2005

A Genetically Evolved Solution to the Firing Squad Problem

Brian J. Tajuddin

Master's Project: A Genetically Evolved Solution to the Firing Squad Problem

Brian J. Tajuddin

Jessica Bayliss, Ph.D., Chair

Peter Anderson, Ph.D., Reader

Chris Homan, Ph.D., Observer

Hans-Peter Bischof, Ph.D., Graduate Coordinator

Thesis/Dissertation Author Permission Statement

Title of thesis or dissertation: A Genetically Evolved Solution to the Firing Squad Problem

Name of author: Brian Tajuddin
Degree: MS
Program: Computer Science
College: GCCIS

I understand that I must submit a print copy of my thesis or dissertation to the RIT Archives, per current RIT guidelines for the completion of my degree. I hereby grant to the Rochester Institute of Technology and its agents the non-exclusive license to archive and make accessible my thesis or dissertation in whole or in part in all forms of media in perpetuity. I retain all other ownership rights to the copyright of the thesis or dissertation. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation.

Print Reproduction Permission Granted:

I, _____, hereby **grant permission** to the Rochester Institute Technology to reproduce my print thesis or dissertation in whole or in part. Any reproduction will not be for commercial use or profit.

Signature of Author: _____ Date: 11/18/05

Print Reproduction Permission Denied:

I, _____, hereby **deny permission** to the RIT Library of the Rochester Institute of Technology to reproduce my print thesis or dissertation in whole or in part.

Signature of Author: _____ Date: _____

Inclusion in the RIT Digital Media Library Electronic Thesis & Dissertation (ETD) Archive

I, _____, additionally grant to the Rochester Institute of Technology Digital Media Library (RIT DML) the non-exclusive license to archive and provide electronic access to my thesis or dissertation in whole or in part in all forms of media in perpetuity.

I understand that my work, in addition to its bibliographic record and abstract, will be available to the world-wide community of scholars and researchers through the RIT DML. I retain all other ownership rights to the copyright of the thesis or dissertation. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation. I am aware that the Rochester Institute of Technology does not require registration of copyright for ETDs.

I hereby certify that, if appropriate, I have obtained and attached written permission statements from the owners of each third party copyrighted matter to be included in my thesis or dissertation. I certify that the version I submitted is the same as that approved by my committee.

Signature of Author: _____ Date: _____

Contents

List of Figures

List of Tables

Abstract

In 1957, J. Myhill presented the firing squad problem. A special case of k-color cellular automata (CA) synchronization, the firing squad problem offers more stringent rules allowing for a provable minimal running time. To date, CA solutions have been found that run in minimal time using as many as sixteen states and as few as six [5]. There have also been arguments against the existence of solutions using only 4 states [11]. Due to the extremely large search space involved with such problems, the existing solutions have all been analytic in nature. We attempt to apply genetic algorithms and genetic programming to create transition tables that solve the firing squad problem. Ideally, the solutions would run in minimal time. No generalized solutions were found, but progress was made towards determining the best strategies for an evolved solution.

Acknowledgements

I would like to thank Jessica Bayliss, Peter Anderson, and Chris Homan for agreeing to be on my committee. Also, I would like to thank my parents for the encouragement they have given me over the course of my academic career. Without Sean, Greg, Noah, and everyone else in the AI lab, I would never have had my weekly comic relief. I would like to acknowledge Chris Orogvany who has consistently made fun of me from a distance. Finally, I would like to thank all of the professors who have worked so hard to teach my courses. There are far too many to name here.

Acronyms

ADF	Automatically-Defined Function
CA	Cellular Automata
EC	Evolutionary Computing
FSP	Firing Squad Problem
GA	Genetic Algorithms
GP	Genetic Programming
IC	Initial Configuration
MCP	Majority Classification Problem

Chapter 1

Introduction

The firing squad problem (FSP) demonstrates a very vivid case of emergent behavior in cellular automata (CA). In the existing solutions to the problem, multiple signals are used to communicate across the CA. As can be seen in Figure 1.1, the different signals move at different speeds to facilitate a divide and conquer solution to the problem. Similar signals have been evolved by evolutionary computing techniques for other problems [1, 8, 9, 11]. However, much of the previous work operated on two-state systems. FSP is necessarily a multi-state system. With this added difficulty, it is necessary to determine the best strategies in evolutionary computing for solving a problem like FSP.

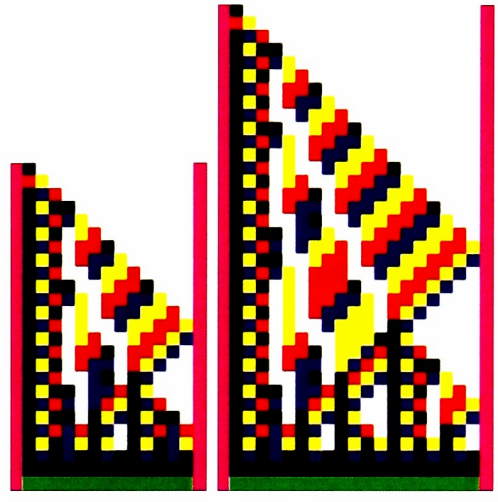
FSP appears to have a very low solution density [2] so a simpler CA problem was used to determine what evolutionary computing strategies might work best for CA problems. In this case, the majority classification problem (MCP) was attempted in its pure binary state as well as with multiple colors. We will now describe these problems in detail.

1.1 Problems

1.1.1 Majority Classification Problem (MCP)

Definition

MCP is a simple concept with no known solution CA solution for neighborhood smaller than half the length of the CA. For any initial configuration of the CA, all elements should relax to the value that is most common in the initial configuration. For example, an initial configuration where 60% of the input bits are `0` should have all output bits set to `0`. In a sense, the rule set for the elements must allow each element to converge on the overall majority value in the initial configuration of the CA. Traditionally, evolutionary computing systems have attempted to solve this problem using a neighborhood of seven, three neighbors on each side and the current bit. MCP is usually considered on a wrapped CA such that the right neighbors of the right-most element are the left-most elements. This reduces the number of necessary transitions and removes the need for boundary conditions [1].



(a) length of 13

(b) length of 19

Figure 1.1: The current 6-state minimal-time solution to the firing squad problem. Each column is a *soldier* in the firing squad. The time steps move from the top of the figure to the bottom, each row being a complete time step based only on the information in the previous step. The different colors represent the different states. White indicates the latent state, and green represents the fire state. In all, there are six states used by the soldiers (including the latent and fire states) plus one state used to show the boundary of the CA, shown in purple. The four unnamed states have no specific significance other than being intermediate steps towards the firing of the CA.

Brief Analysis

While this task is very simple for a human who can look at the entire CA, it is quite difficult if each bit's next state is only determined by its six closest neighbors and itself. No solution exists for the neighborhood of seven instance of this problem. Due to the amount of information that can be communicated in a nieghborhood of seven CA, it is unlikely that any solution for this problem exists for any neighborhood size smaller than half of the length of the CA [3]. The biggest obstacle for MCP solutions is density drift. If the initial configuration contains a group of adjacent bits as long as the neighborhood that have the minority value, those bits could expand to control the entire CA [3]. In such situations, a bias in the solution must become apparent (a mass of one value is preferred over a mass of the other) or the CA will not relax to a single state. If the solution is biased towards one value, the solution will only solve about half of the situations with dramatic density drift. If the solution does not converge, it does not solve any of them.

Existing Solutions

Despite the difficulties of MCP, there are four major solutions to MCP. The first three are hand-crafted solutions that have slowly built up the success rate by fractions of a percent. The earliest solution, called the GKL Rule, is simple and provides a solution for approximately 81.6% of possible initial configurations [1]. While it can be written out as the results of 128 transitions, its root is algorithmic. If the current bit is 0, its next state is the majority of itself, right neighbor one, and right neighbor 3. If the current bit is 1, it is the majority of itself, left neighbor one, and left neighbor 3. Thus, despite the fact that it is a neighborhood of seven solution, it only uses five of the neighbors. It also specifics a propagation direction for the two values. Two more hand-crafted solutions have improved slightly upon the GKL Rule [1].

The current best solution was created using genetic programming (GP), described later. This solution is not as simple as GKL algorithmically, but it exists as a table of 128 transitions that achieve approximately 82.326% accuracy on MCP [1]. Figure 1.2 shows an example of what a solved MCP instance may look like.

1.1.2 Firing Squad Problem (FSP)

Definition

FSP is best explained using the traditional analogy from which this problem received its name. There are soldiers standing in a line in a latent state. The latent state is special in that a soldier cannot leave it unless one of its immediate neighbors is not latent. In order to begin the process of firing, the leftmost soldier, referred to as the general, changes to a command state. At this point, the soldiers begin to change states based only on their immediate neighbors. The states to which they change are not defined or named. It is these transitions which we are trying to evolve. A solution is successful if all of the elements enter

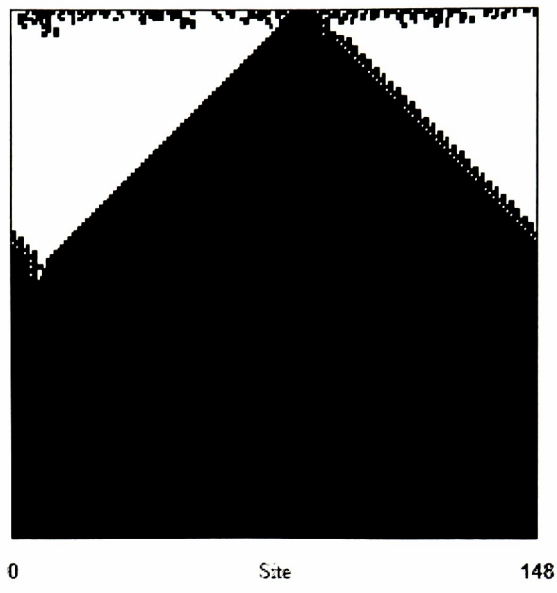


Figure 1.2: An example of MCP at work. [8] Each column represents a cell in the CA. Each row represents a time step. The initial configuration of the CA is at the top. As it progresses, the entire CA relaxes to the state that held a majority of the cells in the initial configuration.

the fire state for the first time simultaneously. The solution must also work on all numbers of soldiers larger than the neighborhood size [5].

