SUDOKUBE – USING GENETIC ALGORITHMS TO SIMULTANEOUSLY SOLVE MULTIPLE COMBINATORIAL PROBLEMS

By

DAVID ISAAC WATERS

B.S. Electrical Engineering

Oklahoma State University

Stillwater, OK

2005

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 2008

SUDOKUBE – USING GENETIC ALGORITHMS

TO SIMULTANEOUSLY SOLVE MULTIPLE

COMBINATORIAL PROBLEMS

Thesis Approved:

Dr. Gary Yen

Thesis Adviser

Dr. Keith Teague

Dr. George Scheets

Dr. A. Gordon Emslie

Dean of the Graduate College

TABLE OF CONTENTS

Chapter Page

I. INTRODUCTION 1

II. LITERATURE REVIEW

II.A Optimization Algorithms..4
II.B Genetic Algorithms/Evolutionary Algorithms 5
II.C Genetic Algorithms..5
II.C.1 Genetic Operators 9
II.C.1.a Parent Selection 9
II.C.1.b Crossover 11
II.C.1.c Mutation...14
II.D Simulated Annealing 16
II.E NP Complete...17
II.F Combinatorial vs. Continuous Optimization 18
II.G Sudoku..24
II.H 3D Sudoku..28

III. DESIGN METHODOLOGY

III.A Design Introduction...31
III.B 2D Solver..31
III.B.1 Genetic Operators..33
III.B.1.a Parent Selection..33
III.B.1.b Aging 34
III.B.1.c Uniform Crossover 36
III.B.1.d Mutation/Natural Growth..38
III.C Puzzle Generation 42
III.D Puzzle Combination 45
III.E 3D Representation -- SudoKube..48
III.E.1 SudoKube Introduction 48
III.E.2 Genotype...48
III.E.3 Genetic Operators...49
III.E.3.a Parent Selection 49
III.E.3.b Crossover...49
III.E.3.c Mutation...52
III.E.3.d Fitness Calculation 52

Chapter Page

III.E.4 Settings 53
III.E.4.a Mutation Ceiling 53
III.E.4.b Reset Count 54
III.E.4.c Mutation Rate Multiplier 57
III.E.4.d Maximum Iterations 58
III.E.4.e Mutation Iterations 59
III.E.4.f Population Size 60
III.E.4.g Difference Degree 61

IV. RESEARCH FINDINGS 64

V. CONCLUSIONS / FUTURE RESEARCH 76

V.A 2D Representation – Success 76
V.B 3D Representation – Success 76
V.C Future Work 77

REFERENCES 79

LIST OF TABLES

Table Page

1. Comparison of continuous vs. combinatorial problems 23
2. Number of solutions per side 30
3. Example of fitness-dependent crossover 52
4. Solutions found out of 30 attempts 65
5. Solutions found out of 100 attempts 66
6. Overall solver results – Helsingin Sanomat..67
7. Overall solver results – Aamulehti 68
8. Comparison of sequential solver vs. 3-D solver 73

LIST OF FIGURES

Figure Page

1. Sample unsolved 'easy' puzzle with symmetric blanks 2

2. Pseudo code outlining a basic genetic algorithm..7

3. Uniform crossover example...12

4. One point crossover example...12

5. Two point crossover example 13

6. Mutation example 15

7. Solution of an N-Queen problem...18

8. Rosenbrock's Valley – DeJong's 2nd function...19

9. Gradient example 21

10. Sample solved puzzle...26

11. Example SudoKube with three solved edges..29

12. Basic genetic algorithm flow chart 33

13. Example of the aging operator performing an exchange 35

14. Uniform crossover flow chart 37

15. Row crossover example 38

16. Unsolved Sudoku grid with duplicates 41

17. Mutation operator flow chart 42

18. Randomly generated mini-grids..44

19. Overview of five linked puzzles 46

Figure Page

20. Initialization flow chart 47

21. Fitness dependent with difference degree crossover 51

22. Solve time vs. mutation ceiling 54

23. Solve time vs. reset count (A) 55

24. Solve time vs. reset count (B) 56

25. Fitness vs. mutation rate multiplier (0.90) 57

26. Fitness vs. mutation rate multiplier (0.991) 58

27. Time vs. mutation iterations 60

28. Time vs. population size 61

29. Time vs. difference degree 62

30. Fitness vs. iterations (Easy) 70

31. Fitness vs. iterations (Medium) 70

32. Fitness vs. iterations (Hard) 71

33. SudoKube screenshot 72

34. Solve time vs. puzzle type & difficulty 74

35. Solved SudoKube 78

CHAPTER I

INTRODUCTION

Sudoku puzzles provide a logical challenge for people of all ages. These puzzles can be found in newspapers, magazines, puzzle books, and even in cell phones. The format of a Sudoku puzzle is of an $n^2 \times n^2$ grid divided into nine smaller mini-grids. Each mini-grid is $n \times n$. Bold lines generally separate the mini-grids. Although one can create similar puzzles to Sudoku with many different n values, n is normally equal to three, which makes the entire grid 9×9. The objective of the game is to fill each row, column, and mini-grid with the numbers 1 through 9 without duplicate numbers in any row, column or mini-grid. Puzzle difficulty is partially related to how many cells are filled at the beginning, although the placement and quality of the starting hints play a much larger role in determining whether a puzzle is 'Easy' or 'Fiendish.' The empty cells are traditionally symmetric with respect to a 180 degree turn around the center cell as shown in Figure 1. The filled cells are considered static for each particular problem **[1]** However, there are some publications that have asymmetric Sudoku grids **[2]**.

9	7	1			6			2
3			7					
5			3				7	
	3	9		7				
			1	6	9			
				3		9	2	
	5				7			9
					3			4
4			9			8	1	7

Figure 1: Sample unsolved 'easy' puzzle with symmetric blanks. Blanks are symmetric about the center cell.

Sudoku puzzles are also an interesting combinatorial problem, and although not *extensively* researched, several papers have been published on the use of searching algorithms with respect to solving Sudoku puzzles **[3][4][5]**. Sudoku is often viewed as an excellent testing application for combinatorial solvers.

Genetic algorithms are powerful tools used in solving optimization problems. They are most effective when applied to problems with large, variable search spaces with unknown patterns. They use the 'survival of the fittest' concept to maintain a population of good quality solutions while working towards an optimum. Genetic algorithms have been applied successfully to the traveling salesman problem **[6][7][8]**, the N-Queens problem **[9]**, flowshop sequencing **[10][11]**, and also to solving Sudoku puzzles **[5][12][4]**. All of these problems are of the type NP-Complete **[13]**.

This thesis will begin by describing optimization problems, genetic algorithms, and the combinatorial and NP-Complete problem types. Chapter II will discuss different approaches to creating a genetic algorithm, and it will explain why the use of genetic algorithms is an effective method for solving optimization problems.

In Chapter III, this thesis will first outline an effective puzzle generation method for Sudoku, followed by a solver that surpasses its genetic algorithm predecessors in terms of effectiveness. Following that, it will describe the creation of a *new* type of Sudoku game – six Sudoku puzzles in the form of a box, called a SudoKube. Most significantly, this section will describe the modified operators used in the algorithm that will effectively and consistently solve a SudoKube, which is a combinatorial optimization problem with varying constraints.

The results will be outlined in Chapter IV. Results from the proposed solver and similar existing algorithms will be compared. Results comparing the 2D and 3D solvers' effectiveness when applied to the SudoKube are also included in Chapter IV. The relevant conclusions will be drawn in Chapter V.

CHAPTER II

REVIEW OF LITERATURE

II.A Optimization Algorithms

Optimization algorithms are designed to find the best solution for a given problem and set of constraints. There are several challenges that face any optimization algorithm. A large search space often makes finding a global optimum difficult. The larger the search space, the more challenging it is to find and verify an optimal solution. Also, any case that has only one global optimum within a plethora of sub-optimal solutions results in the proverbial 'needle in a haystack' scenario. Finally, finding *the* optimal solution may require the algorithm to perform an extensive search any time it reaches a local optimum, which is generally impractical in terms of the time it would take to find the optimal solution. For problems such as the travelling salesman problem, the only way to verify that a solution is the global optimum is an exhaustive search of every possible solution.

Many problems contain local optimums that can 'trap' many searching algorithms (these local optimums are often called 'basins'). A searching algorithm can find a local optimum within a basin and, unless the algorithm is designed to be able to escape said basin, it will likely never find a better solution than the one in its current location. Without the capacity to escape, the algorithm could easily (and incorrectly) determine that it has found the *global* optimum. Local optimums are much easier to find compared to global optimums in the majority of cases **[14]**.

II.B Evolutionary Algorithms/Genetic Algorithms

Evolutionary algorithms are a grouping of heuristic solving techniques that include processes such as genetic algorithms, particle swarm optimization, and evolutionary programming. This thesis will focus primarily on genetic algorithms and their characteristics. Another type of optimization algorithm is simulated annealing. Genetic algorithms can be augmented by methods found in simulated annealing. This section will discuss some concepts behind simulated annealing and will cover some of its unique characteristics **[15]**.

II.C Genetic Algorithms

A genetic algorithm (GA) is a heuristic optimization method originally proposed by Holland **[16]** that is based on some of the most basic biological concepts: survival of the fittest, natural selection, and the transfer of biological characteristics (genetic material) from parents to children.

A potential solution in a genetic algorithm fits a genotype that is defined at the outset of the program. For example, a genotype for a traveling salesman problem may be a one dimensional array containing a permutation of the cities the salesman must visit. The following array could be a population member for a specific traveling salesman problem.

$$< St.Louis, Tulsa, Oklahoma\ City, New\ York, Dallas, Scranton, Springfield >$$

However, in the interest of programming simplicity, each of these cities may be assigned a number according to their position, and the resulting permutation may look like this:

$$< \ 1, 2, 3, 4, 5, 6, 7 \ >$$

Genetic algorithms initially used binary representation, with 1's and 0's defining a parent's genetic material **[17]**. It is more common now to see real valued representation.

A genetic algorithm works by first initializing and then maintaining a population of potential solutions and evolving them over the course of many generations through the use of different types of functions, called operators. The quality of a solution is described by its 'fitness,' which is a problem dependent objective. The better a solution's fitness, the more likely it is that it will be selected for reproduction. This is where the survival of the fittest concept comes into play for a genetic algorithm. For a distance-minimizing problem (say, a sales route for a traveling salesman), an individual population member would be a specific sales route, and the fitness would be the total distance traveled. A good sales route would have a low fitness value, which would be a direct result of short travel distance **[6].**

After determining each existing solution's fitness, a genetic algorithm will apply crossover and mutation operators in order to generate new population members. The crossover operator simulates a mating process, and the mutation operator simulates the unlikely event of a gene being mutated within a population member. Depending on the genetic algorithm, crossover will either generate a new generation equal to the size of the

original generation before re-integration or it will generate one child and integrate it back into the population immediately. Generating a number of children equal to the population size occurs in a 'generational' algorithm, and generating one child at a time occurs in a 'steady state' algorithm **[14]**. In a steady state algorithm, a child could be selected as a parent immediately after its insertion into the general population. In a generational algorithm, the children are not available in the parent selection process until their entire generation has been created and inserted back into the general population. The creator of genetic algorithms, Holland, described the generational algorithm as having 'intrinsic parallelism.' When dealing with large search spaces, this can be very beneficial **[18]**. The process is outlined below in Figure 2.

