deflater();
    d.setInput(sendBuf);
    d.finish();
    byte[] output = new byte[sendBuf.length];
    l2 = d.deflate(output);
    int n = (int)d.getBytesWritten();
    n = d.getTotalOut();
    d.finish();
    n = d.getTotalOut();
}
if (l2 < l)
    //Choose rle
    System.out.println("Chose RLE: !");
    return l2;
}
return 1;
} catch (Exception ex) {
```
7.4. MAP

Logger.getLogger(Map.class.getName()).log(Level.SEVERE, null, ex);

return Integer.MAX_VALUE;

public Map getDiff(Map other)
{
    Map diff = new Map(Map xorMap(other.getMap(), real_map));
    return diff;
}

public static Map xorMap(Map map1, Map map2)
{
    int mapsize = map1.getMap().length;
    Map res = new Map(mapsize);
    for (int i = 0; i < mapsize; i++) {
        for (int j = 0; j < mapsize; j++) {
            if (map1.getState(i, j) && !map2.getState(i, j)) {
                res.setState(i, j, true);
            }
            if (!map1.getState(i, j) && map2.getState(i, j)) {
                res.setState(i, j, true);
            }
        }
    }
    return res;
}

public static boolean[][] xorMap(boolean[][] map1, boolean[][] map2)
{
    boolean[][] res = new boolean[map1.length][map1.length];
    for (int i = 0; i < map1.length; i++) {
        for (int j = 0; j < map1.length; j++) {
            if (map1[i][j] && !map2[i][j]) {
                res[i][j] = true;
            }
            if (!map1[i][j] && map2[i][j]) {
                res[i][j] = true;
            }
        }
    }
    return res;
}

public static byte getAvailableNeighbours(int x, int y, int mapsize)
{
    byte availableNeighbours = 8;
    if (x == 0) {
        // Maybe corner
        if (y == 0) {
            return 3;
        } else if (y == mapsize - 1) {
            availableNeighbours = 5;
        } else {
            availableNeighbours = 3;
        }
    } else if (x == mapsize - 1) {
        // Maybe corner
        if (y == 0) {
            availableNeighbours = 5;
        } else if (y == mapsize - 1) {
            availableNeighbours = 5;
        } else {
            availableNeighbours = 3;
        }
    } else if (y == 0) {
        availableNeighbours = 5;
    } else if (y == mapsize - 1) {
        availableNeighbours = 5;
    }
    return availableNeighbours;
}

public static long diff(Map current, Map origin)
{
    long numWrong = 0;
    if (current.getMap().length != origin.getMap().length)
        return Integer.MAX_VALUE;
    int size = current.getMap().length;
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            boolean curr = current.getMap()[i][j];
            boolean orig = origin.getMap()[i][j];
            if (curr != orig)
                numWrong++;
        }
    }
    return numWrong;
}
public static long diff(Map current, Map origin, boolean[][] dontCares) {
    long numWrong = 0;
    if (current.getMap().length != origin.getMap().length)
        return Integer.MAX_VALUE;
    int size = current.getMap().length;
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++){
            if (dontCares != null && dontCares[i][j])
                continue;
            boolean curr = current.getMap()[i][j];
            boolean orig = origin.getMap()[i][j];
            if (curr != orig)
                numWrong++;
        }
    }
    return numWrong;
}

@Override
public Map clone() {
    boolean[][] mappy = new boolean[real_map.length][real_map.length];
    for (int i = 0; i < mappy.length; i++) {
        System.arraycopy(real_map[i], 0, mappy[i], 0, mappy.length);
    }
    return new Map(mappy);
}

public String RLEncode() {
    // output map as RLE, but only the data, no headers etc. and no $ signs...
    StringBuilder code = new StringBuilder();
    MapIterator mi = (MapIterator) this.iterator();
    boolean last = false;
    int count = 0;
    int maxCount = 0;
    boolean first = true;
    while (mi.hasNext()) {
        mi.next();
        if (count == 0) {
            if (!first)
                code.append(last ? "x": "o");
            else
                first = false;
            last = mi.get();
            count++;
        } else if (mi.get() == last) {
            count++;
            continue;
        } else {
            if (count > 1) {
                if (count > maxCount)
                    maxCount = count;
                code.append(count);
            }
            code.append(last ? "x": "o");
            last = mi.get();
            count = 1;
        }
    }
    if (count > 1) {
        if (count > maxCount)
            maxCount = count;
        code.append(count);
    }
    code.append(last ? "x": "o");
    System.out.println("RLE_length:"+code.length());
    System.out.println("RLE_maxCount:"+maxCount);
    return code.toString();
}

public void writeToFile(File f) {
    try {
        byte[] sendBuf = toByte();
        FileOutputStream fos = new FileOutputStream(f);
    }
}
7.4. MAP