Existing Solution

The current solution can be found in [5] and is illustrated in Figure 1.1. This solution exhibits a divide and conquer strategy towards solving FSP The most important feature of this solution is the types of signals that are sent. The first one moves as fast as possible across the line of soldiers. The second one moves much slower. The third through sixth signals move very slowly. This was the intention of the creator of this particular solution. If that is necessary for a complete solution for all numbers of soldiers, evolutionary computing may not be able to achieve that level of sophistication.

Mazoyer has worked on solutions using fewer states [6]. He presents a three state solution that does not follow the standard Minsky-like representation. It uses messages as well as states, so the elements can send different signals in different directions while only taking on three distinct states. Balzer [2] argued and eventually Yunès [11] proved that a solution with four states is not possible. Balzer also argued that any solution less than 8 states was unlikely, but he restricted the search space handled by the algorithm using rules that limited the definition of the problem. Balzer's system ran until it found five 8-state solutions that fit his restricted definition of FSP. Balzer's hypothesis was contradicted by Mazoyer's discovery of a six-state solution.

1.2 Evolutionary Computing (EC)

Evolutionary computing has many flavors. Some of them are better suited to certain tasks. Since MCP and FSP require a transition table, the result would be a transition table or an algorithm to determine the transitions in the CA. The simplest representations for this purpose would be strings[1] and simple programs. Thus, genetic algorithms (GA) and genetic programming (GP) can be used for these problems.

While GA and GP differ greatly in most ways, they still provide certain genetic operators that work on the same concept. The operators are selection, mutation, and recombination [9]. These three operations allow the evolutionary computing system to develop genes with higher fitnesses.

Selection

Selection is the method by which unfit individuals are removed from the population. Without selection, there is little to encourage the fitness of the population to go up. There are several types of selection, but the two most common methods are tournament selection and proportional selection.

Tournament selection occurs by selecting at least two members from the population and choosing the best individual in that group. In the case of a generational system, the top member will be used to breed the members of the next generation. In a static population system, the best member will be one parent and the worst member will be the location where one child will be placed. In this model, the probability of a single member being selected is based on the probability of it being chosen in a tournament and its fitness as compared to the other members selected in that tournament. As a result, the worst member of the population has no chance of being selected unless there are several members with the lowest fitness value [7].

Proportional selection bases selection on the fitness value as compared to the sum of all fitness values in the population. If we assume that all fitness values are positive, then the probability of any specific member of the population being selected is

$$\frac{f}{\sum_{i=1}^{n} f_i}$$

Using the proportional method, every member of the population has the possibility of being chosen [7]. If the fitnesses are not all positive, as is the case for FSP, the minimum fitness can be added.

$$\frac{f + minFit + 1}{\sum_{i=1}^{n} f_i + minFit + 1}$$

Also explored in this work is elitism. Elitism simplistically selects the best members of the population [7]. From observations, this reduces the genetic diversity of the population. As a result, the power of recombination is weakened and mutation becomes the only means to escape from a fitness locus.

[1] Bit strings or strings of integers

There are other options for selection methods, but these are the most common. All selection methods pose potential problems. In some cases, elitism converges far too quickly to find a useful solution and diversity in the population can be lost. On the other hand, proportional or tournament selection can allow for too much diversity, causing numerous incompatible partial solutions to clash and postpone any sort of conversion. The ideal selection method will preserve a certain amount of diversity while favoring the higher fitness and converging towards a solution.

Mutation

Mutation is the process of randomly changing parts of the genetic material in order to allow for new possibilities in the gene pool. Mutation takes on very different forms in different types of evolutionary computing, so it will be discussed in more detail later.

Mutation helps evolutionary computing increase the diversity of the population slightly, allowing the population to escape from a locus or travel along a neutral network. Without mutation, possible values for certain points in the gene can be lost with selection and recombination. This can cause the system to converge on a locus that may not be optimal. In the absence of neutral networks, mutation may be the only way to escape from such a situation [7].

Recombination

In order to create new combinations of genes, the evolutionary computing methods must take two or more parents and create children that are a mix of genes from more than one parent. Crossover also depends on the type of evolutionary system being used. The details of crossover will be discussed in that context as well.

Ideally, recombination will allow multiple individuals scattered throughout the search space to combine and produce offspring that are better than some or all of the parents.

Fitness

The core of any evolutionary computing method is the fitness function. This function usually provides a quantitative evaluation of the individual's phenotype. Some systems require user input in the ranking of individuals. In either case, the fitness of the individual is a method of ordering the population and determining which individuals are the most fit.

The fitness calculations usually take the bulk of the running time. While evolutionary computing is usually used to solve difficult problems that cannot be solved using hill-climbing or other simple techniques, a good fitness function makes the problem more like a hill-climbing problem. It also gives the EC system an idea of how much of the problem is solved since the fitness function is very dependent on the features of the solution. That is not to say that the solution must be well-defined. For example, if an EC system is used to solve MCP, the fitness function can be adjusted to favor solutions that tend to run faster. In this case, there is no definition as to the minimum running time of an MCP solution, but

Index:	0	1	2	3	4	5	6	7
Input:	000	001	010	011	100	101	110	111
Parent 1:	0	1	2	3	4	5	6	7
Parent 2:	8	9	10	11	12	13	14	15
Child 1:	0	1	2	11	12	13	14	15
Child 2:	8	9	10	3	4	5	6	7

Table 1.1: An example of a one-point crossover

EC can work towards shorter solutions without the exact dimensions of that solution being defined. For MCP and FSP, time is not a concern. Maximum time limits are enforced, but such restrictions are for time conservation rather than constraining the solution.

1.2.1 Genetic Algorithms (GAs)

Genetic algorithms operate on bit strings. In this case, bit strings are simply arrays of integers. GA works well on simple integer arrays as well as permutations. Most integer-based problems can be attempted using GAs. In the case of MCP, the string can be an array of 0 and 1 that represent the transition table for the CA. The specific representations used for MCP and FSP can be discussed in detail later. The nuances of using GA should now be approached.

When an EC is started, the members of the population are all randomly generated. This strategy allows for a large degree of genetic diversity from which to start. From this random initialization point, the three evolutionary operators can be applied to the population until some sort of exit condition is reached.

Mutation can take on several forms in GA. The most common is a simple bit-by-bit probability of mutation. The process is relatively simple. As the mutation system iterates through the string, each bit (it could be an integer) is randomly mutated to another value with a probability specified by the user or the system [7]. In a binary system, it is simple to just flip the bit. Integer-based systems allow for many more possibilities.

Crossover is a slightly more complex topic. There are many types of crossover in GA. The simplest version is called one-point crossover. In this crossover, a random point in the string is chosen as the cut point. The children are combinations of two parts from two different parents. This crossover is very simple, but it does not show any understanding of the problem to be solved. An example of one-point crossover can be seen in Table 1.1. Other crossovers exist that use varying numbers of cut points, but one and two cuts points are most common [7].

The next common crossover technique is uniform crossover. For the child, each bit is taken from a random parent on a bit-by-bit basis. While the one- and two-point crossovers can preserve related bits. Uniform crossover has no sense of related bits being transferred together. If the bits are at all related to each other, the uniform crossover has the potential

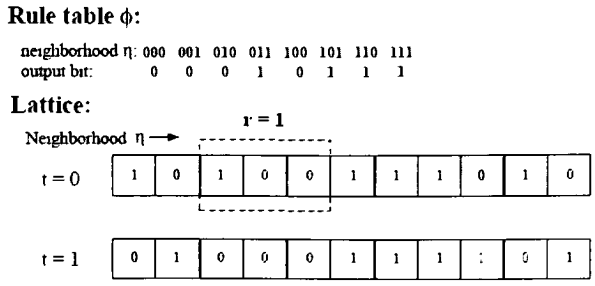


Figure 1.3: Neighborhood selection and the reading of a transition from the bit string representation. The neighborhood, n, is of size three and the radius, r is of size one. When the input is 100 at $t = 1$, the input is translated to index 4 in the gene and the new value of the element is 0 at $t = 1$. [8]

to destroy relations in the gene [7].

The final crossover type is more of a category. A schema cross is based on some sort of schema for the gene. If the values of two bits in different parts of the string directly affect the fitness of the gene, they should be kept together. In this way, the schema cross can preserve related pieces of the gene. It is based on this concept that the author has created two new crossover techniques for CA problems. In order to understand these new crossover techniques, it is necessary to understand how the transition table for CA problems is represented in a bit string.

Transition Table Representation

The transition table is stored in an array of integers. The value of the string at a certain index is the output of the transition table. The index itself represents the input to the transition table. In order to illustrate, assume that there is a binary CA problem with a neighborhood of three (or a radius of one). In order to perform a lookup, take the inputs and translate them to an unsigned decimal and that is the index to use. For example, an input of 101 refers to an index of five in the string. Figure 1.3 shows an example of the neighborhood and the index-to-input translation. This generalizes well to CA problems with more than two states. The radix changes to the number of states in the CA [7].