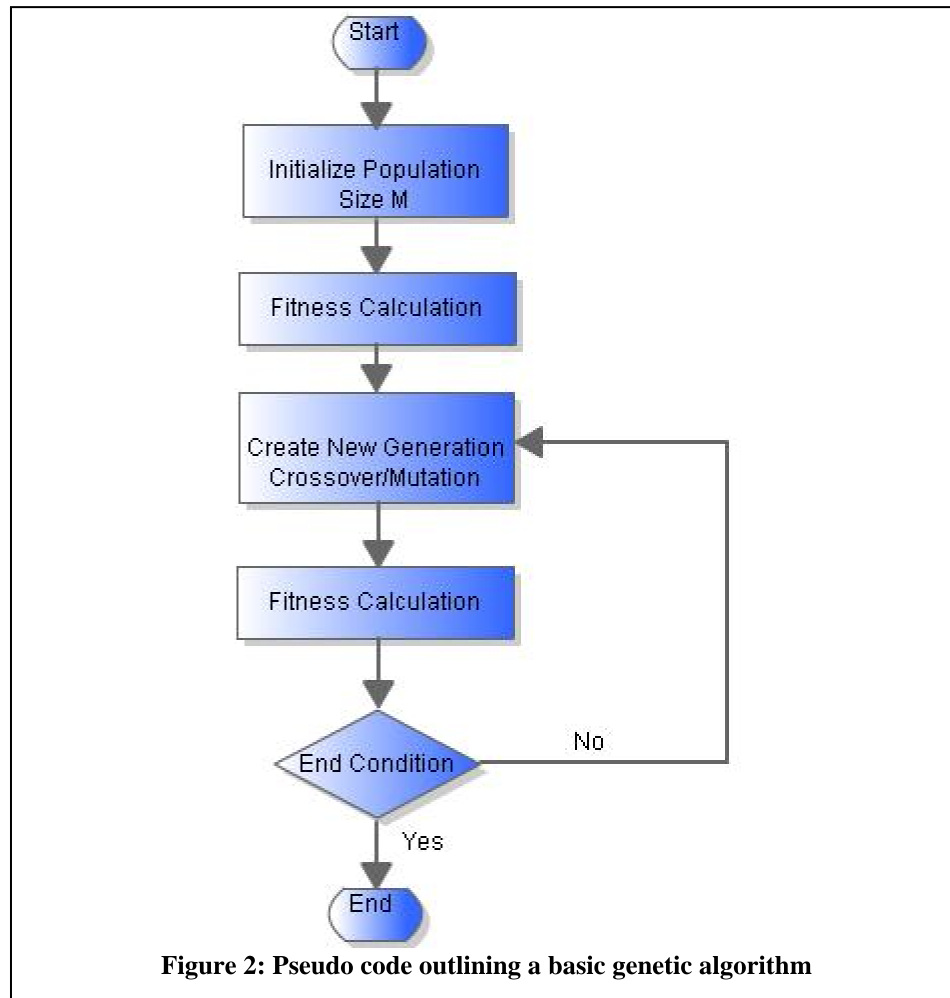


Figure 2: Pseudo code outlining a basic genetic algorithm

Genetic algorithms can be very powerful tools to solve problems that would be too complex or cumbersome to attack with brute force or with an algorithm that relied strictly on problem logic. This includes problems with large search spaces or problems for which the pattern of solutions is not known or not easily found. Sudoku provides an example of a very large, variable search space that depends solely on the initial hint distribution **[19]**. The size of the search space for a specific Sudoku puzzle grows exponentially with each starting blank. A genetic algorithm has the ability to find *optimal* solutions within a large search space without requiring much user supplied information.

The successful use of genetic algorithms in optimization problems of both the continuous and the combinatorial variety indicates a good deal of flexibility within the genetic algorithm concept itself **[20][21]**.

Through the use of standard operators and the situational application of modified operators, a genetic algorithm can be used to solve both individual and *linked* (3D) Sudoku puzzles correctly and efficiently. Genetic algorithms are a method of *searching* for an optimal solution rather than a method of *solving* a Sudoku puzzle to obtain the answer. There are many Sudoku solvers that can use human puzzle-solving techniques to arrive at a valid answer for a given Sudoku puzzle **[1][22]**. This thesis will not compare the results of this genetic algorithm with those solvers, but it is worth noting that there are a few puzzles that are virtually impossible to solve using human logic techniques, such as

the ‘Escargot’ puzzle **[22]**, which requires a player to consider eight cells simultaneously in order to begin the puzzle.

There are a few drawbacks that come with the use of genetic algorithms. Qualitative analysis of these algorithms is quite lacking in the literature. Results are not repeatable due to the stochastic nature of the algorithm itself. While the genetic algorithm may arrive at the same answer on virtually every run, the method by which it reaches the final result will be different each time. These qualities can make genetic algorithms somewhat difficult to troubleshoot and analyze **[16]**.

Parent selection is a staple within any genetic algorithm. As stated previously, crossover is the manifestation of the ‘survival of the fittest’ concept within a solving algorithm, and crossover essentially begins with parent selection. Another operator within a standard genetic algorithm is mutation. There are many different methods for applying crossover and mutation, and there are even more methods that involve slight modifications to those operators. Designing an appropriate crossover and mutation combination to fit a specific problem is both a science and an art **[21]**. There are many different settings to choose and decisions to make before finalizing this operator pair.

II.C.1 Operators

II.C.1.a Parent Selection

Genetic algorithms weed out solutions with bad fitness values partially through the use of parent selection methods. The goal of these methods is to allow the best

solutions to mate more often than the poor solutions so that the next generation will receive the best genetic information from the parents' generation. Simply put, a solution with good fitness will pass on much more genetic material than a solution with bad fitness.

The parent selection operator is exactly what its name implies. It is a method of selecting parents to mate and create a child or a number of children. Traditionally, the number of parents allowed for any individual child is two **[20]**.

Depending on the type of genetic algorithm, parents may or may not pass on 'learned' information to their children. If the algorithm allows for the transfer of learned information from one generation to the next, it is called Lamarckian evolution **[14]**. However, a 'purer' form of genetic algorithm would not pass on learned information via genetics, based on the Darwinian model of evolution. Rather than keeping and using the learned traits, the Darwinian model directs the search toward the areas with those traits. A large portion of genetic algorithms apply Lamarckian evolution over the Darwinian model because the Lamarckian model does not discard the learned information. However, Darwinian evolution can be more useful when basins are large or when the algorithm does not include a mutation operator **[14]**.

Regardless of the evolution type used, Lamarckian or Darwinian, the population members with the best fitness are the most likely to be chosen for reproduction. Selections can be directly proportional to fitness, or they can be based solely on a

solution's rank within its generation. Parent selection is often a random process that has its probabilities based on solution fitness **[21]**. In a traditional genetic algorithm, the crossover operator is applied immediately after parents are selected.

II.C.1.b Crossover

A crossover operator in a genetic algorithm is designed to combine attributes of two parents in creating new members of the population. Normally only one or two children will be created from two parents, and the biological basis for genetic algorithm does not seem to allow more than two parents for a single child. Several forms of crossover were considered during the design of the proposed algorithm **[20]**.

One of the most basic crossover methods is uniform crossover **[20]**. In uniform crossover, a child is created from equal parts of two parents. This would be analogous to a child having his or her mother's hair, but father's eyes, mother's nose, but father's mouth and so on, all the way down to the feet. In a combinatorial problem (described further on in this section), uniform crossover is often not an ideal option. For illustrative purposes, consider the following rows taken from a potential solution to a Sudoku puzzle. Each number represents the digit placed in its corresponding cell on the Sudoku grid. For example, Row 1 would have the number 1 in its first cell.

$$Row\ 1 \quad < 1, 2, 3, 4, 3, 6, 9, 7, 8 >$$

$$Row\ 2 \quad < 1, 1, 2, 3, 5, 6, 7, 8, 9 >$$

Notice that Row 1 has duplicate 3's in its third and fifth slots, and Row 2 has duplicate 1's in its first and second slots. If fitness is defined as the number of duplicates

in a given solution, both of these rows would have fairly good fitness values individually due to each of them only having one duplicate. However, if a crossover operator used them for uniform crossover, the child would have a much worse fitness value than either of the parents as shown in Figure 3.

$$Row\ 1 \quad < \mathbf{1}, 2, \mathbf{3}, 4, \mathbf{3}, 6, \mathbf{9}, 7, \mathbf{8} >$$

$$Row\ 2 \quad < 1, \mathbf{1}, 2, \mathbf{3}, 5, \mathbf{6}, 7, \mathbf{8}, 9 >$$

$$Child \quad < 1, 1, 3, 3, 3, 6, 9, 8, 8 >$$

Figure 3: Example of uniform crossover

The child in the previous example would have more than double the duplicates (and therefore double the fitness value) of either of its parents. Applying uniform crossover resulted in a population member that has a fitness value that is much worse than either of its parents' fitness values.

Single point crossover is another simple way to create new children from two parents. An example is below in Figure 4. An initial 'cut' point is determined at random, and the beginning segment of one parent is attached to the ending segment of the other. Again, consider the same two rows from a Sudoku puzzle as an example.

$$Row\ 1 \quad < \mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{3}, 6, 9, 7, 8 >$$

$$Row\ 2 \quad < 1, 1, 2, 3, 5, \mathbf{6}, \mathbf{7}, \mathbf{8}, \mathbf{9} >$$

$$Child \quad < 1, 2, 3, 4, 3, 6, 7, 8, 9 >$$

Figure 4: Example of single point crossover

Notice how the number of duplicates in the child is equivalent to the duplicates in each parent. This will not always be the case, but single point crossover can often yield better results than uniform crossover in a combinatorial problem **[20]**. This is due to the fact that each parent may have a segment that is a completely correct permutation. This type of crossover is more useful when applied to a problem where the ordering of a single permutation is the process by which a final answer is obtained.

Another form of crossover that lends itself more to the combinatorial problem domain is 2-point crossover. Two endpoints of a genotype segment are selected, and two children are created. The segments are then swapped between genotypes, which generates two new members of the population. In the form of crossover shown below in Figure 5, a 'fixing' operator is applied to the new generation after the initial crossover in order to iron out duplicates – making the child a valid permutation **[21]**. The fixing operator could also be applied to other types of crossover, such as single point crossover, and it probably should be used for combinatorial problems that require every population member to be a valid permutation.

$$Row\ 1 \qquad < 1, 3, 6, \mathbf{5}, \mathbf{4}, \mathbf{2}, 7, 8, 9 >$$

$$Row\ 2 \qquad < 6, 4, 3, \mathbf{7}, \mathbf{1}, \mathbf{5}, 2, 8, 9 >$$

$$Child \qquad < 1, 3, 6, 7, 1, 5, 7, 8, 9 >$$

$$Legal\ Child \qquad < 1, 3, 6, 4, 2, 5, 7, 8, 9 >$$

Figure 5: Example of two point crossover

Arithmetic crossover (described below) is a useful tool in continuous optimization problems.

As mentioned above, many forms of crossover do not work as well for combinatorial problems compared to continuous problems. This is due to the fact that the combination of two completely dissimilar parents often yields a child with poor fitness, regardless of the parents' fitness quality. For combinatorial problems, the difference degree method **[23]** allows an algorithm to use crossover methods while addressing this issue. In difference degree crossover, after two parents are selected, all of their individual elements are compared. If the difference is greater than a set threshold percentage, an alternate parent set is selected. This helps limit crossover to 'couples' that have a sufficient amount of similarities to produce useful children.

II.C.1.c Mutation

A mutation operator introduces an essential element of randomness into the search algorithm. The goal of the operator is to apply occasional changes to members of the new generations as they are created. This allows the algorithm to discover different areas in which to search – otherwise it would be permanently limited by the starting population's 'gene pool'. For example, if every initial population member for a Sudoku puzzle had its empty cells filled with 1's, there would be no way to reach the optimal solution. All of the genetic material passed from parents to children would be completely incorrect, regardless of the crossover method used.

$$Row\ 1\ --< 1,1,1,1,1,1,1,1,1 >$$

$$Row\ 2\ --< 1,1,1,1,1,1,1,1,1 >$$

$$Child\ --< 1,1,1,1,1,1,1,1,1 >$$

While the above example is quite extreme, it illustrates the need for mutation for diversity preservation. Somehow the elements within the row need to change, but crossover will not provide that change, but mutation will introduce variants into the gene pool.

Typically, the probability of a mutation occurrence is very low – often as low as 1% for any given population member. In a traditional genetic algorithm (one with binary representation), mutation would require merely a single bit flip. For a combinatorial problem such as Sudoku, mutation will often involve a random reordering of nodes **[5]**. Figure 6 is an example of mutation applied to a population member defined as a row within a Sudoku puzzle.

$$Original\ Offspring < 1,2,3,4,5,6,7,8,9 >$$

$$Mutated\ Offspring < 1,2,3,4,7,6,5,8,9 >$$

Figure 6: Example of mutation

The mutated offspring had its 5th and 7th values exchanged. This change was not based on any previous genetic information. It was a random alteration that the mutation operator imposed on the offspring.

II.D Simulated Annealing

Simulated annealing is another powerful optimization algorithm. It is a combination of global and local search techniques, and it is generally regarded as an effective method to reach an acceptable (if not optimal) solution **[14]**. Simulated annealing is based on the phenomenon that occurs when cooling certain metals – if done correctly, the metal reforms with a purer lattice structure than it was before it was heated. The molecules move from a high heat (and therefore high energy) state to a low heat state, where they can settle in to an ideal structure.