```java
BufferedOutputStream bos = new BufferedOutputStream(fos);
bos.write(sendBuf, 0, sendBuf.length);
bos.close();
fos.close();
System.out.println("Map saved: "+sendBuf.length);
return;
}

try {
    byte[] sendBuf = toByte();
    byte[] output = new byte[sendBuf.length];
    Deflater d = new Deflater();
    d.setInput(sendBuf);
    d.finish();
    int l = d.deflate(output);
    FileOutputStream fos = new FileOutputStream(f);
    bos.write(output, 0, l);
bos.close();
    fos.close();
    System.out.println("Compressed Map saved: "+l);
    return;
} catch (IOException ex) {
    Logger.getLogger(Map.class.getName()).log(Level.SEVERE, null, ex);
}
System.out.println("Failed to save map!");

public void writeToCompressedFile(File f) {
    try {
        byte[] sendBuf = toByte();
        byte[] output = new byte[sendBuf.length];
        System.out.println(" going to save : "+sendBuf.length+"Bytes .. ");
        Deflater d = new Deflater();
        d.setInput(sendBuf);
        d.finish();
        int l = d.deflate(output);
        FileOutputStream fos = new FileOutputStream(f);
        bos.write(output, 0, l);
bos.close();
    }
```
```
7.4. MAP

```}

```{|}
| else  | for(byte r : ruleB){
|     if(neighbours == r){
|         next[i][j] = true;
|         break;
|     }
| }
| return new Map(next);
|} // Older version of code, saved for reference.

```}

```{|}
| MapIterator it = (MapIterator) current.iterator();
while(it.hasNext()){
    boolean alive = (Boolean) it.next();
    byte neighbours = it.getNeighbours();
    if(alive){
        for(int i = 0; i < ruleS.length; i++){
            if(neighbours == ruleS[i]){
                next[state(i, j, true);
                break;
            }
        }
    } else {
        for(int i = 0; i < ruleB.length; i++){
            if(neighbours == ruleB[i]){
                next[state(i, j, true);
                break;
            }
        }
    }
} return next;/**

```}

```{|}
| static long compare(Map current, Map origin) {
|     // if 100% match then MAX_VALUE;
|     // if less than 100%, then 10 points for each living cell that fits, -1 point for each cell wrong.
|     long hits = 0;
|     long misses = 0;
|     long wrong = 0;
|     long points = 0;
|     for(int i = 0; i < current.getMap().length; i++){
|         for(int j = 0; j < current.getMap().length; j++){
|             boolean curr = current.getMap()[i][j];
|             boolean orig = origin.getMap()[i][j];
|             if(curr & orig) { hi++;}
|             else if(curr & !orig) { wong++;
|             else if(!curr & orig) { miss++;}
|         }
|     } check if 100%.
|     if(misses == 0 && wrong == 0) {
|         points = Long.MAX_VALUE;
|     } else {
|         points = hits*10;
|         points = wrong;
|         points = misses;
|     }
|     return points;
|}

```}

```{|}
| private final boolean[][] real_map;

```}
7.4. MAP

```java
for (int j = 0; j < real_map.length; j++) {
    if (rnd.nextBoolean())
        real_map[i][j] = true;
}
}

public Map(byte[] bytes) {
    real_map = fromByte(bytes);
    population = this._getPopulation();
}

/* *
* Creates a Map from the given file, reading it byte for byte.
* The file MUST be divisible by eight, otherwise the behavior is
* undefined.
* @param filename
*/
public Map(File f) throws FileNotFoundException, IOException {
    InputStream is = new FileInputStream(f);
    long length = f.length();
    if (length > Integer.MAX_VALUE) {
        throw new IOException("File is too big!: " + f.getName());
    }
    if (length * 8 % 2 != 0) {
        throw new IOException("File size not divisible by eight!: " + length);
    }
    // Create the byte array to hold the data
    byte[] bytes = new byte[(int) length];
    // Read in the bytes
    int offset = 0;
    int numRead = 0;
    while (offset < bytes.length && (numRead = is.read(bytes, offset, bytes.length - offset)) >= 0) {
        offset += numRead;
    }
    // Ensure all the bytes have been read in
    if (offset < bytes.length) {
        throw new IOException("Could not completely read file: " + f.getName());
    }
    // Close the input stream
    is.close();
    real_map = fromByte(bytes);
}