Radix Cross

The radix cross splits the string into two pieces based on a randomly chosen input. Use the example 101 again. The crossover selects a random input, say the first one. Thus, the transitions for any input whose left neighbor is 1 will be in one group, and the rest of the string will be in the other group. In this case, one group would be 100, 101, 110, and 111. The other group would be 000, 001, 010, and 011. Like the one-point crossover, the child is the product of one group from one parent and the other group from another parent. Another example can be seen in Table 1.2.

Index:	0	1	2	3	4	5	6	7
Input:	000	001	010	011	100	101	110	111
Parent 1:	0	1	2	3	4	5	6	7
Parent 2:	8	9	10	11	12	13	14	15
Child 1:	0	1	10	11	4	5	14	15
Child 2:	8	9	2	3	12	13	6	7

Table 1.2: An example of the radix crossover in a two-state system where the important bit is the middle input

Index:	0	1	2	3	4	5	6	7	8	9	...
Input:	000	001	002	010	011	012	020	021	022	100	...
Parent 1:	0	1	2	3	4	5	6	7	8	9	...
Parent 2:	27	28	29	30	31	32	33	34	35	36	...
Child 1:	0	1	2	30	31	32	6	7	8	9	...
Child 2:	27	28	29	3	4	5	33	34	35	36	...

Table 1.3: An example of the radix crossover in a two-state system where the important bit is the middle input

The example in Table 1.2 does not properly demonstrate the full function of this crossover on a multi-state CA. FSP is currently solved with six states. Thus, each input can take on values from zero through five. An example of a radix cross in a three-color system is shown in Table 1.3 Instead of choosing a single value to make the groups, half of the values are chosen for one group. With this selection scheme, the two groups take half of the string for each of them. Restriction of the size of the groups reduces the flexibility of the EC system, but the crossover seems to understand the problem more than a blind one-point or uniform crossover.

Hamming Cross

The other crossover created for CA problems involves selecting groups on a hamming distance. The hamming distance of two binary strings is the number of bits that differ in the strings. The hamming cross extends that same concept to integer strings. The first step of the crossover is to choose a random hamming distance as the boundary. This value can be anywhere between zero and the length of the string, inclusive. Since the hamming distance is calculated on the potential inputs to the transition table, the length is not that of the bit-string gene, but it is the size of the neighborhood. The second step is to choose a random input. This can be done by selecting a random index in the gene and encoding it into the input it represents. That input represents the reference string for the hamming cross. All inputs within the randomly generated hamming distance of the reference string will be part

Index:	0	1	2	3	4	5	6	7
Input:	000	001	010	011	100	101	110	111
Parent 1:	0	1	2	3	4	5	6	7
Parent 2:	8	9	10	11	12	13	14	15
Child 1:	0	9	2	3	12	13	6	15
Child 2:	8	1	10	11	4	5	14	7

Table 1.4: An example of the hamming crossover where the reference point is the index 2 and the hamming distance is 1

of one group. The rest of the strings will be part of the other group. The size of the two groups can vary greatly. In order to avoid cloning members of the population, the maximum hamming distance is usually restricted to the size of the string minus one. That avoids everything becoming part of a single group. A demonstration is shown in Table 1.4.

The hamming cross also demonstrates a greater understanding of the how the representation works. Unlike the radix cross, it also shows a flexibility of size, more like the classic crossover techniques. Through the exploration of MCP and FSP, tests were run to determine if these new crossover techniques provide any advantages over the simple, classic crossovers.

1.2.2 Genetic Programming (GP)

Genetic programming takes a different approach to representing problems. Instead of using a bit-string to represent a transition table for the CA, it attempts to write a logical program to determine the outputs. There are multiple types of GP that use different languages for the evolved programs. The most common type is lisp-based and was pioneered by Koza [4]. This representation is particularly useful in that it is tree-based. In previous work, this method was used to work on MCP [1]. That work had the distinct advantage of being binary in nature. In order to properly solve FSP, a multi-state CA is required. This fact forces GP to use a wider set of constants and operators to work on the problem.

Another useful feature of GP is called automatically defined functions (ADFs) [4]. ADFs attempt to solve sub-problems in order to facilitate the solution to a much larger problem. They have been proven to help on large parity problems by solving smaller parity problems and combining the sub-solutions to build the larger result [4]. In order to avoid halting issues, ADFs are only allowed to call lower numbered ADFs. For example `ADF0` cannot call itself and it cannot call `ADF1`. On the other hand, `ADF1` can call `ADF0`. The argument count for the ADFs is usually statically set and is shown in Table 1.5.

For lisp-based GP, the mutation operation involves destroying a random node on the tree and randomly generating a replacement for it. In this way, mutation can be catastrophic if certain nodes are selected. Unlike GA where the mutation of any location in the gene has equal effect on the gene, most mutations in GP will change other nodes and may even change the size of the gene. Many GP systems do not use mutation due to the already

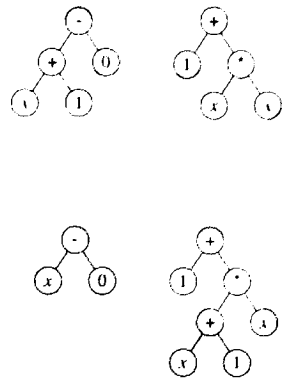


Figure 1.4: Single-point crossover with functional programs. The top trees are the parents, and the bottom trees are the children.

diverse population involved. A large number of randomly generated programs span a wide variety of structures and values. Mutation may not provide much more than a system that will never begin to converge [4].

GP crossover is a relatively simple tree operation. Assuming two parents, a node is selected in each parent and they are swapped. See Figure 1.4 for an example. The nodes could be in the main program or the ADF. In the event that the nodes are in different parts of the program (i.e. one is in the program and one is in ADF1), a verification process must be pursued. Usually any nodes that are not allowed in a specific part of the program (i.e. a call to ADF1 in ADF0) are randomly replaced with newly-generated nodes [4].

As mentioned above, there are several types of GP. Another type uses a stack-based language called Push3 [10]. This language uses type-based stacks to perform the operations. Unlike the lisp-based GP, the programs are sequences of commands and constants. This allows for GA-type mutation and crossover of a Push3 program. Due to lack of simple Push3 library and time constraints, Push3 was not used to attempt MCP and FSP. It also provides a much larger search space for the EC system which would require much longer runs.

Unlike GA, GP allows for a more algorithmic approach to solving CA problems. While it may take a while for GA to evolve the GKL Rule for MCP, that is the sort of algorithmic result that GP could create. The primary advantage of GP in CA problems is the flexibility for using neighbors. In a GP system, a neighbor can be ignored by changing a node or two. In GA, several bits would have to be considered together in order to duplicate that sort of behavior. In the same respect, GA can make fine adjustments to a transition table while GP may have to add another branch to the tree.

As stated earlier, the GP system for a multi-color problem requires more operators. Table 1.5 shows all the operators, their argument types, and their return types. The overall program is required to return a numeric value and take numeric values as arguments. The ADFs follow the same rules. The argument types specified here only apply to randomly-generated nodes. Once crossovers occur, `0` is equivalent to `false`. All other values are

Operator	Argument Types	Return Type
AND	boolean, boolean	boolean
OR	boolean, boolean	boolean
NOT	boolean	boolean
NAND	boolean, boolean	boolean
NOR	boolean, boolean	boolean
XOR	boolean, boolean	boolean
ADD	integer, integer	integer
SUB	integer, integer	integer
MULT	integer, integer	integer
DIV	integer, integer	integer
MOD	integer, integer	integer
POW	integer, integer	integer
IF	boolean, integer, integer	integer
EQ	integer, integer	boolean
GT	integer, integer	boolean
GTE	integer, integer	boolean
LT	integer, integer	boolean
LTE	integer, integer,	boolean
ADF0	integer, integer	integer
ADF1	integer, integer, integer, integer	integer

Table 1.5: The operators and their signatures used in the GP system

equivalent to `true`.

Chapter 2

System Implementation

The EC system used for this work was written in Java in order to leverage polymorphism and the simple distribution afforded by remote method invocation (RMI). There are two main components to the system: the solver and the problem. The generalized architecture is shown in Figure 2.1.

2.1 Solver

The solver controls the population and applies the evolutionary operators as they are configured. It can also be a non-evolutionary solver, applying a classic algorithm to the problem rather than a genetic approach. The primary solver used in this work was a generational genetic solver. Once a result is retrieved from the generational solver, it can be put through a hill-climbing solver to perform a local search on the solution. This allows for a slightly better solution to be found without the evolutionary system finding the absolute local optimum by itself. The following is a brief summary of how these solvers operate.

2.1.1 Generational Solver

The generational solver operates on a population by choosing elites to create the next generation. There are several minor variations on this type of system, but the general approach is the same.

The first step in any evolutionary system is to randomly generate the initial population. The problems generate the random string themselves to allow for polymorphism between GA and GP problems. This will be discussed more in the next section. To simplify the implementation, the solver does not actually generate an entire population. Instead, it generates a certain number of *elites*. The elites do not really live up to their name in the first generation, but they will in later generations. From the original elites, the rest of the population is the product of recombination and mutation of randomly selected pairs of elites. When all of the children have been created and their fitnesses evaluated, the generation is considered to be complete. The chosen selection algorithm (usually a tournament) selects

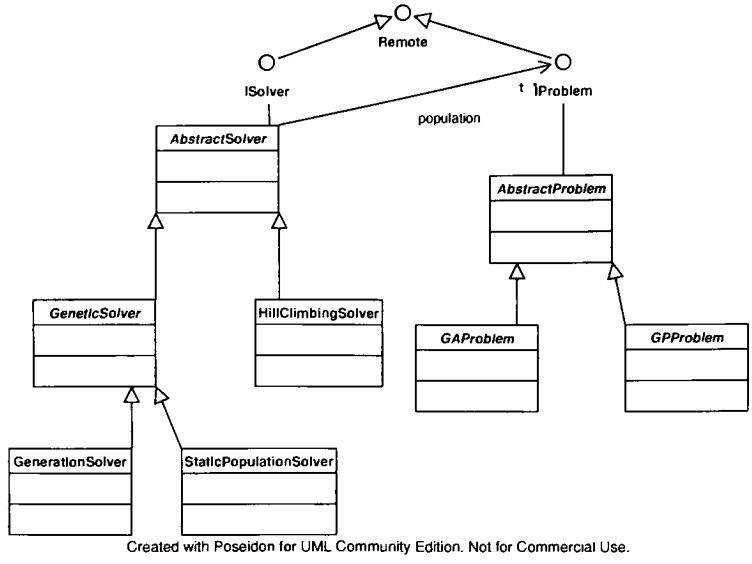


Figure 2.1: Framework architecture

the elites for the next generation and the cycle continues.