Unlike genetic algorithms, simulated annealing does not maintain a population. Instead, it sustains one solution from start to finish, choosing whether or not to accept a 'move' to a new solution based on its energy state. Whether the algorithm is in a high or low energy state depends on the 'temperature'. When starting the algorithm, a high initial temperature is used. Each iteration brings a reduction to the temperature, with the amount of temperature reduction being dependent on a user-defined cooling schedule. Two common cooling schedules are linear and proportional. In linear cooling, the temperature is reduced by a set *number* of degrees each iteration, and in proportional cooling the temperature is reduced by a set *percentage* of degrees each iteration.

In simulated annealing, when the algorithm is in a high energy state, the probability of accepting a move to a solution with a worse fitness than the current solution is relatively high. In a low energy state, it is highly unlikely (although not

impossible) for the algorithm to allow a move to a solution with a poorer fitness. A move to a solution with a *better* fitness is always allowed within simulated annealing as described in the equation below **[14]**. The current temperature is represented by 'Temp', and the change in fitness from the current solution to the proposed solution is represented by Δfit.

$$P(accept\ move) = \begin{cases} 1 & if\ \Delta fit \geq 0 \\ e^{\frac{\Delta fit}{Temp}} & if\ \Delta fit < 0 \end{cases} \tag{1}$$

II.E NP-Complete

Simulated annealing and genetic algorithms are solving methods that are often applied to problems of the NP-Complete class. A decision problem, X, is considered NP-Complete if it is of the Non-Deterministic Polynomial Time (NP) type and if every other problem in the NP set can be reduced to X. Sudoku, the Traveling Salesman Problem (TSP), and Tetris are all well-known examples of NP-Complete problems **[24]**.

Another famous combinatorial problem that is also of the type NP-Complete is the N-Queens problem, in which the objective is to place n queens on a $N \times N$ sized chessboard in such a way that no queen can 'take' another queen with a single horizontal, vertical, or diagonal move. A solution to the N-Queens problem is shown in Figure 7.

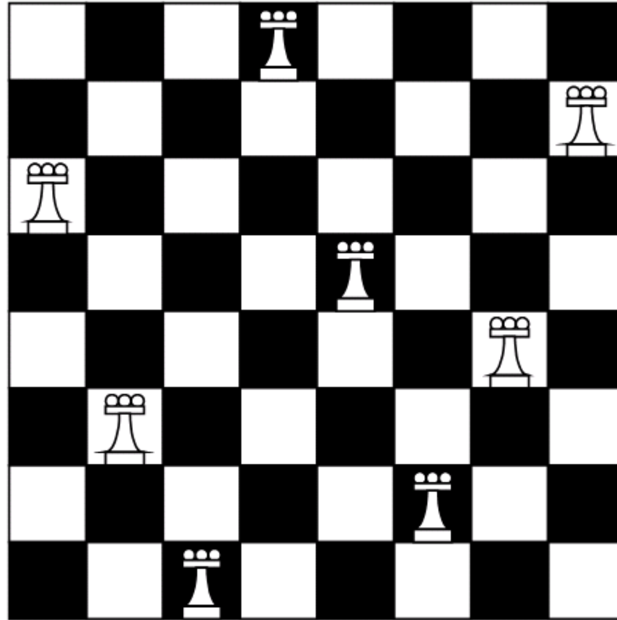


Figure 7: Example of a solved N-Queen problem [25].

Some practical examples of NP-Complete problems are scheduling and network routing **[14]**.

II.F Combinatorial vs. Continuous Optimization

Both genetic algorithms and simulated annealing are two powerful searching algorithms that can be applied to different types of optimization problems.

There are two distinct types of optimization problems – combinatorial and continuous. A combinatorial problem is concerned with the reordering of a given set of elements in order to achieve an acceptable solution. For Sudoku, this is essentially a search for permutations of {1, 2…9} that satisfy the row, column, and mini-grid constraints. A continuous problem is concerned with exploring a continuous range of values (possibly to the extent of $(-\infty, \infty)$) in order to discover the solution.

In general, the combinatorial optimization problem set is considered to be more difficult to handle than the continuous optimization problem set. There are several reasons for this. For a continuous optimization problem (such as one of DeJong's functions – Figure 8 **[26]**), there is a full range of options available for crossover operators as a result of the function's use of a practically infinite number set. With continuous problems, solving techniques that involve using the gradient of the fitness with respect to the change of the population can be applied. Continuous problems have easily definable neighborhoods and local minima/maxima. Combinatorial problems do not **[14]**.

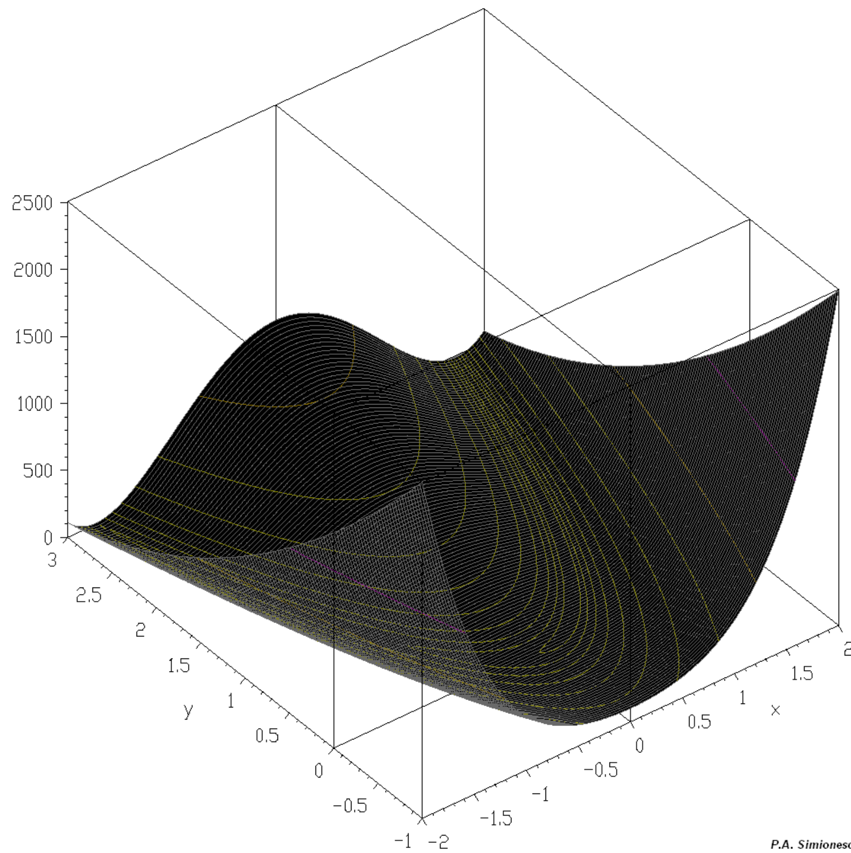


Figure 8 [26]: Rosenbrock's Valley – DeJong's 2nd function, described by the following equation:

$$\sum_{i=1}^{n-1} \left(100 \cdot (x(i+1) - x(i)^2)^2 + \left(1 - x(i)\right)^2 \right) \quad (2)$$

One of the more useful crossover operators for a continuous problem is the arithmetic crossover, where the new value 'c' is equal to the mean of the previous parents' values 'a' and 'b' that occupy the same place within a chromosome.

$$c = \frac{a+b}{2} \tag{3}$$

For example, if the optimal solution were '3.5' and the parents selected had values of '3' and '4', the algorithm would discover the optimal solution simply by performing arithmetic crossover. However, this averaging strategy would not work if the given problem were of a combinatorial nature, because a combination of this type could give a result that is not permitted.

Consider the case of the traveling salesman again. If two cities on the route are St. Louis (a) and Oklahoma City (b), using a crossover operator to yield a midpoint may land the salesman in Joplin, MO (c). If the salesman is expected to travel to both St. Louis *and* Oklahoma City and *not* to Joplin, this would be unacceptable. The arithmetic crossover operator would result in a point that is not an option in the given problem.

Another key difference between combinatorial and continuous problems is the concept of 'direction' **[14][27]**. In a continuous problem, if the algorithm moves from 2.0 to 2.1 and finds that 2.1 is a better solution, it can continue on the same gradient toward 2.2. In a combinatorial problem, if the algorithm switches nodes 2 and 3 to reach a better solution, it cannot use that information to determine that switching nodes 3 and 4

would also be beneficial. Many optimization techniques are gradient based, but these algorithms can only be applied to continuous problems due to the lack of direction inherent to the combinatorial problem.

The Figure 9 is a plot that demonstrates the value of being able to use gradient information when searching for a global optimum. Assuming that the objective of the optimization problem is to minimize fitness, and assuming that the plot shows the entire fitness function for the problem, the global optimum will have a fitness of 0 at a value of 13. Suppose the algorithm found its way to the corner of the plateau at value 11. If it took a small step to the right, it would calculate the gradient to be negative. The algorithm could potentially follow the same monotonically decreasing path until it found the global optimum at a value of 13.

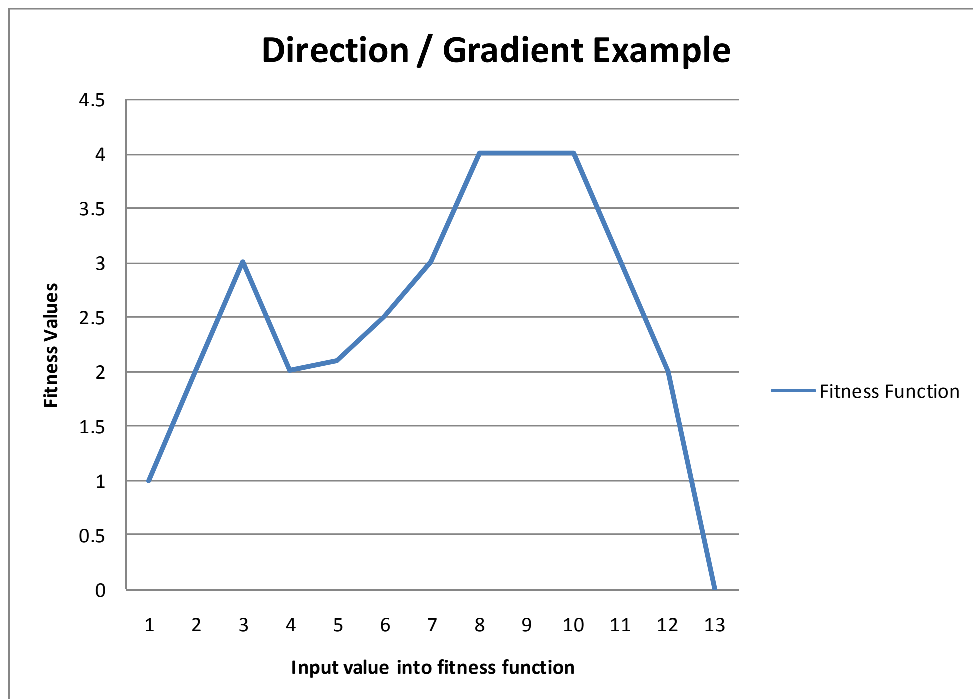


Figure 9: Example of gradient usage

In a combinatorial problem, an algorithm cannot make use of the gradient information, in part because the fitness function cannot be represented as a continuous curve with a slope. Again, the lack of an exploitable ‘direction’ for combinatorial problems limits a programmer’s options.

Often coupled with direction is the idea of step size. In a continuous problem, an algorithm can use step sizes that are very large or very small, with the limits dependent only on the system on which the algorithm is being run. This allows the optimization algorithm to make great leaps away from its current neighborhood to explore a different area. It also allows the algorithm to take small steps in order to fine tune a solution.

A combinatorial problem can make use of different step sizes, but only by increasing or decreasing the number of nodes swapped per step, since there is no way to perform fractions of a swap or permutation. However, increasing the step size for a combinatorial problem from one swap to two greatly increases the gap between the original solution and the new solution **[14]**.

For combinatorial problems, a single step cannot move a solution out from a basin. However, a solution can still improve its fitness in one step by moving from one basin to another, even without being able to escape **[14]**. Below, table 1 outlines some of the key differences between continuous and combinatorial optimization problems.

Table 1: Comparison of continuous vs. combinatorial problems

	Continuous	**Combinatorial**
Able to use direction information	Yes	No
Able to fine tune step size	Yes	No
Well defined local search	Yes	No
Easy to scale/define neighborhoods	Yes	No

One of the characteristics of the combinatorial problem type is that every possible solution is essentially reordering a given set of elements. For example, in Sudoku, an algorithm can sort through different potential solutions by shuffling the numbers {1, 2…9} in each row. This limits the styles of crossovers and mutations available to the algorithm. Arithmetic crossover would not be possible for a Sudoku puzzle, because it could result in numbers that are not part of the set of allowable values in a Sudoku grid.