/* *
* Returns a boolean[][] with all the elements of this Map. Can be used to
* copy maps (See: Map(boolean[][] initial_values)).
*/
public boolean[][] getMap() {
    return real_map;
}

/* *
* Creates an empty Map with the given size. Maps are always square.
* @param size of the map, so the map will contain size * size elements.
*/
public Map(int size) {
    real_map = new boolean[size][size];
}

public boolean getState(int x, int y) {
    if (x >= 0 && x < real_map.length) {
        if (y >= 0 && y < real_map.length) {
            return real_map[x][y];
        }
    }
}
```
7.4. MAP

```java
private boolean _getState(int x, int y)
|
  if(x >= 0 && x < real_map.length)
  if(y >= 0 && y < real_map.length)
    return real_map[x][y];
  return false;

public boolean _getState(long number)
|
  if(number < 0 || number > numCells() - 1)
    System.err.println("Map.java:_getState(strange number?");
  return false;
  double y = Math.floor(number/(0.0+real_map.length));
  long x = number - (real_map.length*(int)y);
  return _getState((int)y, (int)x);

public long numCells()
|
  return (real_map.length*real_map[0].length);

public boolean setState(int x, int y, boolean state)
|
  if(x >= 0 && x < real_map.length)
  if(y >= 0 && y < real_map.length)
    population += (state ? 1 : -1);
    real_map[x][y] = state;
    return true;
  return false;

public void toggleCell(int x, int y)
|
  if(x >= 0 && x < real_map.length)
  if(y >= 0 && y < real_map.length)
    if(real_map[x][y])
      population--; // skip the center
      real_map[x][y] = false;
    else {
      population++;
      real_map[x][y] = true;
    }
    return;

public byte getNeighbours(int x, int y)
|
  if(x >= 0 && x < real_map.length)
  if(y >= 0 && y < real_map.length)
    byte count = 0;
    for(int i = x-1; i <= x+1; i++)
      for(int j = y-1; j <= y+1; j++)
        if(i == x && j == y)
          continue; // skip the center
        if(_getState(i, j))
          count++;
    return count;