For some problems, there are randomly generated values that need to be re-generated every generation in order to train the solutions on a larger space. Such is the case for MCP where there are a very large number of initial configurations. To facilitate this, the solver tells the problems to recreate the information between generations. Since the initial configurations for MCP change every generation, it is essential to re-evaluate the elites. By default, the solver tells all of the elites to recalculate their fitnesses at the beginning of each generation. In the case of FSP, this would be wasteful, so the problem itself provides logic to ensure that unnecessary fitness evaluations are avoided.

At the end of the specified number of generations, the generational solver outputs information about the best member of the population. It also performs a selection on the last generation and outputs information about all of the elites selected. From this information, it is quite simple to run another test using the output from a previous test as a starting point.

2.1.2 Hill-Climbing Solver

Each problem has the ability to return its neighbors. In the case of GA problems, this simply involves returning all genes that differ in a single position. In GP, this is not defined. The solver takes the neighbors and evaluates their fitnesses. If one of them is better than the original, it becomes the new hero and its neighbors are found. This continues until a cycle goes by without a new hero emerging.

At the end, the last hero is output for use in other runs or for informational purposes. Since the problems addressed in this work are not hill-climbing problems, the hill-climbing solver provides limited use. It is simply provides a local search on solutions produced by the

generational solver by systematically changing a single element of the gene at a time.

2.2 Problem

The problem has several responsibilities in the system. Anything that can change depending on the specifics of the problem needs to be handled here. Behavior specified by the problem includes: random generation, crossover, mutation, and fitness evaluation as well as related activities. In order to facilitate more flexibility, the problems also implement behavior allowing recall of genes from previous runs.

Since this work uses multiple strategies to solve MCP and FSP, the problems are simply wrappers for a simulator. Each problem instance has a simulator instance. The simulator can call the problem instance as a lookup table. Given an input for a specific cell, the problem will decode the gene for that input and return the output. With this design, all problem instances that work on MCP have the same simulation and fitness evaluation routines.

Chapter 3

Majority Classification Problem (MCP)

Before discussing the current work on MCP, Table 3.1 shows the four best solutions that currently exist [1]. Currently, the best solution was created by a GP system. It is important to note that this solution was created using only boolean operators. Since the goal of the work presented here is to solve FSP, the GP system is allowed a much larger set of operators.

3.1 Representations

For MCP, the representations are relatively standard. The GA gene represents the transition table as discussed earlier. Since the CA is wrapped (there are no edges), the decoding of input into an index in the gene is quite simple. The length of the string is dependent on two variables. The first variable, k, is the number of states that each element of the CA can have. MCP is usually binary, so $k = 2$. The other parameter is the size of the neighborhood, n. Usually, this is expressed in terms of radius, r, as $n = 2r + 1$. In Figure 1.3, $r = 1$. For most work done on MCP, $r = 3$. From these two parameters, the size of the transition table can be represented as k^{2r+1}. All of the solutions shown above as well as the binary solutions produced here are 128 bits long. In all solutions, produced by GP and GA, the input of all zeroes or all ones produces zero or one respectively. This forced value has been inserted because it is a necessary feature of an MCP solution and it reduces the size of the search space.

GP has already been proven to be useful in this problem [1]. However, as mentioned earlier, previous attempts limited the instruction set to boolean operators only. In order to properly train it to work on FSP, more operators are necessary. FSP is never a binary problem as MCP can be. In order to compensate for this difference, several mathematical and conditional operators were added. For the binary operators, zero is assumed to be `false` and all other values to be `true`. This was done to avoid type issues.

After test were run on the pure binary MCP, it was tried with more colors. The initial configuration remains binary, but the solution is allowed to explore more colors. The

Name	Year	Method	Bit String	Success Rate
GKL	1978	hand	00000000 01011111 00000000 01011111 00000000 01011111 00000000 01011111 00000000 01011111 11111111 01011111 00000000 01011111 11111111 01011111	81.6%
Davis	1995	hand	00000000 00101111 00000011 01011111 00000000 00011111 11001111 00011111 00000000 00101111 11111100 01011111 00000000 00011111 11111111 00011111	81.8%
Das	1995	hand	00000111 00000000 00000111 11111111 00001111 00000000 00001111 11111111 00001111 00000000 00000111 11111111 00001111 00110001 00001111 11111111	82.178%
Koza	1995	GP	00000101 00000000 01010101 00000101 00000101 00000000 01010101 00000101 01010101 11111111 01010101 11111111 01010101 11111111 01010101 11111111	82.326%

Table 3.1: Current solutions for MCP [1]

only requirement is that it converges to whatever color was in the majority in the initial configuration.

3.2 Fitness

The fitness of MCP is relatively simple. A simulation is run on an initial configuration. This simulation is run until the time limit is reached or the CA does not change. When the simulation stops, the number of elements of the CA in the proper state is the first element of the fitness. The second element of the fitness is a bonus in the event that the tested transition table solves the initial configuration perfectly.

This fitness function differs from the fitness functions used in previous work [1, 8]. Previously, the fitness function has simply been the number of initial configurations in the training set that were solved perfectly using the individual. Since the ultimate goal of this work is to solve FSP and it is highly unlikely that a solution to a single instance could be randomly generated, such an approach will not work. Therefore, partial credit is used for MCP to provide a more useful stepping stone.

If a potential solution returns an invalid value, the fitness contribution for that specific initial configuration is zero. This situation can only occur in GP systems, but it is important to note the influence of such behavior on the fitness of the individual.

Parameter	Description	Value
Population Size	The number of individuals that exist in a single generation	100
Elites	The percentage of individuals selected as the elites from which to create the next generation	20%
Selection Method	The type of selection used to choose the elites from the population	elitism, tournament, and proportional
Preserve Best	Whether or not to preserve the best member of the population for non-elitism selection methods	`true`
Time Limit	The number of generations to run before stopping	100
Random Seed	The seed for the random number generator	randomly typed values

Table 3.2: The parameters for the generational solver running MCP

3.3 Parameters

There are many parameters involved in an EC system. For this problem, there are two things to parameterize: the solver and the MCP simulator. Table 3.2 displays the solver parameters, a brief description, and how it was set for MCP.

At the solver level, the selection method is the primary variable to be tested. In theory, all of the selection methods have merits. The tests of these methods simply serve to pick out which selection types work best for the specific problem and representation.

There are many more MCP parameters and most of them are modified in the course of this work. Those that have a variety of values will be marked as *variable*. The parameters are shown in Table 3.3. Several parameters for MCP matter in the realm of MCP only, but they do not hold much use when considering MCP as a stepping-stone to FSP. For this reason, some of the parameters should be set to lighten the processing load and remove variables that are unnecessary.

First, FSP only has one initial configuration. This would be a trivial problem for MCP, so several initial configurations (IC) will be used. However, the same set of IC can be used for all generations instead of generating a new set each time. In order to preserve some difficulty while limiting processing power, twenty initial configurations can be used to train the populations.

It should be noted that keeping the same initial configurations for each generation may allow for overfitting of the transition tables to the training set. In other work [1, 8], the initial configurations have been recreated at each generation.

The CA length can be limited to a point where density drift is an issue but small enough

Parameter	Description	Value
Wrapped	Whether or not the CA is wrapped	`true`
Length	The length of the CA	31
Steps	The number of steps to simulate the CA before stopping	120
Initial Configurations	The number of initial configurations of the problem to use	20
Radius	The radius of the CA neighborhood	3
Threshold	The bias on the random number generator for creating the initial configurations	0.6
Bonus Multiplier	When multiplied by the length, the bonus for solving an initial configuration perfectly	0 or 1
Same Initial Configurations	Whether or not to keep the same initial configurations for the entire run	`true`
Cross	Which crossover technique to use	*variable*
Values	The number of colors used in the CA	*variable*
Mutation Rate	The mutation rate of the evolutionary system	*variable*

Table 3.3: The parameters for MCP

to allow for faster calculations. Therefore, a short CA of length 31 makes calculations faster while still allowing for the shortcomings of a CA solution to the problem to appear. Similarly, the majority threshold has been set to 0.6 to simplify the calculations and make density drift slightly less likely. These changes are tuned to balance the ability to make progress in solving MCP and the amount of time it takes to run the evolutionary system.

The random generation of ICs, with or without bias, creates a binomial distribution of the densities of each value in the IC. A 0.6 bias has been used to here to reduce the number of density drift issues and difficult configurations that would arise using a fair random number generator. Since larger densities of the majority value allow for simpler solutions, uniform distributions of densities have also been used as a starting point [3]. Over the generations, the uniform distribution was slowly converted to a binomial distribution. This strategy never produced individuals that solved more than 80% of the training sets. Previous work has shown that the four solutions presented above solve more than 80% of the testing set [1]. Since no benefit is achieved from uniform distributions, a biased binomial distribution is used here.