Many heuristic algorithms have been applied to the N-Queens problem with varying degrees of success. Genetic algorithms performed fairly well when applied to the N-Queens problem **[9]**, and so the transition to Sudoku – a similar type of problem – is quite logical.

There are many different parent selection methods from which to choose for a combinatorial problem, including roulette, tournament, and partially matched. Roulette selection is a simple and effective selection method. Parent selection can also be in direct proportion to the fitness of a solution **[20]**.

Local minimums are a significant stumbling block for many genetic algorithms when it comes to combinatorial problems. With a limited population, it is quite possible to get stuck in a local minimum in which swapping one or two cells (in an N-Queen or Sudoku problem) would result in a worse fitness value than the previous solution. As stated previously, a combinatorial problem cannot remove itself from a basin with a single swap. A way to counter this issue of reaching 'dead-ends' is re-initializing the starting population for the algorithm. However, one must be careful to limit the use of this method, as it can severely impact the efficiency of the algorithm **[11]**. If an algorithm restarts itself often enough, it acts similar to a random search, which is ineffective at best.

II.G Sudoku

Sudoku is a logical puzzle in which a player attempts to fill in all blanks with the numbers {1, 2…9} such that no row, column, or 3×3 mini-grid contains a duplicate number. Sudoku puzzles are actually a subset of an older puzzle called a 'Latin Square.' According to Will Shortz **[24]**, Sudoku was likely developed in 1979 by Howard Garns, and it was initially called 'Number Place.' Dell Magazines published it, but it did not catch on in the U.S. initially. In 1984, Sudoku was introduced in Japan by Nikoli – a publisher that specializes in logic puzzles. Even then it did not gain in popularity until Nikoli imposed restrictions on the game: no more than 32 clues were allowed, and the puzzles were made to be symmetrical 180 degrees around the center cell. There are approximately $6.67 * 10^{21}$ valid Sudoku grids **[28]**.

The lowest possible number of starting hints that can provide a unique solution is 17. It has not been proven that there are no puzzles with 16, but to date, none have been found. Another prerequisite for a unique solution is using 8 of the 9 possible values when giving hints. If only 7 values are used (say, the numbers 1 through 7), then any solution found would not be unique. This is due to the fact that another trivial solution could be found just by exchanging the two unused numbers (the 8 and the 9) **[29]**.

For a player, there are many simple logical checks to perform in order to find the values that belong in each of the blank cells. For the simplest of puzzles, the solution can be found by using just a straightforward process of elimination. For more complicated grids, the user needs to identify multiple possible values in each of the cells and proceed from there. There are also very complex logical solving methods, with names like 'X-Wing', 'Y-Wing', and 'Death-Blossom' **[22]**.

Puzzle difficulty often hinges on which solving techniques are *required* in order to complete a Sudoku without guessing. Many times this will relate to the number of starting hints, but not always **[1]**. Difficulty of Sudoku puzzles varies drastically from one puzzle to the next, and a puzzle's given difficulty level (e.g. – 4 Star, 5 Star) does not always accurately indicate how challenging it may be **[5]**.

Standard (read: non – evolutionary based) algorithms can use this same type of logic or a combination of logic and brute force in order to solve Sudoku puzzles **[1]**. This approach is feasible for solving one grid, but when attempting to apply straight logic or

the brute force and logic combination to multiple puzzles simultaneously, it can begin to become unwieldy. Figure 10 is an example of a solved Sudoku puzzle.

9	7	1	5	8	6	4	3	2
3	8	4	7	2	1	5	9	6
5	2	6	3	9	4	1	7	8
8	3	9	2	7	5	6	4	1
2	4	5	1	6	9	7	8	3
6	1	7	4	3	8	9	2	5
1	5	2	8	4	7	3	6	9
7	9	8	6	1	3	2	5	4
4	6	3	9	5	2	8	1	7

Figure 10: Sample solved puzzle. This is the same puzzle as in Figure 1.

Sudoku has become very popular in the US over the past few years. It can be found in many magazines and newspapers, and it is available for handheld gaming systems such as the Nintendo DS, the Sony PSP, and even cell phones.

Sudoku is in the NP-Complete problem class, which indicates that it is a difficult problem to solve consistently. For combinatorial problems, genetic algorithms are typically designed to quickly *approach* the optimal solution, because waiting to find the global optimum is not always practical. In fact, if the global optimum is unknown, the algorithm would never have a set (problem-defined) stopping point.

For Sudoku, these guidelines do not apply. The objective is to reduce the number of duplicates in every row, column, and mini-grid to zero. The optimum for any given

problem is ‘zero duplicates’. In one respect, knowing the objective before beginning is quite helpful, since the algorithm has a very clear stopping point. However, designing a Sudoku solving genetic algorithm only to approach the optimum but not reach it is as impractical as it is unacceptable. No Sudoku solver should be considered complete if it frequently solves puzzles down to one or two duplicates but not to the optimal solution.

Several papers have been published on using evolutionary algorithms to search for Sudoku solutions, including **[4]** and **[5]**.

In **[4]**, the authors tested many different novel forms of crossover, including what they called ‘product geometric’ crossover. Their puzzle representation was a single array of 81 integers, with every 9 integers making one row of the Sudoku grid. Starting solutions were initialized by inserting random numbers from the set {1, 2…9} into every blank cell, or by creating a random valid permutation of the numbers 1 to 9 in each row. The population size was set to 5000, and the top half of the population was retained after each iteration. Most of the crossover operators tested in **[4]** were applied to individual rows, with the exceptions of two-point crossover. The authors designed the algorithm to apply both point mutation (changing one number to a random number from the set {1, 2…9}) and swap mutation. The algorithm stopped its search after making no progress for 20 generations.

In **[5]**, the authors used a somewhat different approach to solving Sudoku puzzles. The puzzle representation was the same as the one in **[4]**, but their crossover and

mutation methods were different. The population size was merely 21, and they only applied elitism (saving the best solutions) to a single population member. The authors used two types of mutation, swap mutation and ‘cataclysmic mutation’, which is a random reset. Rather than checking for the number of duplicates in each row, column, and mini-grid, the algorithm in **[5]** checked to make sure that each individual row and column had values that both summed to 45 and had a product equal to 9!. The fitness function also verified that every value from the set {1, 2…9} appeared in each row and column. The algorithm would only stop if a solution was found.

II.H 3D Sudoku

After extensive research, it seems evident that there is not much (if any) information on the problem of 3D Sudoku. The 3D variety of Sudoku is just an extension of the well known NP-Complete problem of 2D Sudoku, and an example is shown in Figure 11. Six individual Sudoku puzzles are used as the faces of a cube. A requirement of the puzzle is that adjacent edges must match.

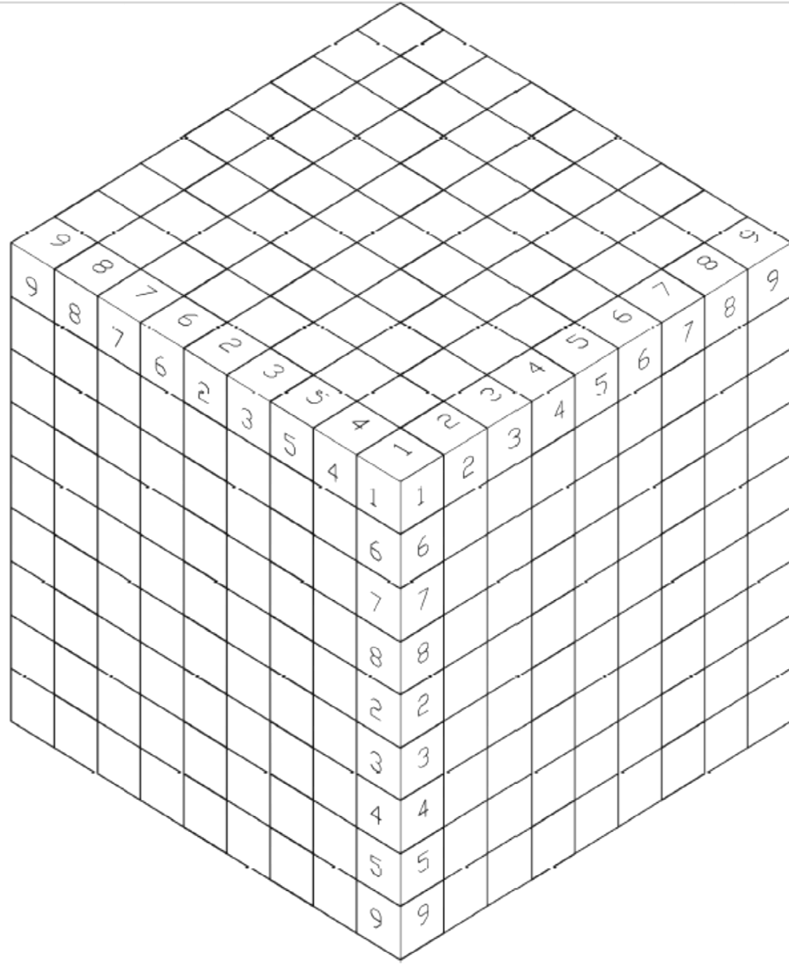


Figure 11: Example SudoKube with three solved edges

For the purposes of this research, the 3D puzzle does not follow all of the traditional Sudoku guidelines. For a standard Sudoku puzzle, there tends to be only one solution. Also, the blanks are typically rotationally symmetrical about the center cell on the grid **[1]**.

For this 3D representation, each individual side may have more than one solution. This is due to the need to present a problem in which the goal is to discover simultaneous solutions with intertwined constraints. If the sides could be solved individually, then

there would be no guaranteed need for a 3D solver. A single Sudoku solver could be applied to each side in turn, the edges would match by default, and then the cube would be solved. However, if a side has multiple possible solutions, there is only a slim chance that solving each side individually would generate a valid 3D solution.

For a SudoKube, the set of six puzzles itself is more limited in the number of solutions available compared to one of its single sides. For example, the number of solutions per side for one particular 'hard' rated puzzle is displayed below (Table 2) **[22]**. The reason an entire cube has a reduced number of solutions is due to the fact that having a side with only one solution essentially increases the number of givens for adjacent sides, which reduces the number of possible solutions for the adjacent sides.

Table 2: Number of solutions per side

Side Number	Number of Solutions	Number of solutions when given edges of sides with only 1 solution
Center – 1	1	1
Top – 2	143	3
Right – 3	4	2
Left – 4	1	1
Bottom - 5	8	4
Back - 6	44	15

CHAPTER III

DESIGN METHODOLOGY

III.A Design Introduction

There are several different elements in the proposed genetic algorithm that creates and solves 3-D Sudoku puzzles, called SudoKubes. These elements include a 2D solver, a 3D generator, and a 3D solver. There is no existing standard publication of 3D Sudoku cube puzzles. Therefore, the algorithm was designed to first *create* six 'linked' Sudoku puzzles before solving them. The program was coded in Microsoft Visual Studio 2005 using C#.

The creation process has several key steps. First, the 2D puzzle generator sets up an initial blank puzzle. Next, the remainder of the cube's sides are linked to the first puzzle and solved sequentially. This yields a solved SudoKube with touching edges set equal to each other. Finally, the algorithm removes a random amount of numbers from the completed puzzle so that it can be solved. The pattern of removal and significance of the amount of numbers removed in order to create a puzzle with a unique solution is an interesting problem, but it will not be addressed here.

III.B 2-D Solver

In order to effectively generate SudoKubes, a 2D solver was developed. A benefit to having a 2D solver was being able to research and fine-tune operators that

would also work on SudoKubes. This provided useful information when coding the 3D solver.

For the 2D problem, each member of the population is represented by a filled Sudoku grid with dimensions of 9×9. The given numbers for each puzzle are static, so they will not change as the algorithm works towards a solution. Each grid space that does not contain a given (the ones that would be blank if the puzzle was taken from a newspaper) holds a number from the set {1, 2…9}.

Fitness is calculated by counting the number of duplicates in each row, column, and mini-grid. The goal is for the fitness of each row, column, and mini-grid to be as low as possible, with an optimum value of zero. Fitness is calculated separately for rows, columns, and mini-grids. Total fitness for a given population member is the sum of each individual fitness value for every row, column, and mini-grid. Therefore, a correctly solved puzzle will have a total fitness of zero, because there will be no duplicates within the population member. A flowchart for a basic genetic algorithm is shown below in Figure 12.