public int getPopulation()
|
  return population;
```

91
public int _getPopulation() {
    int count = 0;
    for (int i = 0; i < real_map.length; i++)
        for (int j = 0; j < real_map.length; j++)
            if (real_map[i][j])
                count++;
    return count;
}

//@SuppressWarnings("unchecked")
@Override
public Iterator<Bottom> iterator() {
    return new MapIterator(this);
}

7.5 ThreadedColourBacktracer

public Map run() {
    // Initiate..
    int numAlmost = 0;
    colorMap = new ColorMap(origin._getPopulation(), _ruleB, _ruleS, origin);
    if (this.startColors != null)
        // We want to resume?
        boolean could = colorMap.tryGoto(startColors);
    if (!could)
        System.err.println("Invalid startPosition!");
    return null;
    System.out.append("Starting from ",
        colorMap.prink());
}

boolean[][] donCares = null;
// Start rolling
long startTime = System.currentTimeMillis();
long lastPrint = 0;
byte[] last = null;
long exploration = 0;
long exploitation = 0;
ColorInstance best = null;
int nextval = 100;
boolean second = false;
byte size = (byte)origin.getMap().length;

// Agenda, a list of colorMap iterations worth checking out...
Agenda agenda = new Agenda();
AgendaKeeper keeper = new AgendaKeeper(colorMap, agenda);
Thread keeperThread = new Thread(keeper);
keeperThread.setPriority(keeperThread.getPriority() + 1);
// since its a slow process... give it its own thread so it can run uninterrupted (hopefully)...
keeperThread.start();

LinkedList<PMSolver> PMJobs = new LinkedList<PMSolver>();

while (ip.hasNext())
    ip.next();

for (int i = 0; i < nn.size(); i++)
    dots[i] = nn.get(i);
while(true)
{
    if(System.currentTimeMillis()-lastPrint > 4000)
    {
        System.out.println("Speed = "+(0.0+System.currentTimeMillis()-startTime)/1000);
        if(second)
            System.out.println("Agenda size = "+agenda.size() + " / "+numJobs);
        System.out.println("PMJobs size = "+PMJobs.size());
        colorMap.print();
        lastPrint = System.currentTimeMillis();
        if(second)
            System.out.println("Done!");
        break;
    }
    if(agenda.size() == 0 && PMJobs.size() == 0 && !keeperThread.isAlive())
    {
        break;
    }
    if(PMJobs.size() < 10)
    {
        // Create some jobs?
        if(PMJobs.size() == 0 && agenda.size() > 10000 && numJobs < 500000)
            numJobs *= 2;
        for(int i = 0; i < numJobs; i++)
        {  
            Byte[] cl = agenda.get();
            PermMap pl = null;
            Thread t1 = null;
            if(cl != null)
            {
                if(dontCares != null)
                    pl = new PermMap(cl, dots, size);
                else
                    pl = new PermMap(cl, dots, size, dontCares);
                if(staticMap != null)
                    pl.setStaticMap(staticMap);
                t1 = new Thread(pl);
                PMJobs.add(new PMSolver(pl, t1));
                t1.start();
            }
            else
            {
                synchronized(keeper)
                {
                    if(keeper.desiredAgendaSize < 500000)
                        keeper.desiredAgendaSize = 2;
                    keeper.notify();
                    break;
                }
            }
        }
    }
    else if(agenda.size() == 0 && numJobs > 50)
    {
        // Reduce the number of jobs
        numJobs /= 2;
    }
    if(!PMJobs.isEmpty())
    {
        // Any job finished?
        LinkedList<PMSolver> toRemove = new LinkedList<PMSolver>();
        for(PMSolver job : PMJobs)
        {
            if(!job.t.isAlive())
            {
                byte test = job.pm.runValue;
                toRemove.add(job);
            }
        }
        // Check if we got a solution here
        Map solution = job.pm.buildSolution();
        if(solution == null)
        {
            Map testMap = Map.nextGeneration(solution, ruleB, ruleS);
            long diff = Map.diff(testMap, origin, dontCares);
            if(diff == 0)
            {
                currentSolution = solution.clone();
                if(last)
                {
                    System.out.println("Found a solution ");
                    System.out.println("Here comes the solution "+cbytes);
                }
                return solution;
            }
            else
            {
                // print all the solutions
                System.out.println("Here comes the solution "+cbytes);
            }
        }
    }
}
7.6. REVERSIBLE CELLULAR AUTOMATA

```java
public void gotoNext() {
    // Rule to use is specified as two byte arrays. Here the rule used is B135/S024
    Map here = Map.nextGeneration(current, new byte[]{1, 3, 5}, new byte[]{0, 2, 4});
    for (int x = 0; x < current.getMap().length; x++) {
        for (int y = 0; y < current.getMap().length; y++) {
            boolean curr = here.getState(x, y);
            boolean prev = previous.getState(x, y);
            if (prev != curr) {
                next.setState(x, y, true);
            }
        }
    }
    previous = new Map(current.clone().getMap());
    current = new Map(next.clone().getMap());
}
```

7.6 Reversible Cellular Automata

Only most central methods included.

7.6.1 Second-Order CA

```java
public void gotoNext() {
    Map here = Map.nextGeneration(current, new byte[]{1, 3, 5}, new byte[]{0, 2, 4});
    for (int x = 0; x < current.getMap().length; x++) {
        for (int y = 0; y < current.getMap().length; y++) {
            boolean curr = here.getState(x, y);
            boolean prev = previous.getState(x, y);
            if (prev != curr) {
                next.setState(x, y, true);
            }
        }
    }
    previous = new Map(current.clone().getMap());
    current = new Map(next.clone().getMap());
}
```

7.6.2 Block Cellular Automata