The remaining variables provide the search space for this work. The real benefit that might be revealed by MCP lies in the crossover types. If one crossover type appears to emerge as better than the others, it provides some insight into what kind of crossover might work best for CA problems in general. Of course, the overall purpose of attempting MCP is to determine the best combination of crossover, selection, and fitness parameters for FSP.

3.4 Results

The results of the evolutionary attempts at MCP can be split into the two different strategies. Due to the very small populations and short runs of these systems, the results are not very precise or impressive compared to the existing solutions. For informational purposes, the existing solutions were tested on the same ten thousand initial configurations as all of the ultimate heroes. The success rates can be seen in Table 3.4. Most of the solutions discovered in previous work gain an extra ten percent accuracy with the weighted random initial configuration generator. This is due to the reduced probability of disruptive density drift in the initial configuration. Since there are fewer minority values in the CA (less than 40% of the initial configuration is minority on average), an entire neighborhood composed of the minority is much less likely than if the initial configuration were generated with an equal probability for each color.

3.4.1 GA Results

The goal is to determine which strategies work best and use those to gain some more information about multi-color system. The two-color test run results can be seen in Table 3.5. Now, it is possible to break down the average success rate of each parameter and determine if any strategy is overwhelmingly more efficient than any other.

Solution	Success
GKL	91.55%
Davis	88.98%
Das	91.85%
GP	90.58%

Table 3.4: The success rates of the old MCP solutions as tested with the solutions presented here

Colors	Cross	Selection	Bonus	Success
2	one-point	proportional	yes	30.70%
2	one-point	proportional	no	0%
2	one-point	tournament	yes	24.54%
2	one-point	tournament	no	21.98%
2	one-point	elitism	yes	41.60%
2	one-point	elitism	no	44.25%
2	radix	proportional	yes	32.40%
2	radix	proportional	no	28.39%
2	radix	tournament	yes	10.53%
2	radix	tournament	no	18.96%
2	radix	elitism	yes	22.41%
2	radix	elitism	no	46.06%
2	hamming	proportional	yes	15.55%
2	hamming	proportional	no	16.78%
2	hamming	tournament	yes	11.35%
2	hamming	tournament	no	9.67%
2	hamming	elitism	yes	10.59%
2	hamming	elitism	no	15.68%

Table 3.5: The results from the MCP tests using GA and the parameters as specified in the table

Strategy	Average Success Rate	Standard Deviation
One-point crossover	27.18%	16.03%
Radix crossover	26.46%	12.24%
Hamming crossover	13.27%	3.07%
Proportional selection	20.64%	12.38%
Tournament selection	16.17%	6.46%
Elitism selection	30.10%	15.72%
Perfection bonus	22.19%	11.13%
No bonus	22.41%	15.11%

Table 3.6: A comparison of the different strategies (crossover, selection, and bonus) used in the GA and the average success rates of the heroes they produced

Table 3.6 shows the strategy breakdown and the average success rates of those strategies. For the crossover, it determined that the hamming cross is not terribly effective. The one-point and radix crosses performed about the same in the short tests. The selection strategies did seem to make a difference. Surprisingly, elitism beat out proportional selection and tournament selection by a sizable margin except that the standard deviation is quite large. This is possibly due to the short length of the EC test run. Elitism will tend to converge to a higher value in early generations but will make very little progress after that point. Other selection methods will take longer to converge, but they will tend to have higher end fitnesses due to the diversity in their populations. Finally, the bonus for perfectly solving a single initial configuration seemed to have no effect. It produced the highest and the lowest success rates, so it seems to be hit and miss.

At any point, some of these numbers could change. However, they will be used as a guide in multi-color examples where the initial configurations are still binary, but the system is allowed to use more colors. In order to reduce the number of runs necessary, four-color tests can be run using the two best crossovers, elitism for selection, and bonuses will be tested both on and off. Unfortunately, the results of the four-color tests are very poor. Even on a relatively simple CA length of 31, none of the four-color tests solved any initial configuration perfectly in one thousand generations.

While the multi-color test results are disappointing, they are not completely unexpected. The gene for a two-color MCP problem is 128 bits long for a search space of 2^{128} The gene for a four-color MCP problem is 16,384 integers long for a search space of $2^{32,768}$. The disparity in the size and search space of the genes is the main cause for the poor results in such a limited test. The longer gene requires a larger population and more generations to produce a proper solution. The advantage lies in the fact that a multi-color MCP solution could handle larger density drift situations. Since the time and computing power to achieve this are not available and MCP is not the goal of this work, the results of MCP have to be considered less useful in the ultimate pursuit of a FSP solution.

Colors	Selection	Bonus	Success
2	proportional	yes	50.18%
2	proportional	no	0%
2	tournament	yes	50.18%
2	tournament	no	0%
2	elitism	yes	49.26%
2	elitism	no	0%

Table 3.7: The results from the MCP tests using GP and the parameters as specified in the table

3.4.2 GP Results

The GP system takes an extra parameter that has not been previously specified. This parameter is the depth of the trees that form the genetic programs. For the tests that produced the results presented below, the maximum depth was ten. While this limit may not be terribly restrictive as far as the number of nodes, it almost eliminates the possibility of an evolved program that has a case for each input.

The results of the tests on GP are shown in Table 3.7. The perfection bonus certainly made a difference in this system. However, it is important to notice that the solutions that were evolved with the bonus are very poor solutions. For the most part, they always converge to the same value. This may provide better success rates than the GA produced in the same amount of time, but they provide less versatile solutions. If we consider the fitness without the perfection bonus of the best GA solutions above and the best GP solution, the GA solution has the higher fitness because more than half of the cells were in the proper state at the end of the simulation. The best GP solutions were not significantly better than half and only surpassed the halfway mark because the initial configurations were randomly generated.

The best results of this test may have returned all `0` or `1`, but they are not that simple. An example of the solutions produced by the GP system is shown in Table 3.8. This solution looks very complex, but it has many useless nodes. Table 3.9 simplifies the program by removing redundancies and simplifying anything constant expressions.

For the sake of argument, the GP system was run on a four-color problem. It seemed to learn that only two colors should be used, so the final fitnesses of the four-color tests were similar to those of the two-color test.

3.5 Considerations for FSP

Unfortunately, the results of MCP when generalizing it to multiple colors provide little insight into FSP. The size of the genes and the necessary diversity to produce solutions in these conditions are out of the scope of this work. Section 4.1.4 describes the representation

Part	Code
Program	(IF (NOR (GT (ADF0 (IF 1 -2 (ADD (MULT 0 arg[1]) 0)) (MOD (ADD -1 (IF (OR 0 (LT (SUB (DIV 1 0) (ADF0 -2 1)) 0)) -2 -1)) (IF (AND 0 arg[0]) (MULT (IF 1 -2 (ADD (MULT 0 arg[1]) 0)) -1) (MOD (ADF0 -1 (MULT (POW 0 (SUB -1 -2)) arg[1])) (ADF1 (DIV (MULT -1 (POW 1 0)) 1) 1 (ADF1 -1 arg[4] 0 -1) 0))))) (DIV (IF arg[0] (ADD -2 (SUB -2 arg[6])) arg[3]) -1)) 0) arg[0] arg[4])
ADF0	(MULT (SUB (DIV arg[6] 0) (MOD (DIV arg[6] 0) 0)) (MULT -1 -1))
ADF1	(POW (DIV -1 -1) (SUB -2 (POW (MULT (MOD 1 (DIV -2 (MULT 0 0))) 1) (ADF0 (DIV (DIV -2 arg[7]) arg[2]) (POW 1 arg[6]))))))

Table 3.8: This example GP solution will always return 1.

Part	Code
Program	(IF (NOR (GT 1 (DIV (IF arg[0] (ADD -2 (SUB -2 arg[6])) arg[3]) -1)) 0) arg[0] arg[4])
ADF0	1
ADF1	1

Table 3.9: A reduced version of the solution shown in Table 3.8.

of FSP genes and explains why the genes for that problem are still suited for evolutionary computing while multi-color MCP genes seem to be less useful without massive computing power.

The one piece of information that will be carried to FSP involves the GA crossovers. The hamming cross did not appear to work well on MCP where the maximum hamming distance was seven, allowing for a very large variation in how the gene was partitioned. In FSP, the maximum hamming distance will be three. This will provide even less variation and will probably function no better than the other two crosses. Therefore, the hamming cross will not be used in any FSP tests.

Chapter 4

Firing Squad Problem (FSP)

4.1 Representation

4.1.1 String Size

The bit-string representation for FSP is similar to MCP, but it has a few advantages. In MCP, any input that consists of valid states for the entire neighborhood is a valid input to the lookup table. FSP is allowed to set certain inputs to specific values and not deal with other inputs at all. For example, an input with a fire state should never occur in a FSP solution. Thus, this transition can be removed from the table.

FSP also works with a much smaller neighborhood. Instead of $r = 3$ like MCP, FSP uses $r = 1$. As a disadvantage, FSP is required to use many more colors than MCP. The current minimal time solution on the fewest states uses six states. If the evolutionary system starts from the same point, there are three inputs which can have six different values. This configuration produces a transition table with 216 different inputs. This is only a starting point. Now, the exceptions can be used to reduce the number of transitions and, consequently, the size of the search space. All of these changes are documented in Table 4.1. Similar logic can be seen in the description of Balzer's 8-state solution [2].

The biggest step in trimming the transition table is to remove any inputs that have a fire state in them. Since no member of the firing squad should fire early, the simulation ends as soon as one element fires and the transition will never get a fire state in its input. There can be one, two, or three fire states in any given input. There are three ways to have one fire state in the input. Each of those configurations have 5^2 configurations for the other inputs. Therefore, $3 \times 5^2 = 75$ inputs can be removed. For two fire states in the input, there are three configurations of the fire states and five configurations for the remaining input in each case. Thus, another 15 inputs can be removed from the table. Finally, there is only one way to have three fire states in the input, so that can be eliminated from the table as well.