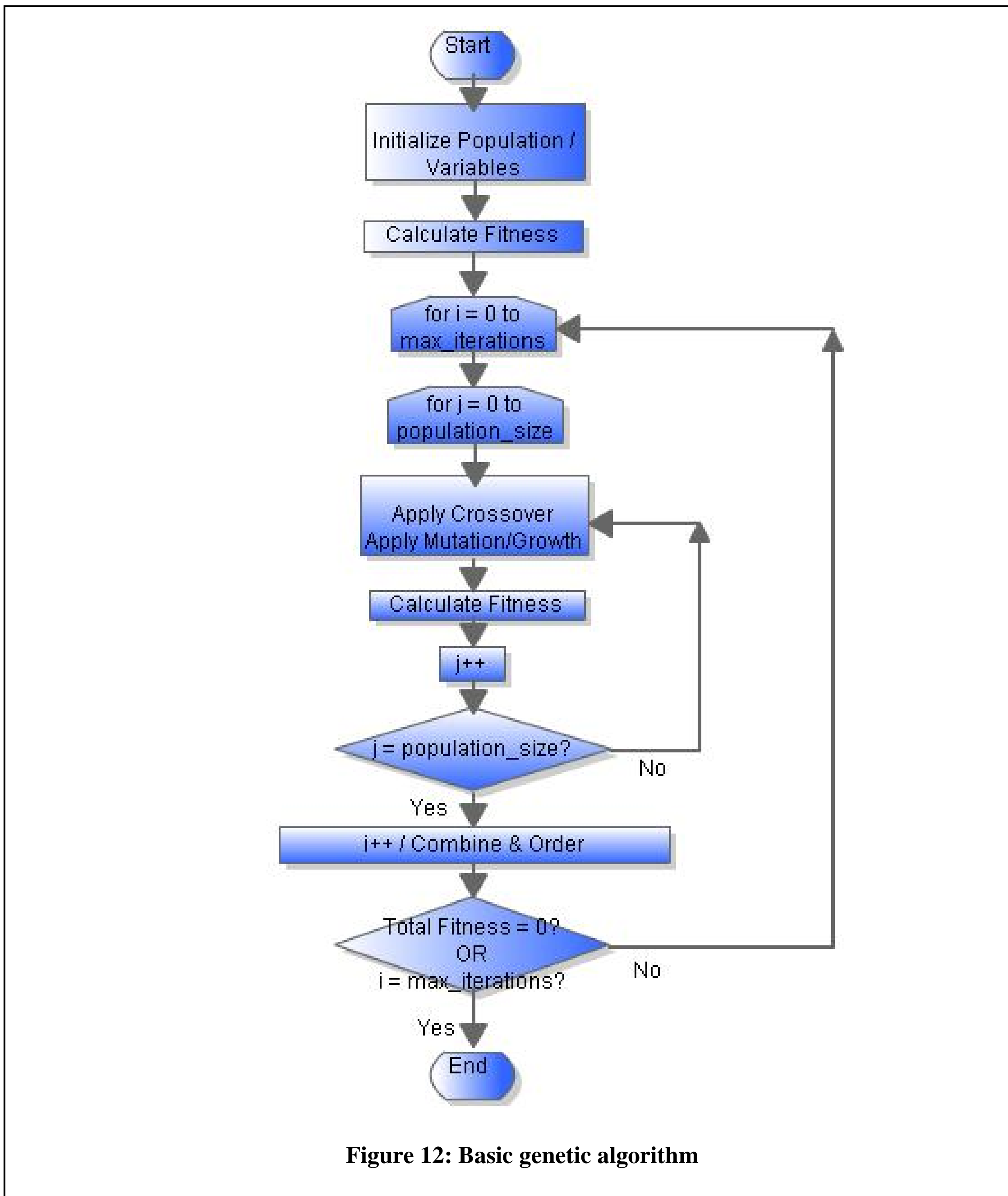


Figure 12: Basic genetic algorithm

III.B.1 Genetic Operators

III.B.1.a Parent Selection

The 2D algorithm uses a modified version of a parent selection method taken from **[10]**. Originally, the algorithm selected a random member from the top half of the

population and another member from the whole population to determine a pair of parents. Although somewhat effective, this approach did not fully utilize the ability of a genetic algorithm to select the best population members more frequently than unfit population members. In the final algorithm, the probability of selecting a population member *k* from a population of size M is given by the following equation:

$$p(k) = 2 * \frac{M-k}{M*(M+1)} \qquad (4)$$

The population is ordered from best to worst before parent selection occurs, so the population member with the best fitness (at the top of the ordered list) is approximately twice as likely to be selected for reproduction as the population member halfway down the list. Two parents are selected to create one child, and the selection process occurs once for each population member.

After the crossover step is complete, the new generation of solutions is combined with the previous generation, and the fitness of each potential solution is calculated. Next, the combined solution set is ordered from best to worst. Finally, the top half – the half with the best fitness – of the combined solution set is kept and used as the next parent generation.

III.B.1.b Aging

After much testing of the standard 2D solver, an aging operator was introduced. The operator was designed to help the algorithm escape local minima without having to

resort to ‘cataclysmic mutation’, which is essentially a random reset **[5]**. Each population member has an age assigned to it. When a child is generated, its age is set to zero. For every generation a particular solution survives, its age is incremented by one. A solution that has reached a predetermined age threshold is replaced with the next best solution that is not due to be retained in the top half of the population.

Figure 13 illustrates the aging operator. For clarity and brevity, the population size is only 5 and the maximum age is 5. After performing crossover to make a new generation and then evaluating the fitness of each solution, the solutions are sorted in order of ascending quality. Typically, the top half of the list is retained and used for creating a new generation, while the bottom half of the list is discarded.

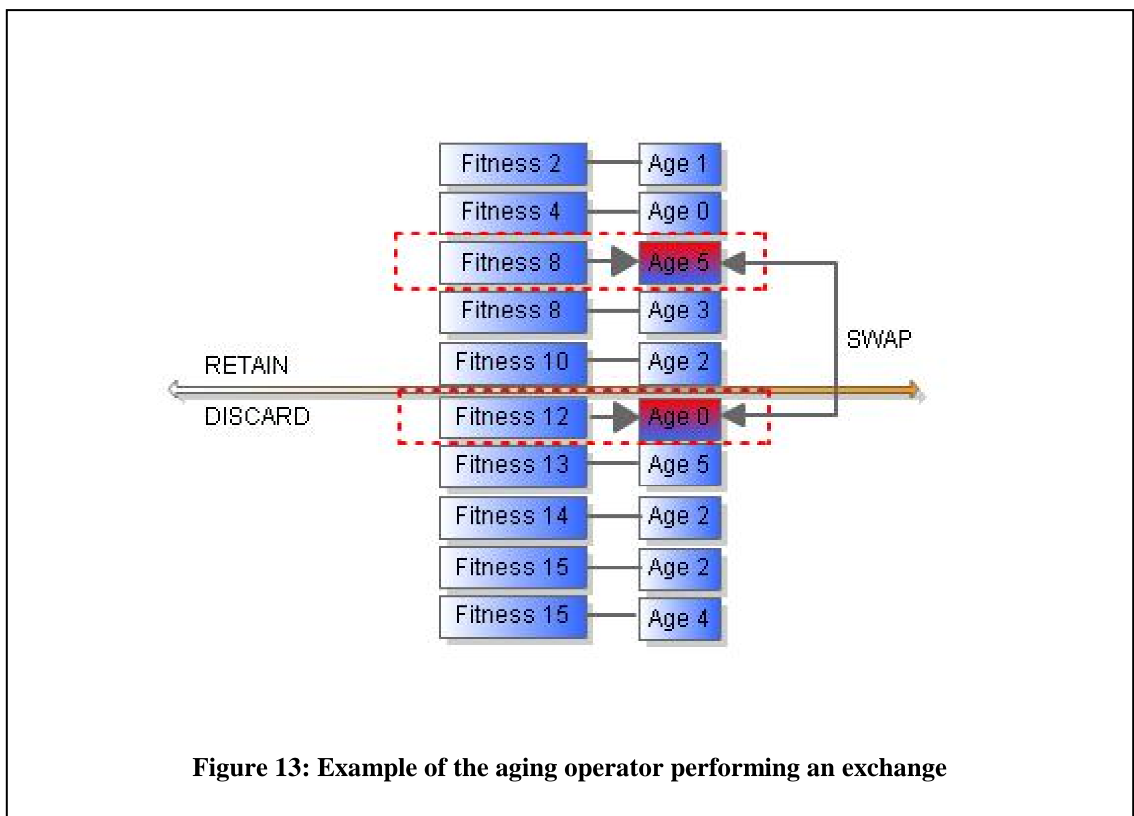


Figure 13: Example of the aging operator performing an exchange

In the above example, the third population member on the list has aged to the limit set in the algorithm. The aging operator determines that it must be discarded. The next best member of the population (member 6) then takes the place of the 'elderly' individual (member 3).

III.B.1.c Crossover

There are three forms of crossover used in the proposed genetic algorithm: row crossover, column crossover, and grid crossover. These are all applied with equal probability to two parents in order to create one child. Figure 14 is a chart that outlines the crossover operator.

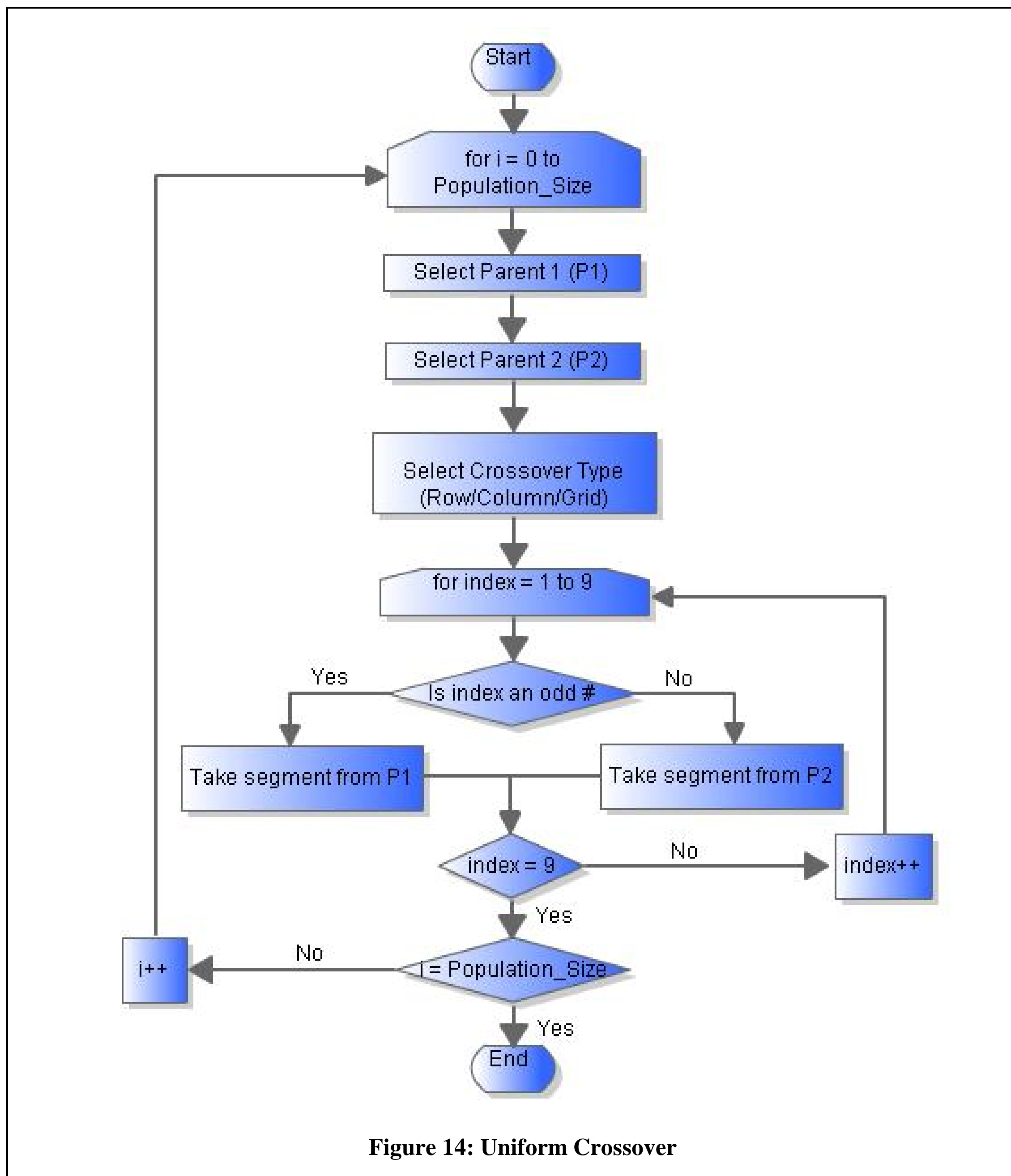


Figure 14: Uniform Crossover

In row crossover, rows 1, 3, 5, 7, and 9 are taken from the first parent, and rows 2, 4, 6, and 8 are taken from the second parent. The rows hold the same positions in the resulting child as they held in their respective parents. For example, row 1 from parent 1 would be row 1 in the child. An example of this concept is shown in Figure 15.