```java
public static Map next(Map m, boolean odd, boolean reverse) {
    int length = m.getMap().length;
    Map tmp = new Map(length);
    int n = length/2;
    int row = 0;
    int column = 0;
    boolean[] here = new boolean[4];
    int sm = n+n;
    int r2 = 0;
    int c2 = 0;
    ```
7.6. REVERSIBLE CELLULAR AUTOMATA

for (int square = 0; square < nn; square++) {
    // iterates over bca squares..
    if (odd) {
        row = (int) Math.floor((0.0 + square) / n);
        column = square - (n * row);
        r2 = row + 2;
        c2 = column + 2;
        here[0] = m.getState(wrapMinus(c2, 3), wrapMinus(r2, 3)); // wrapMinus is a simple helper function defined below.
        here[1] = m.getState(wrapMinus(c2, 3), r2);
        here[2] = m.getState(c2, wrapMinus(r2, 3));
        here[3] = m.getState(c2, r2);
        here = critterBlock(here, reverse); // By changing the local rule the Block Cellular Rule will change
        //here = tronBlock(here); // Both the Critters and Tron local rules are defined below.
        tmp.setState(wrapMinus(c2, 3), wrapMinus(r2, 3), here[0]);
        tmp.setState(wrapMinus(c2, 3), r2, here[1]);
        tmp.setState(c2, wrapMinus(r2, 3), here[2]);
        tmp.setState(c2, r2, here[3]);
    } else { // even
        row = (int) Math.floor((0.0 + square) / n);
        column = square - (n * row);
        r2 = row * 2;
        c2 = column * 2;
        here[0] = m.getState(c2, r2);
        here[1] = m.getState(c2, r2 + 1);
        here[2] = m.getState(c2 + 1, r2);
        here[3] = m.getState(c2 + 1, r2 + 1);
        here = critterBlock(here, reverse);
        //here = tronBlock(here);
        tmp.setState(c2, r2, here[0]);
        tmp.setState(c2, r2 + 1, here[1]);
        tmp.setState(c2 + 1, r2, here[2]);
        tmp.setState(c2 + 1, r2 + 1, here[3]);
    }
}
return tmp;

/// Helper function, basically a decrement operator that wraps to the given maximum.
private static int wrapMinus(int cur, int max) {
    if (cur < 0) return max;
    return cur - 1;
}

7.6.3 Tron Local Rule

/// Tron local rule. Given the binary states of the 4 cells in a block,
/// the Tron local rule is applied and the states returned.
/// The Tron rule simply inverts all cells if all of them have the same state,
/// otherwise no change is made.
public boolean[] tronBlock(boolean[] input) {
    byte countOn = 0;
    for (byte i = 0; i < input.length; i++)
        if (input[i])
            countOn++;
    if (countOn == 0) // All cells dead -> inverted -> all cells live
        return new boolean[]{true, true, true, true};
    else if (countOn == 4) // All cells live -> inverted -> all cells dead
        return new boolean[]{false, false, false, false};
    else // no change
        return input;
}

7.6.4 Critters Local Rule

/// Critters local rule. Given the binary states of the 4 cells in a block,
/// the Critters local rule is applied and the states returned.
/// The Critters rule inverts all cells unless there are exactly 2 live cells.
/// If there are 3 live cells the block is also rotated 180 degrees (turned up-side down).
/// Note that in reverse the rotation is applied to blocks with 1 live cell instead of 3.
public static boolean[] critterBlock(boolean[] input, boolean reverse) {
    byte numAlive = 0;
    int rotaOne = 0; // for calculating the rotation
7.7. GACABacktracer

Only most central methods included.

7.7.1 Tournament Selection

```java
class GACAAlgorithm {
    public int tournamentSelection() {
        int numToCompete = 3; // only 1 victor
        int best = -1;
        long score = Long.MIN_VALUE;
        for (int i = 0; i < numToCompete; i++) {
            int current = (int) (Math.random() * population_size); // Selects a random gene to compete
            if (fitness[current] > score) {
                best = current;
                score = fitness[current];
            }
        }
        return best;
    }
```

7.7.2 Random Crossover

```java
class GACAAlgorithm {
    public Map randomCrossover(Map map1, Map map2) {
        Map offspring = new Map(map1.getMap().length);
        MapIterator oit = (MapIterator)offspring.iterator();
        MapIterator iter1 = (MapIterator)map1.iterator();
        MapIterator iter2 = (MapIterator)map2.iterator();
        while (oit.hasNext()) {
            boolean p1 = (Boolean)iter1.next();
            boolean p2 = (Boolean)iter2.next();
```
7.7. GACABACKTRACER

7.7.3 Edge Flip Mutation

```java
public Map edgeFlipMutate(Map map) {
    // Flip mutate only bits with neighbours
    MapIterator it = map.iterator();
    int x, y;
    while (it.hasNext()) {
        it.next();
        x, y = it.getXY();
        if (map.getNeighbours(x, y) > 0 && Math.random() <= mutation_chance)
            map.toggleCell(x, y);
    }
    return map;
}
```