At the edges of the FSP CA, there are cells that have a false state. This state allows for the table to handle edge detection gracefully. The false states can only exist in the first and

Symbolic	Numeric	Description
k^n	216	The initial number of possible inputs to the lookup table
$\binom{n}{1} \times (k-1)^{n-1}$	-75	The number of inputs that contain one "fire" state
$\binom{n}{2} \times (k-1)^{n-2}$	-15	The number of inputs that contain two "fire" states
$\binom{n}{3} \times (k-1)^{n-3}$	-1	The number of inputs that contain three "fire" states
$2 \times \sum_{i=1}^{r} (k-1)^{n-r}$	+50	The number of inputs that contain the false state used for edge detection
$2 \times r$	-2	The number of inputs with false states and *latent* states only
	-1	The number of inputs that contain all *latent* state
	+1	The input to determine the first state
	173	*Total*

Table 4.1: The number of transitions needed in the FSP transition table. n is the size of the neighborhood, k is the number of colors, r is the radius of the neighborhood. The FSP restrictions allow us to remove certain transitions from the transition table and add others for edge detection. The symbolic column shows generalized formulas for calculating the number of transitions. The numeric column shows the actual numbers for the six-color case.

last positions in the input, but not both[1]. In both cases, there are two other inputs that can take on five values[2]. Thus, $2 \times 5^2 = 50$ new inputs must be added to the transition table to handle the edges of the CA.

Next, the latent rule must be handled. Any set of inputs that do not contain any non-latent values should return the latent state. Thus, the table itself does not need to handle this and three more inputs can be removed.

Finally, the GA should be able to choose what state the general enters at the very beginning. One of the states removed because it is latent is put back into the table. The danger of using this input is that it could be used more than once. If it is, the solution is not a valid FSP solution.

4.1.2 Input Mapping

Unlike MCP, the decoding of an input into an index in the gene is slightly more complicated. If all of the legal inputs are decoded to indices like MCP and are sorted, the index of the sorted list corresponds to the index in the gene that should be returned. Thus, the input `001` maps to index 0 since `000` is not an input in the transition table. As a result of the more complex decoding, it is a bit slower than the simple decoding used in MCP.

4.1.3 Gene Values

The existing solutions to FSP all have very few transitions that output the fire state. In order to compensate for this, the gene is not composed of values pertaining to states. Instead, the range of values can be much greater. For example, the range of values can be from 0 to 50, inclusive, and the system is attempting to create a solution with 6 states. In this case, only the value 50 translates to a fire state. Values 0 through 49 translate to states 0 through 4 evenly. Thus, non-fire state is ten times more likely to appear than the fire state. In order to demonstrate the value of this strategy, it will be considered a variable in the tests.

4.1.4 Representation Benefits

They key to solving FSP using GA lies in the representation. Since the neighborhood is small, the gene for FSP is much smaller than for MCP on multiple colors. In addition, partial credit is much more difficult to achieve with FSP than with MCP. Once a certain approach has been started in the EC system, significant changes tend to hurt the fitness of the individual. For the same reason, if the EC attempts the wrong approach to solving FSP, it is very difficult for it to correct itself.

[1]The solution will not work on a CA of size one.

[2]It is important that these values cannot be fire states since those have already been removed.

4.2 Fitness

Since FSP is so much more complex than MCP, the fitness function is more complex as well. There are three main contributing (or detracting) factors:

1. the number of elements in the fire state at the end of the simulation
2. the number of elements that changed value during the course of the simulation
3. the length of the simulation, especially if the length is too short

The number of elements in the fire state is always used. It can be calculated in two ways. First, it can be taken as simply the number of elements that fired. The second method squares the size of any group of adjacent firing elements. For example, the first method might award a fitness of three while the second method would award the individual a fitness of five if two of the firing states are adjacent. By encouraging adjacency, it is hoped that the EC will evolve firing transitions that can coexist in close proximity.

As will be seen in the tests, it is important to encourage the solutions to change all of the elements in the CA. Without this encouragement, the solutions will usually simulate a few steps and then fire before many of the elements ever get a chance to change value. To avoid this, a bonus is given for every element that changes. However, this bonus is only given when the simulation lasts at least $2n - 1$ steps.

Since $2n - 1$ is the minimum time a solution must use to solve an instance of FSP[5], a penalty is detracted from an individual's fitness for every time step early that the simulation ends. By forcing the solutions to run for the minimum amount of time, solutions that simply fire after a set number of steps each time are less likely to succeed and propogate.

Once the three pieces of the fitness function have been calculated, they are weighted based on the parameters described in Section 4.3. The weighting allows for the variation for the effect of a piece as well as the ability to turn it completely off.

During the simulation of a solution, the halting rules are the same as for MCP with one addition. Now, the simulation will halt at the end of the time step if any of the elements fired. By doing this, transitions whose input contains the fire state are not necessary and can be ignored, as stated in Section 4.1.1.

Since perfect solutions to instances of FSP will be much less likely than for MCP, the FSP solutions will be compared by what percentage of elements fired in the last time step of their simulations. This number will be referred to as the raw fitness of the individual.

4.3 Parameters

For FSP, there are fewer parameters than for MCP, and they are much simpler. Table 4.2 shows the paramters with a short description and their values.

The first parameter provides the lengths of CA on which the FSP solution will be tested. The boundaries were chosen for two reasons. First, they are small. Second, the specified range contains three prime numbers and five relatively prime numbers. It has been seen

Parameter	Description	Value
Mininumum and Maximum Length	The range of CA lengths on which to test the individuals	13 and 19
Time Limit	A multiplier on the length of the CA to determine the number of steps in the simulation	4
Values	The number of possible values each index in the gene can have	51 and 6
Colors	The number of colors that can actually be used in the FSP simulation	6
Changed Weight	The fitness contribution of a changed element	*variable*
Early Firing Weight	The fitness detraction of the simulation firing a single step earlier than $2n-1$ steps	*variable*
Firing Weight	The fitness contribution of a single element firing	*variable*
Crossover Type	The crossover method to use	One-point and Radix
Square Firing Fitness	Square adjacent firing elements in the CA	*variable*

Table 4.2: Parameters for FSP simulation

Parameter	Description	Value
Population Size	The number of individuals that exist in a single generation	100
Elites	The percentage of individuals selected as the elites from which to create the next generation	20%
Selection Method	The type of selection used to choose the elites from the population	elitism, tournament, and proportional
Preserve Best	Whether or not to preserve the best member of the population for non-elitism selection methods	`true`
Time Limit	The number of generations to run before stopping	1000
Random Seed	The seed for the random number generator	randomly typed values

Table 4.3: The parameters for the generational solver running FSP

before that sequences of states may be used as signals, allowing for solutions that only work on multiples of a certain number. Thus, the large number of relative primes encourages more robust solutions.

The values parameter works as described in Section 4.1.3 to discourage a large number of transitions to the fire state.

All of the weights are multiplied by their respective pieces of the fitness function in order to determine the final fitness value for an individual. The square firing fitness parameter specifies whether the squaring bonus of adjacent firing cells is turned on or off.

The generational solver also has parameters. For the most part, they are the same as those used for MCP With a few changes, all of the solver parameters are shown in Table 4.3.

4.4 Results

Since there are more variables involved in FSP than in MCP, the first step is to remove the binary variable. Using five test runs of each setting with all other parameters held constant, Table 4.4 shows the average raw fitness of the heroes produced by the runs. The numbers show almost no difference between the two tactics. Since the best result came from not using the squaring strategy, that parameter will be set to false for the rest of the tests.

Using the same type of test, the crossover method can also be eliminated from the variables. Using the same static variables as before with square being false, Table 4.5 shows the results of one-point and radix crossover techniques. Luckily, the data from the squaring

Values	Changed	Early	Fire	Square	Crossover	Selection	Raw Fitness	σ
51	1	1	1	`true`	one-point	tournament	42.9%	22.50%
51	1	1	1	`false`	one-point	tournament	43.8%	19.99%

Table 4.4: Results of running five tests with and without squaring of adjacent firing elements turned on

Values	Changed	Early	Fire	Square	Crossover	Selection	Raw Fitness	σ
51	1	1	1	`false`	radix	tournament	25.5%	14.09%
51	1	1	1	`false`	one-point	tournament	43.8%	19.99%

Table 4.5: Results of running five tests using one-point crossover and radix crossover

test can be used for the one-point crossover here. There is a large disparity in the raw fitnesses between the two crosses. Since the one-point crossover appears to work better, it can be set as a constant despite the large overlap when the standard deviation is considered.

The last parameter to eliminate before adjusting the fitness weighting is the selection method. All three methods can be attempted with five runs each. Again, the data from the previous test can be used to fill in one piece of this test. The data for this test is shown in Table 4.6. Unlike MCP where tournament selection did not seem to perform as well, proportional selection has the worst success rate of the three. Tournament selection and elitism have very similar success rates. Since tournament selection has the slightly better average and since it has produced the best individual thus far, it will be used as the default selection method for the remainder of the tests.

Now, it is time to adjust the weighting of the different components of fitness function. For now, the *changed* and *early* weights can take on the values 0, 1, and 2. The firing weight must not be turned off, so it will only have the values 1 or 2. Several other combinations of weights have been removed due to redundancy. Table 4.7 shows the results of the weight tests, running three each times.

The results show some combinations of weights to be detrimental to the task. No single combination seems to significantly out-perform the others. However, there is an interesting pattern. There are four configurations where the early firing weight is zero. All of these configurations rank in the top six. In three of those cases, the changed weight is not zero.