Parent 1	Parent 2	Child
1		1
	2	2
3		3
	4	4
5		5
	6	6
7		7
	8	8
9		9

Figure 15: Row crossover

Column crossover works in the same fashion, taking alternate columns from their respective parents and combining them into a child. Mini-grids are also numbered 1-9 starting from the top left corner of the puzzle and ending at the bottom right corner. As with row and column crossover, alternate mini-grids are taken from each parent to create a child.

Another tested method was fitness based crossover, which is discussed in the 3D Sudoku section found further in this thesis.

III.B.1.d Mutation / Natural Growth

Although not always beneficial, there is something to be said for applying some problem-specific logic to solving a Sudoku puzzle, even with a heuristic algorithm such as this. For the proposed algorithm, the logic is implemented in the mutation operator. Concepts from simulated annealing are also applied within this operator. As with crossover, there are three types of mutations available for application. Row, column, and mini-grid mutation all occur with equal probability.

To begin, a random number from [1, 100] is generated. If the number is below the threshold set by the user, the operator will perform standard mutation. Otherwise it will perform 'Natural Growth' (described below) on the given population member.

The standard mutation operator selects a number from the set {1, 2 … 9} and places it in a random cell that is not specified by the initial problem. This type of operation occurs infrequently. The likelihood of mutation is predetermined. Generally, the threshold that defines the frequency of standard mutation is set so that mutation occurs between 1 and 3 percent of the time. This component of the mutation operator is typical to many genetic algorithms in combinatorial problems, and it often results in an invalid row, column, or mini-grid. However, the placement of a random number, while not always *immediately* helpful, often helps the algorithm explore regions of the search space that may be inaccessible through crossover alone.

If the random number does *not* fall below the given value, the second component of the mutation operator, called 'natural growth', is applied. It is not a true mutation operator, as it is guided by problem logic and puzzle-specific constraints. It uses the simple rules of the Sudoku problem to 'mature' a solution. The algorithm randomly chooses to look at row, column, or mini-grids, and it performs growth on the selected sections.

Row, column, or mini-grid 'growth' is selected at the start of the natural growth operator – all with equal probability. If 'row' is selected, the operator references the

fitness value of each row to see if it contains duplicates. If at least one duplicate is present, the operator replaces one of the duplicate numbers with a number from the set {1, 2…9} that is not already used within the row. If there are no duplicates within a row (or column, or mini-grid), the operator will generate a second random number from [1, 100]. If the random number is less than the current set mutation rate (e.g.: if the mutation rate is 3, and the random number is a 1), then a random swap is performed.

A swap takes two existing values in a given puzzle segment (row, column, or mini-grid) and exchanges their positions. This ensures that the segment in question maintains its fitness value of zero, but it also allows the population member to change in a way that could potentially aid the search. As mentioned previously, numbers given in the original problem are static and will not be altered by *any* operators.

The mutation/natural growth operator goes through several iterations before moving to the next member of the population. This allows each individual member of the population to realize its full potential. The growth operator eliminates duplicates in any segment on which is it working, but each step in the growth process has the potential to introduce more duplicates in *other* segments. Fortunately, the crossover operator and the iterative nature of the growth operator solve the problem of these duplicates.

For standard mutation, any non-static number could be replaced with any number from {1, 2…9}. For row *growth*, one of the 7's or 6's in the first row of Figure 16 would

be replaced with a 1 or a 2 (the unused numbers for that row). The chart following the Sudoku example (Figure 17) describes the mutation process.

3	8	7	7	6	6	9	4	5
9	1	4	5	3	8	6	2	7
6	5	2	7	9	4	1	3	8
4	7	3	2	5	9	8	6	1
8	9	5	6	4	1	3	7	2
1	2	6	8	7	3	4	5	9
2	3	1	4	8	7	5	9	6
5	4	8	9	2	6	7	1	3
7	6	9	3	1	5	2	8	4

Figure 16: Unsolved Sudoku grid with duplicates.

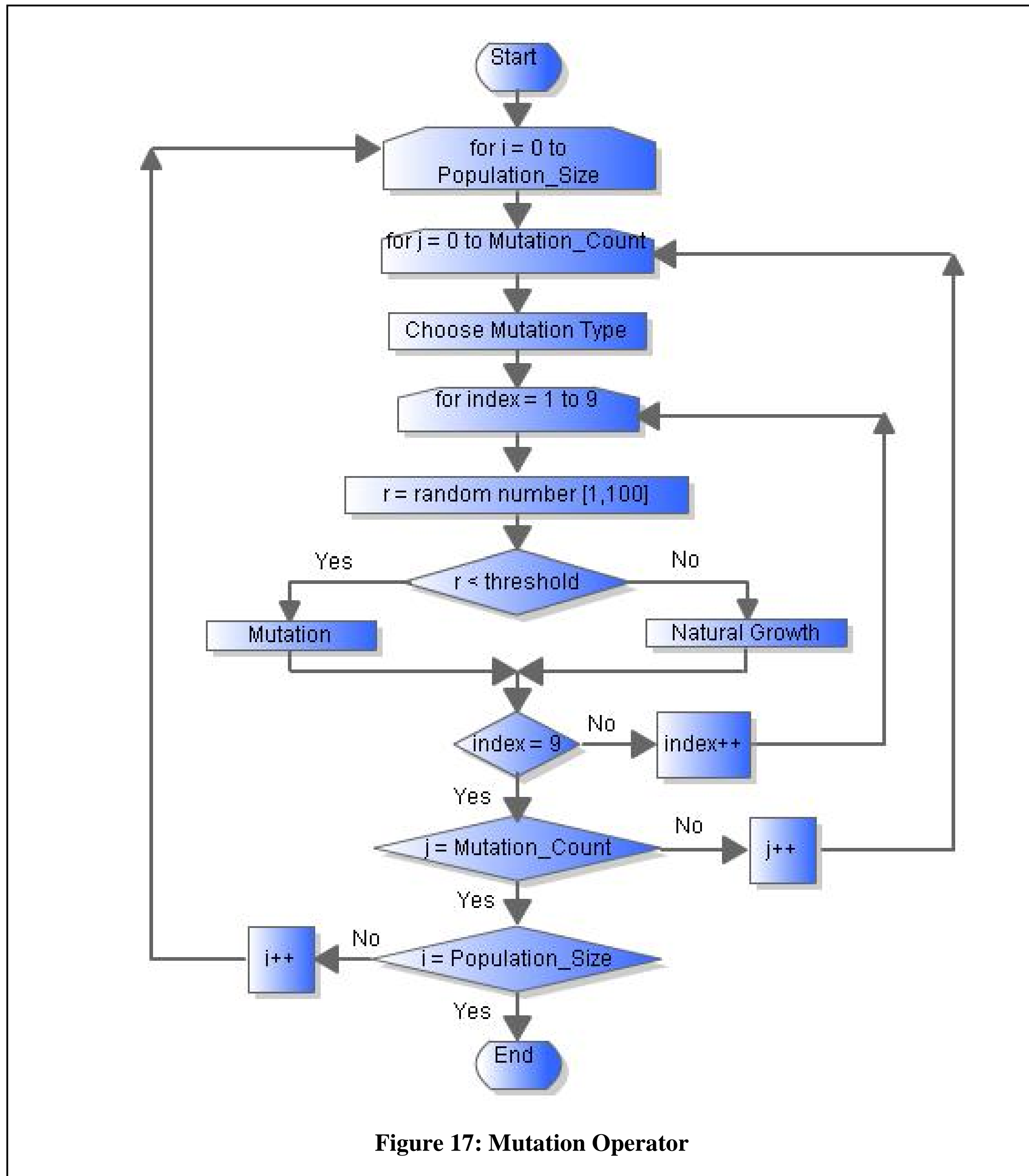


Figure 17: Mutation Operator

III.C Puzzle Generation

Generating Sudoku puzzles is more complex than solving a given puzzle out of a newspaper or book, because it is not difficult to generate unsolvable puzzles if an algorithm places starting hints incorrectly.

Initialization presented an interesting problem when attempting to generate a valid Sudoku puzzle. It is not necessary to explicitly define every cell. Initialization of the puzzle can be done in stages. If one attempts to completely initialize a blank grid in only one step, the process essentially leads to solving a completely blank Sudoku puzzle, which can be difficult for a genetic algorithm due to the very large search space and the availability of multiple solutions.

Originally, if the proposed algorithm was given a blank Sudoku grid to solve, the algorithm would run for an extended length of time before finding a solution. There are ~6.67×10^{21} possible Sudoku grids **[28]**, and therefore there are that many solutions with a fitness of zero – assuming the given puzzle is an empty set of cells. The large number of potential solutions created challenges for the proposed algorithm, because the algorithm was designed to take one puzzle and converge to its *one* optimal solution. With a blank puzzle, there are many optimal solutions. Therefore, the proposed algorithm's selection and crossover operators did not function as intended. The algorithm could select two parents with similar (good) fitness values but entirely different puzzle layouts between them. If crossover were to be performed on these two good (but different) solutions, it would most likely result in a solution with high fitness. As a result of this crossover challenge, the introduction of difference degree crossover became essential. Potentially useful for combinatorial problems in general, difference degree crossover was especially useful in the situation in which each population member had the potential to differ greatly from every other population member. Difference degree crossover prevented solutions that were too disparate from mating.

At first, the puzzle generation component of the proposed algorithm utilized an effective initialization tool in order to overcome the problem of having too many possible solutions. The algorithm randomly generated three mini-grids and placed them in opposite corners and in the middle of the Sudoku grid as shown in Figure 18. Generating a mini-grid was simple. The algorithm placed the numbers {1, 2…9} in a 3 x 3 box to create a valid mini-grid for a Sudoku puzzle.

9	7	1						
3	8	4						
5	2	6						
			2	7	5			
			1	6	9			
			4	3	8			
						3	6	9
						2	5	4
						8	1	7

Figure 18: Randomly generated mini-grids, one of two choices for puzzle initialization.

The key to this particular initialization technique was that no mini-grid interfered with any other mini-grid. Each initialized mini-grid was completely independent of the other two mini-grids. This independence ensured that there would be no duplicates in any row, column, or mini-grid when the solver began its work.

Giving the solver 27 static numbers to start with allowed it to reach a solution quickly and efficiently, but adding difference degree crossover eventually eliminated the need for this particular initialization technique.

Another way to initialize a Sudoku puzzle is by setting the four edges of a puzzle to valid permutations of the numbers {1, 2 … 9} before solving it. Doing this serves the same purpose as initializing the mini-grids, but edge initialization was more effective when forming the linked grids.

III.D Puzzle Combination

In order to fully realize the potential of using a genetic algorithm to solve Sudoku puzzles, 'linked' or '3-D' puzzles were connected and generated while the solver and generator mentioned in the above sections.

First, a 'seed puzzle' is generated and is used as the base for the remainder of the puzzles that are placed around it. Rather than initializing the starting puzzle with three mini-grids, the algorithm initializes all four sides of the Sudoku grid. Each attached puzzle shares an edge with the main grid. This results in top, right, left, back, and bottom puzzles as well as the main (front) puzzle. The auxiliary puzzles (top, right, left, bottom, and back) share sides with each other as well as shown in Figure 19.