Values	Changed	Early	Fire	Square	Crossover	Selection	Raw Fitness	σ
51	1	1	1	`false`	one-point	proportional	22.3%	13.07%
51	1	1	1	`false`	one-point	tournament	43.8%	19.99%
51	1	1	1	`false`	one-point	elitism	42.0%	15.17%

Table 4.6: Results of running five tests using the different seleciton methods

Values	Changed	Early	Fire	Square	Crossover	Selection	Raw Fitness	σ
51	0	0	1	false	one-point	tournament	44.6%	1.55%
51	0	1	1	false	one-point	tournament	36.9%	26.73%
51	0	1	2	false	one-point	tournament	24.1%	12.37%
51	0	2	1	false	one-point	tournament	30.4%	7.14%
51	1	0	1	false	one-point	tournament	41.1%	21.34%
51	1	0	2	false	one-point	tournament	47.3%	3.57%
51	1	1	1	false	one-point	tournament	43.8%	19.99%
51	1	1	2	false	one-point	tournament	25.3%	11.09%
51	1	2	1	false	one-point	tournament	22.3%	9.45%
51	1	2	2	false	one-point	tournament	29.2%	20.85%
51	2	0	1	false	one-point	tournament	39.6%	27.98%
51	2	1	1	false	one-point	tournament	34.2%	27.87%
51	2	1	2	false	one-point	tournament	37.5%	12.08%
51	2	2	1	false	one-point	tournament	40.2%	22.43%

Table 4.7: Results of running three tests using different weights for the pieces of the fitness function. The raw fitness where all weights are 1 is the same as was used in Tables 4.4, 4.5, and 4.6.

Values	Changed	Early	Fire	Square	Crossover	Selection	Raw Fitness	σ
6	1	0	2	false	one-point	tournament	32.9%	13.97%
51	1	0	2	false	one-point	tournament	51.8%	12.88%

Table 4.8: Results of running five tests using direct and ranged mapping from values to colors

Since the changed component of the fitness is dependent on the simulation using enough steps, the apparent ineffectiveness of the early firing component is expected.

Now, the best result from Table 4.7 can be used to test whether or not using 51 values is better than using 6. Table 4.8 shows that the 51-value configuration has a much better raw fitness. It is also important to note that none of the 6-value tests produced a hero that ran for more steps than the length of the CA. The hero that ran for the most steps is shown in Figure 4.1. Therefore, none of them received fitness points for having changed the values of the cells.

4.5 Result Comparison

Figure 4.2 shows the current 6-state solution. It demonstrates a very human organization and attention to signal speed and type. In effect, Mazoyer's solution exhibits a divide and conquer strategy to solving FSP. Up to this point, the FSP heroes produced by the EC

Figure 4.1: This solution did not run long enough to receive any length or element change bonuses. It is the 6-value solution that ran for the most time steps.

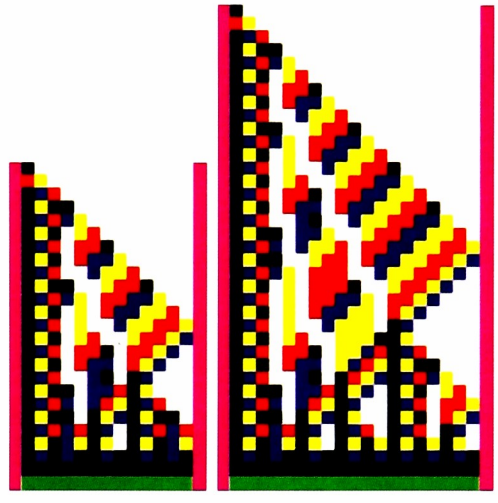
system have been judged on percentage of cells that fire in the last step of the simulations. However, this information does not guarantee that the solution is viable in any other way.

The length of the CA and, subsequently, the minimal time for a real solution are usually long enough to avoid having a solution that always fires after a certain number of steps. It is difficult to create a timing sequence 37 steps long with only 6 states. That does not keep the GA from providing some more interesting variations on that theme. Figures 4.3 and 4.4 show that the GA learned to start the timing sequence from the right side of the CA. Still, though, the sequence is not long enough to allow all of the elements in the larger CA to fire.

Perhaps even more creative, the solution shown in Figure 4.5 combines two approaches to cheating the rules of the problem. It uses a much slower signal to notify the right soldier. Once the right soldier is non-latent, a very short timing sequence begins and the CA fires 11 time steps later. This problem with timed firing can be solved in one of two ways. First, the CA lengths could be longer. This would make timing even more difficult. Such a change would also greatly increase the amount of time it takes to run the GA. The other option is to adjust the fitness function to be more strict in some way. This could be in the form of ensuring that all elements change in a certain amount of time, rather than see if they ever change.

Since developing a better fitness function without being too strict is difficult, lengthening the CA may be the answer. The only issue with lengthening the CA, other than processing time, is how long to make it. Figure 4.6 shows a solution that has a 44 step timing squence. At the end, there seems to be a fast signal sent out. If that signal reaches the right soldier instead of firing and starts a similarly long timing sequence, the same could occur in a CA of length 50 or 60, and the solution would still be invalid.

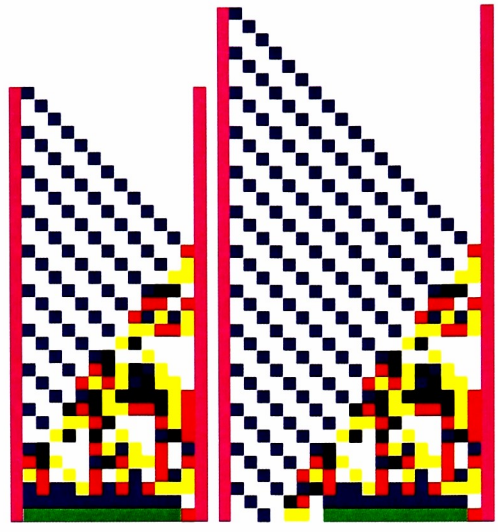
It was mentioned earlier that the large number of relative primes in the CA lengths allowed the system to avoid patterned signals. Such signals can be seen in Figure 4.7. The solution presented was trained on a single CA length of 12. As a result, it evolved a signal that, when attempted on a CA of length 13, breaks down. Though it looks shorter, the signal is actually the length of the CA. The solution breaks down by only firing on the right or left. Thus, it is not a timing sequence like we have seen previously. It is a very specifically evolved signal that only works on the training CA. This result is probably the most complex set of interactions presented here.



(a) length of 13

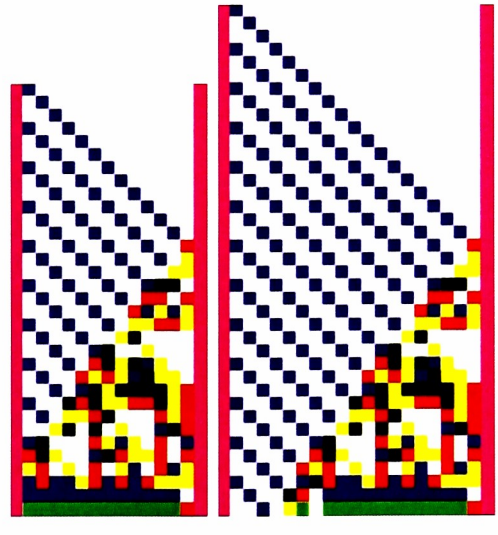
(b) length of 19

Figure 4.2: The current 6-state minimal-time solution to the firing squad problem. Each column is a *soldier* in the firing squad. The different colors represent the different states. White indicates the latent state, and green represents the fire state. In all, there are six states used by the soldiers (including the latent and fire states) plus one state used to show the boundary of the CA, shown in purple. The four unnamed states have no specific significance other than being intermediate steps towards the firing of the CA.



(a) CA length of 13

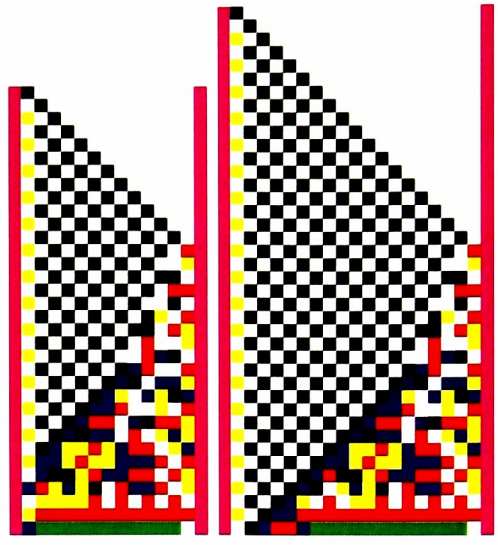
(b) CA length of 19



(c) CA length of 13 after local search

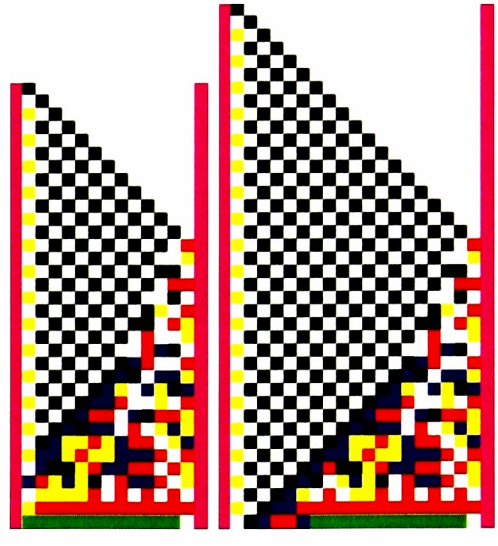
(d) CA length of 19 after local search

Figure 4.3: This FSP solution fired on 83 of the 112 elements. This was generated using a changed weight of 1, an early firing weight of 0, and a fire weight of 2. Once the initial signal reaches the right edge of the CA, a timing sequence is begun and the CA fires after a set number of steps.