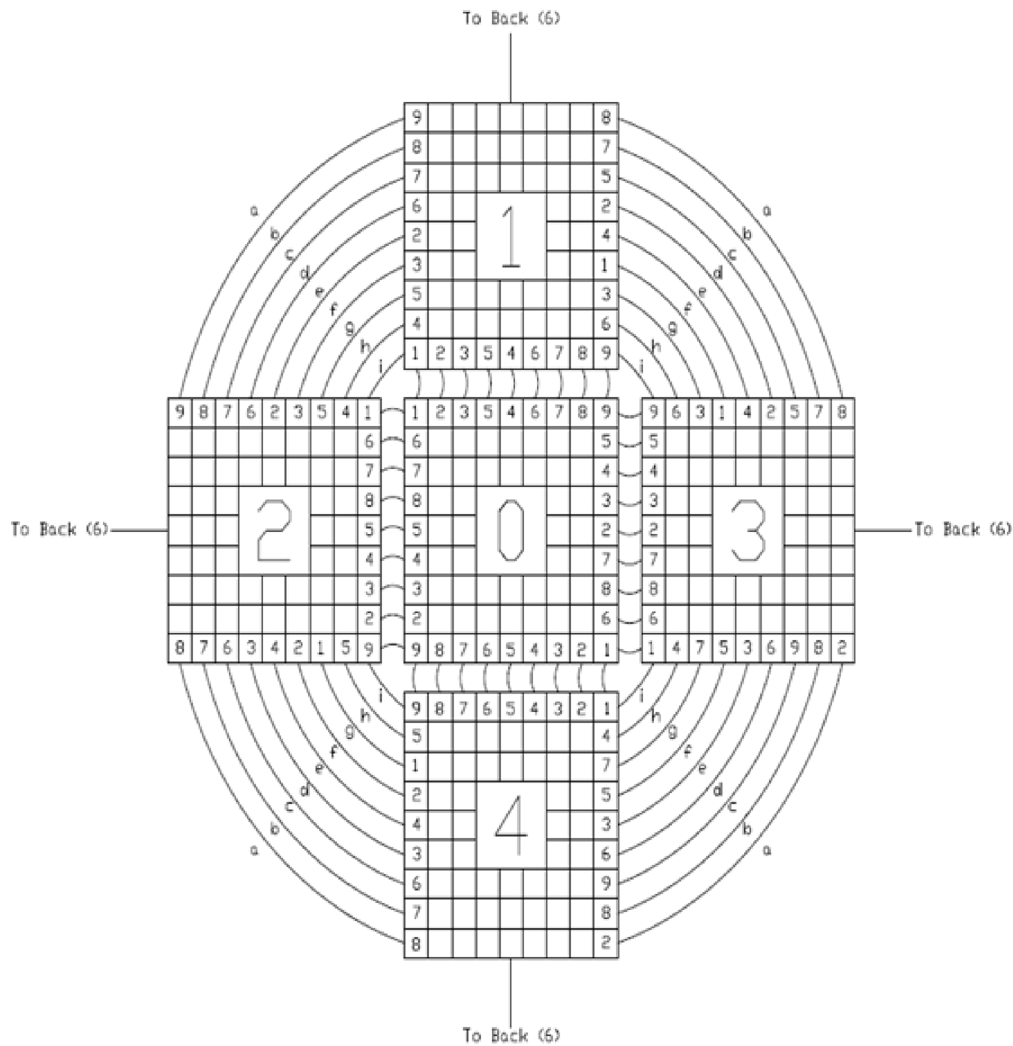


Figure 19: Overview of five linked puzzles. It is displayed like an unfolded cube. For clarity, the puzzle that would be the back of the cube is not shown.

If one were to imagine the puzzles folded into a cube, the sides that would join together to form an edge would contain the same numbers.

In order to create this linked puzzle setup, some constraints must be imposed upon the auxiliary puzzles. For example, in Figure 19, the top row of section 3 cannot contain the numbers 7, 8, 5, or 4 in the two cells immediately to the right of the 9. If it did

contain a 7 or an 8, the bottom right mini-grid in puzzle 1 would be unsolvable, and if it contained a 5 or a 4, the top left mini-grid in puzzle 3 would be unsolvable. Ensuring that the resulting auxiliary puzzles are solvable is imperative when generating a linked Sudoku puzzle.

After the linked puzzles are initialized, the solver is applied to every puzzle, and the result is a set of six solved, linked Sudoku grids. This process is shown in Figure 20.

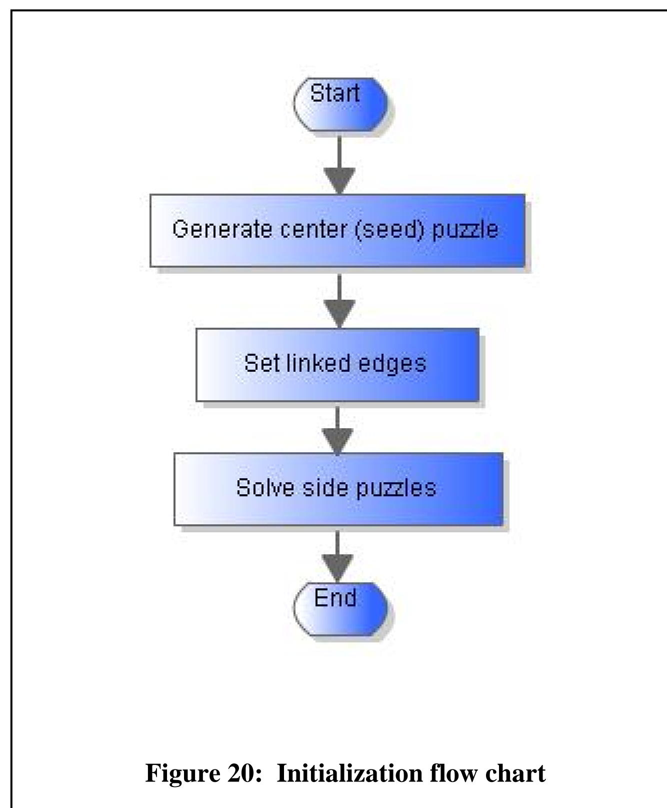


Figure 20: Initialization flow chart

A traditional Sudoku puzzle typically has a unique solution. When generating the 3D cubes, the proposed algorithm does *not* check for uniqueness of solutions. There are two reasons for this. First, checking for uniqueness would make the puzzle generation

process slower. Second, if each member within a set of six puzzles had a guaranteed unique solution, then they are essentially six different puzzles and can be solved individually. If there is no guarantee of a unique solution for each side, then changing the values of any of the linked edges has a very real effect on a side's four neighbors.

III.E 3D Representation – SudoKube

III.E.1 SudoKube Introduction

In order to truly test the ability of genetic algorithms in the combinatorial arena, a method of solving six linked puzzles *simultaneously* was developed. Due to the lack of availability of a six sided Sudoku puzzle, a new brainteaser had to be created in order to provide the algorithm with something on which to operate. The previous sections of this thesis outlined the generation of the 3D puzzle (SudoKube). The following pages describe the solver algorithm and detail its effectiveness.

III.E.2 Genotype

In the previous section covering the 2D representation, the genotype was a 9×9 grid. Each population member was its own copy of the puzzle on which the genetic algorithm was operating. However, in the 3D solver, each population member is a $6 \times 9 \times 9$ array – corresponding to the six sides of the generated puzzle and each of the nine rows and columns per side.

III.E.3 Operators

III.E.3.a Parent Selection

Originally, the algorithm selected a random member from the top half of the population and another member from the entire population to choose a pair of parents. As stated previously, this did not fully utilize the ability of a genetic algorithm to select the fittest members the majority of the time. Instead, the probability of selecting a population member k is given by equation 4, defined earlier and shown again below, which is a modified form of a selection probability equation that is found in **[10]**.

$$p(k) = 2 * \frac{M-k}{M*(M+1)} \tag{4}$$

III.E.3.b Crossover

The proposed algorithm initially employed uniform crossover, much like the 2D representation. However, after comparing uniform crossover to fitness based crossover, the performance of uniform crossover was not as effective. Also integrated into the proposed algorithm is the difference degree crossover method described above and as taken from **[23]**.

The final version of the proposed algorithm utilizes a *fitness based + difference degree* crossover method. Two parents are selected based on the parent selection method described above. After selection, every cell in parent 1 is compared to every corresponding cell in parent 2. The proposed algorithm tracks the total number of cells

that contain different values. For a SudoKube there are 81 * 6 (486) cells. The difference degree equation is a simple ratio of the number of cells that differ to the total number of cells.

$$D_d = \frac{N_d}{N} \tag{5}$$

In the above equation, N is the total number of cells and N_d is the total number of cells that differ between parent 1 and parent 2.

Next, a crossover *type* is selected based on total grid, column, or row fitness. The probability of selection for each type is inversely proportional to the fitness values with respect to the first parent. For example, if parent 1 has an excellent row fitness total but an abysmal column fitness total, it is likely that row crossover will be selected over column crossover. Figure 21 describes the crossover operator.

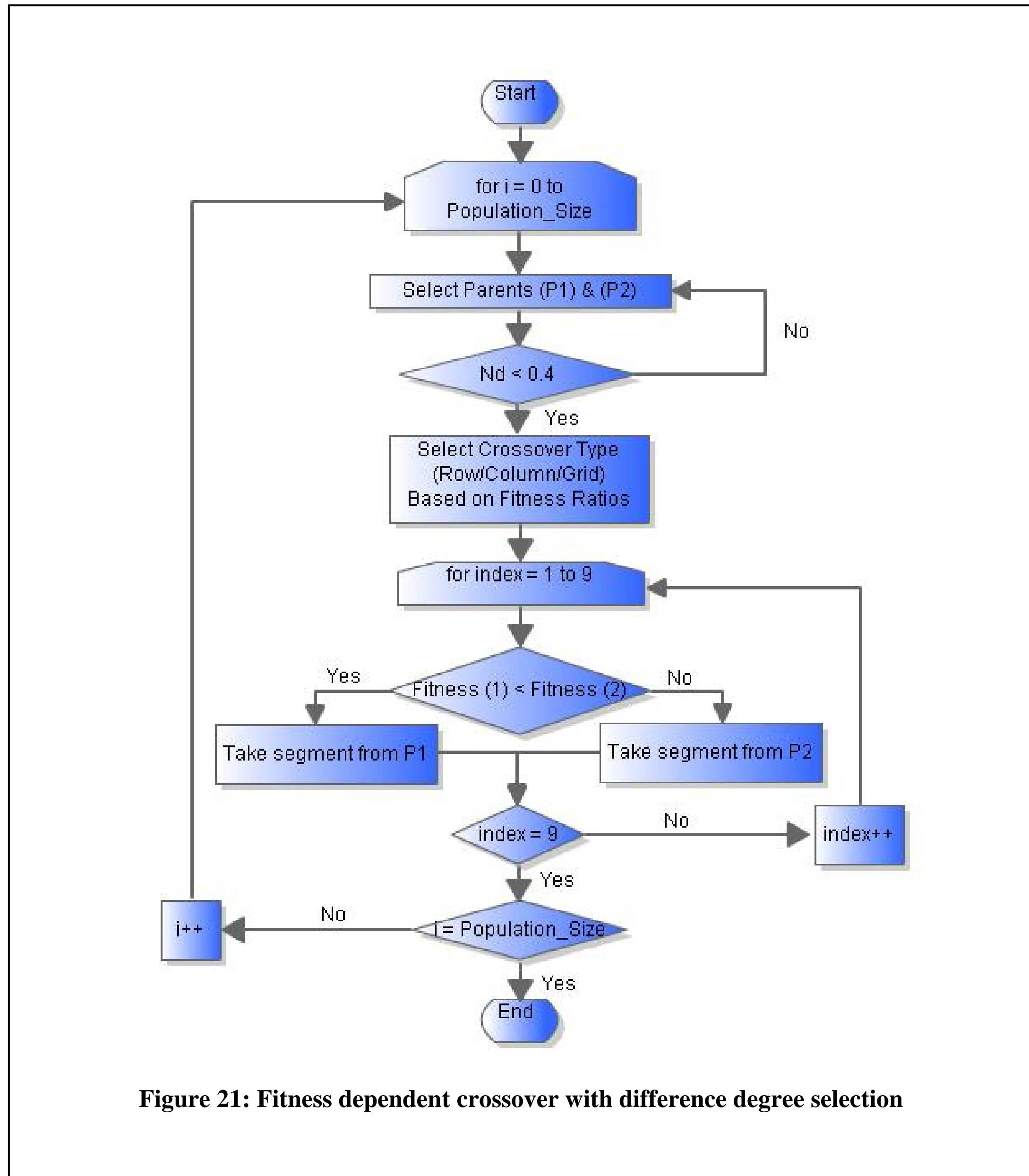


Figure 21: Fitness dependent crossover with difference degree selection

After both parents and the crossover method are selected, each segment (row, column, or mini-grid, depending on the crossover type) from each parent is compared. The segment with the better fitness is kept and passed on to the new generation. For example, if P1 has a first row with fitness 2, and P2 has a first row with fitness 9, the child would acquire the first row from P1. An example is below in table 3.

Table 3: Example of fitness-dependent crossover.

Row/Column/Grid Index	Segment Fitness Parent A	Segment Fitness Parent B	Selected Segment – Fitness Value
1	2	0	B - 0
2	4	6	A - 4
3	4	4	B - 4
4	2	6	A – 2
5	8	10	A – 8
6	12	10	B - 10
7	2	4	A – 2
8	0	2	A – 0
9	0	0	B - 0

III.E.3.c Mutation

The mutation operator for the 3D representation is very similar to the operator for the 2D representation. However, there is a key difference. The operator is applied to all six sides of the cube, not just to one puzzle. If row mutation/growth is selected, the proposed algorithm applies row mutation/growth (as described previously) to each side in succession.