(a) CA length of 13

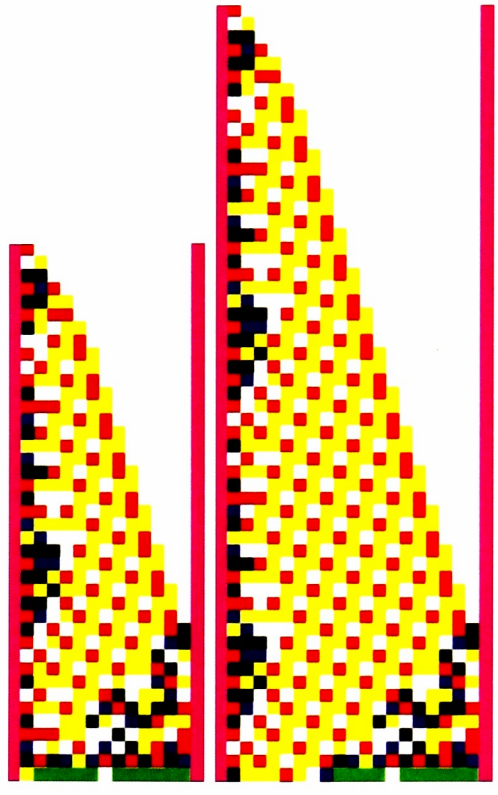
(b) CA length of 19



(c) CA length of 13 after local search

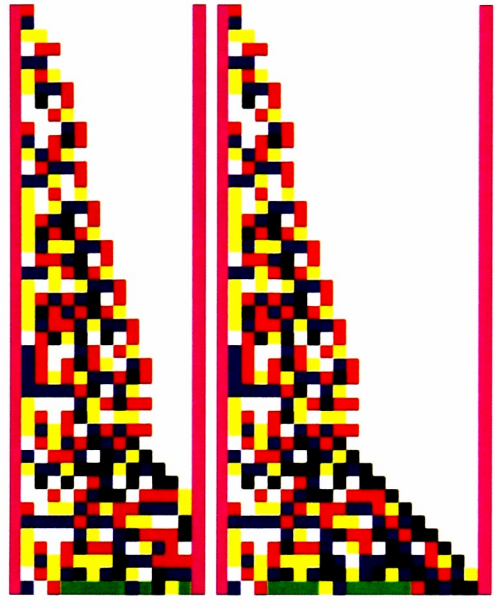
(d) CA length of 19 after local search

Figure 4.4: This GA-generated FSP solution fired on 85 of the 112 elements in all of the CAs (13 through 19). This was generated using a changed weight of 1, an early firing weight of 1, and a fire weight of 1. The black signal reaches the right edge and starts a timing sequence that fires always after the same number of time steps.



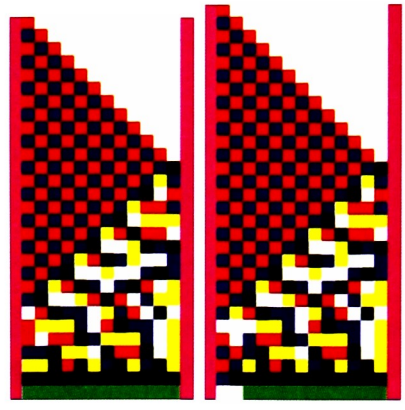
(a) CA length of 13 (b) CA length of 19

Figure 4.5: This GA-generated FSP solution fired on 74 of the 112 elements in all seven CAs (13 through 19). This was generated using a changed weight of 2, an early firing weight of 1, and a fire weight of 1. A very slow yellow and red signal eventually starts a very short timing sequence once it reaches the right edge. The shorter CA gets an extra element to fire due to its proximity to the noise on the left side of the CA.



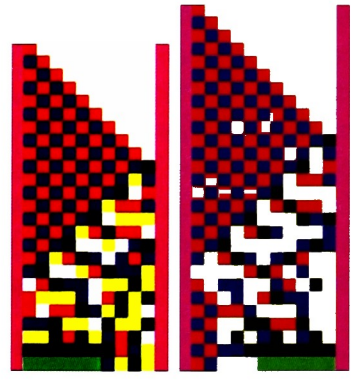
(a) CA length of 13 (b) CA length of 19

Figure 4.6: This GA-generated FSP solution fired on 60 of the 112 elements. This was generated using a changed weight of 2. an early firing weight of 2, and a fire weight of 1. This result shows a very slow timing sequence that fires in the same number of steps regardless of the size of the CA. Unlike the previous timing sequences, this one starts on the left side in the very first state and does not interact with the right edge in any meaningful way during the simulations.



(a) CA length of 12

(b) CA length of 13



(c) CA length of 10

(d) CA length of 11

Figure 4.7: This FSP solution was trained on a CA of length 12 only. This was generated using a changed weight of 1, an early firing weight of 0, and a fire weight of 2. A pattern can be seen moving from the right soldier to the left. This pattern is specifically tuned to the length of the CA. Different lengths produce different behaviors that are not consistent with simple timing sequences.

Chapter 5

Conclusions

Through the course of this work, certain strategies have been tested against FSP. Certain combinations of settings have proven to work better than others. While the EC may not be able to solve this problem consistently, it does make progress toward a solution. The difficulty lies in the constraints placed on the system. If the signals that exist in Mazoyer's solution could be parameterized and added as a sort of fitness for the EC system, a solution may present itself more readily. However, such restraint of the evolutionary system would remove the possibility for a different approach to the problem. The constraints used in the fitness function were designed to encourage the solution to have certain necessary characteristics while leaving the implementation of the solution completely in the hands of evolution.

It has also become apparent that MCP and FSP are not as similar as hoped. Though they are both very strict CA problems, MCP is more difficult to train, since t has a large number of initial configurations, has a much larger neighborhood, and does not have an existing solution. For MCP, a useful solution is still only partial. With that fact, it is possible that the fitness function used in previous work is better since it assigns less partial credit. Since FSP is completely reliant on partial credit, a similar strategy was attempted for MCP and yielded poor results.

As mentioned before, longer training CA could be more effective but will take much longer to run. In the worst case, a neighborhood of three with six states could create a timing sequence 216 steps long. In order to ensure that timing sequences do not occur in GA-accepted solutions, a CA of length 217 or higher would be required in the training set. However, even a long CA may not be enough. We could conceive of a CA that sends different signals back and forth across the CA several times and eventually ends in a timing sequence.

Since FSP is such a complex problem, there may also be room for alternative fitness evaluations. Since humans can detect patterns more easily, human input in a tournament selection scheme may be the best option for problems such as FSP. Again, that constrains solutions to those that a human can see and understand.

In the end, no solution to FSP was found, but a start has been made on approaching FSP with evolutionary computing. Further experimentation may yield substantial results. Because EC systems are Markov chains [7], there is always a probability of creating a population with a solution. The key is to increase that probability to a level where it becomes

common.

The easiest method of adjusting the probability is through forcing more transitions than those specified by FSP. For example, Mazoyer defines a class of transitions where the left input is non-latent and the other two inputs are latent. In this case, the transition table would return the non-latent state [5]. With just this transition, the need for the *changed* portion of the fitness function becomes unnecessary.

Another way of biasing the EC is through a more complex fitness function. If the solution should have Mazoyer-like signals, the fitness function could try to quantify the existence of those signals. This sort of addition could prove very computation intensive and might be unable to properly identify such signals in the general case. Similarly, a measurement of the divide and conquer methodoloy could be added.

From the biases and allowances used in this work, we can see that EC has a very powerful ability to meet the rules allowed for in the fitness function without going too much further. In order to properly evolve a general solution to FSP, we must define some features of the solution beyond what the problem specifies. The difficulty is determining how to quantify these qualities in the fitness function or force them through the transitions that are allowed to appear in the table.

Bibliography

[1] David Andre, III Forrest H. Bennett, and John R. Koza. Evolution of intricate long-distance communication signals in cellular automata using genetic programming. In *Artificial Life V: Proceedings of the Fifth International Workshop on the Synthesis and Simulation of Living Systems.* The MIT Press, 1996.

[2] Robert Balzer. An 8-state minimal time solution to the firing squad synchronization problem. *Information and Control*, 10(1):22–42, 1967.

[3] R. Breukelaar and Th. Bäck. Using a genetic algorithm to evolve behavior in multi dimensional cellular automata: emergence of behavior. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 107–114, New York, NY, USA, 2005. ACM Press.

[4] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* The MIT Press, 1992.

[5] Jacques Mazoyer. A six-state minimal time solution to the firing squad synchronization problem. *Theor. Comput. Sci.*, 50(2):183–238, 1987.

[6] Jacques Mazoyer. On optimal solutions to the firing squad synchronization problem. *Theor. Comput. Sci.*, 168(2):367–404, 1996.

[7] Melanie Mitchell. *An Introduction to Genetic Algorithms.* The MIT Press, Cambridge, Massachusetts, USA, 1998.

[8] Melanie Mitchell, James P. Crutchfield, and Rajarshi Das. Evolving cellular automata with genetic algorithms: A review of recent work. In *Proceedings of the First International Conference on Evoluationary Computation and Its Applications (EvCA'96).* Russian Academy of Sciences, 1996.

[9] Jonathan Rowe. Genetic algorithm theory. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, New York, NY, USA, 2005. ACM Press.

[10] Lee Spector, Jon Klein, and Maarten Keijzer. The push3 execution stack and the evolution of control. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1689–1696, New York, NY, USA, 2005. ACM Press.

[11] J. B. Yunès. Seven-state solutions to the firing squad synchronization problem. *Theor. Comput. Sci.*, 127(2):313–332, 1994.