III.E.3.d Fitness Calculation

Fitness calculations for the 3D representation of Sudoku are calculated in much the same manner as the 2D representation. Duplicates in each row, column and mini-grid

are added together to acquire a total side fitness. Each population member goes through this process to calculate the number of duplicates for all six of its sides, and the total overall fitness is the sum of the total side fitness values.

The difference between the fitness calculation for the 3D representation and the one for the 2D representation is that two rows and two columns have a *weighted* fitness in the 3D representation. A row or column that is an *edge* of the cube has its fitness value multiplied by ten. For example, the top row of a side (which is the bottom row of an adjacent side) with two duplicates would have a fitness value of 20. If the fitness is zero, this obviously has no effect. Weighting helps ensure that the outer edges of each Sudoku puzzle will not have any duplicates.

III.E.4 Settings

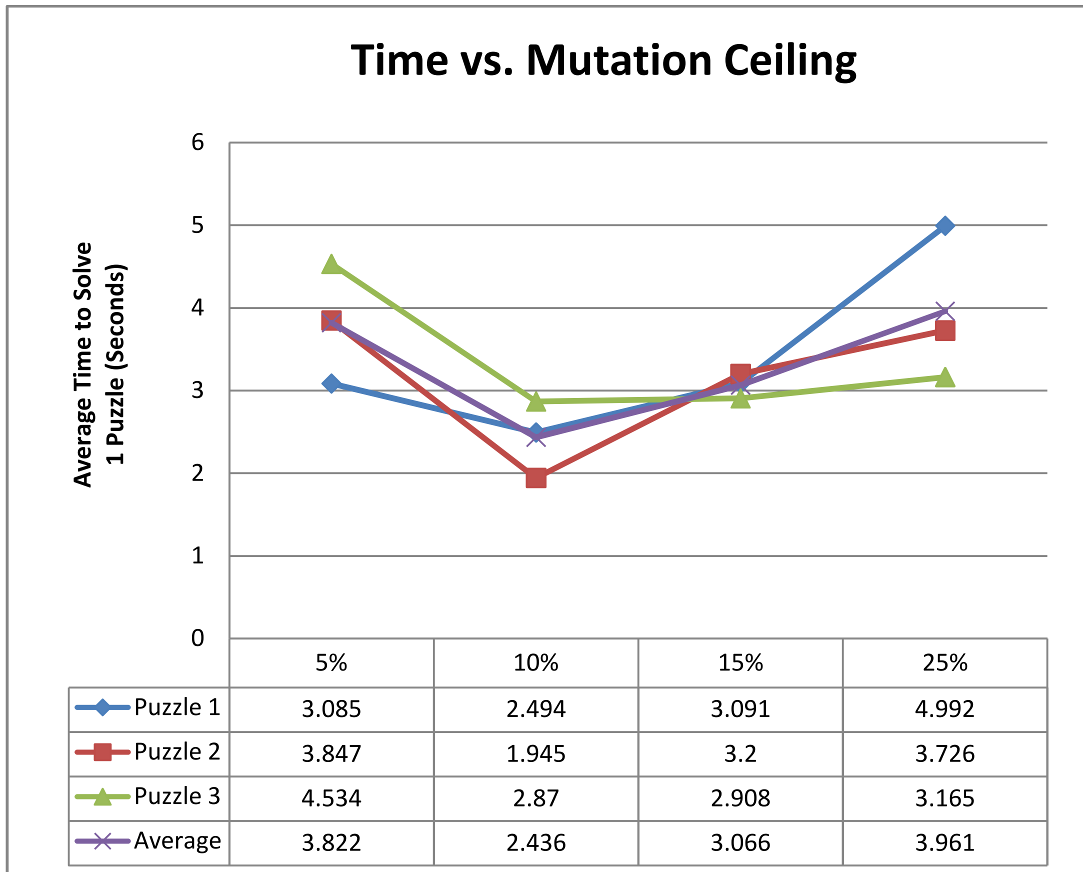
There are several significant settings that have an effect on how well the proposed algorithm performs. Some of these settings, such as 'population size' and 'maximum iterations,' are common to any genetic algorithm. Others, such as 'mutation ceiling' and 'mutation rate multiplier' are not. These settings were optimized when testing the 2D solver, and the settings were carried over and used in the 3D representation.

III.E.4.a Mutation Ceiling

The mutation ceiling is the highest value that the mutation probability can attain. Setting this number too high causes a chaotic placement of random numbers within puzzles for several iterations after the mutation probability is increased to its ceiling.

Setting this number too low greatly increases the chance that the algorithm will remain stuck in a basin once it enters one.

Figure 22 displays a segment of sample data. It shows that 10% is an value for the mutation ceiling. Using this value led to finding solutions quickly and effectively.



	5%	10%	15%	25%
Puzzle 1	3.085	2.494	3.091	4.992
Puzzle 2	3.847	1.945	3.2	3.726
Puzzle 3	4.534	2.87	2.908	3.165
Average	3.822	2.436	3.066	3.961

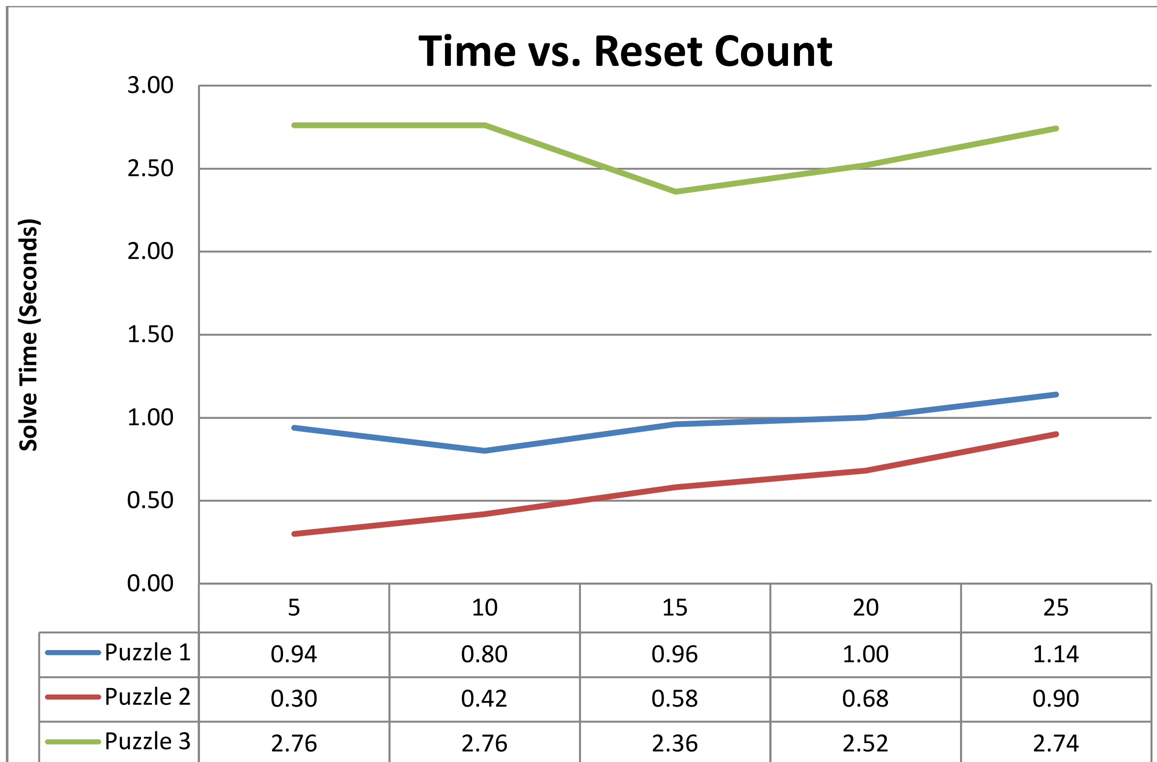
Figure 22: Time vs. mutation ceiling

III.E.4.b Reset Count

The reset count represents the number of iterations the algorithm will run before changing the mutation rate to the mutation ceiling. Like the mutation ceiling, if this number is set incorrectly, it would be detrimental to the proposed algorithm. If the reset

count is set low, it does not give the crossover and mutation operators many iterations to improve the fitness of the overall population before setting the mutation rate to a high value, which could result in the algorithm failing to find a solution for difficult puzzles. If the reset count is set high, then the algorithm may spend a disproportionate amount of time in a local minimum before the change in the mutation rate allows it to escape.

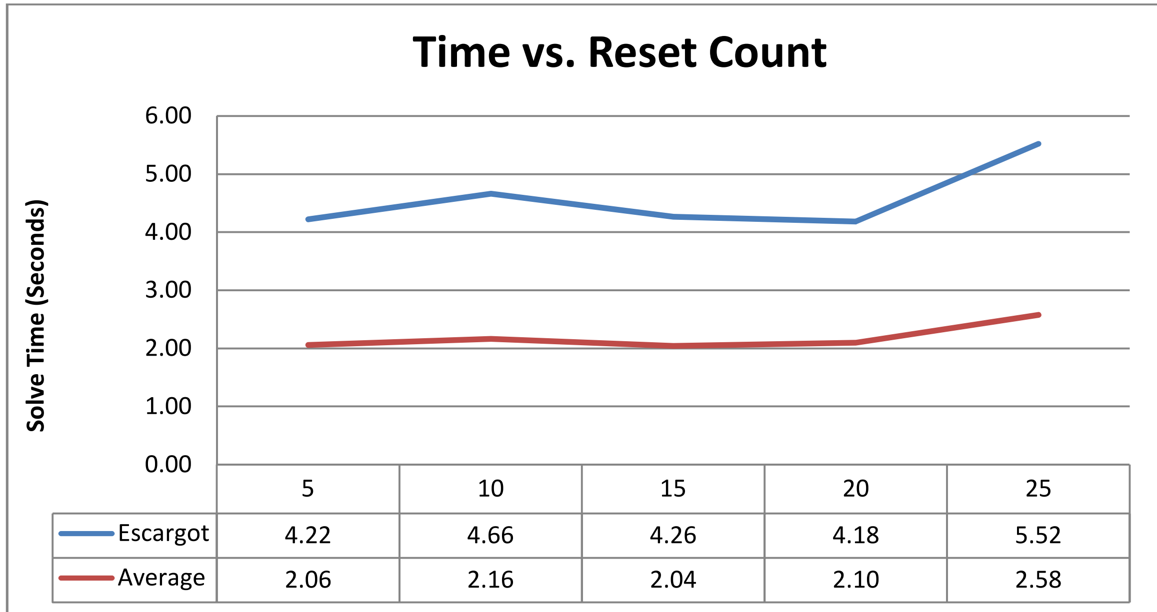
An additional check is included in the check for the reset count. If the total fitness of the best population member is equal to the total fitness of the worst population member, the algorithm is likely stuck in a basin. In this case the mutation rate would be set to the mutation ceiling. Figure 23 is a plot of solve time versus the reset count value.



	5	10	15	20	25
Puzzle 1	0.94	0.80	0.96	1.00	1.14
Puzzle 2	0.30	0.42	0.58	0.68	0.90
Puzzle 3	2.76	2.76	2.36	2.52	2.74

Figure 23: Time vs. reset count (A)

For clarity, the results for the 'Escargot' puzzle (a very difficult instance of a Sudoku puzzle) were moved to its own chart in Figure 24 along with the average of all solve times. The 'Escargot' puzzle was used to test the limits of the solver.



	5	10	15	20	25
Escargot	4.22	4.66	4.26	4.18	5.52
Average	2.06	2.16	2.04	2.10	2.58

Figure 24: Time vs. reset count (B)

The results seem to indicate that the solver benefits from a low reset count if the puzzle is not difficult. This suggests that the proposed algorithm finds local minima quickly when solving an easy puzzle, and the low reset count allows the solver to escape said minima just as quickly by increasing the mutation rate.

The reset count used in the final version of the proposed algorithm was 20. Lower reset counts often yielded slightly faster results, but they did not *always* enable the proposed algorithm to reach a solution.

III.E.4.c Mutation Rate Multiplier

The mutation rate multiplier is the percentage by which the mutation rate is multiplied. The mutation rate is reduced by this factor each iteration until it reaches the *starting* mutation rate. For example, if the mutation ceiling set the mutation rate to 25%, and the multiplier was 0.5, the sequence of mutation rates over the following few iterations would be 12.5%, 6.25%, 3.125%…etc until reaching the initial setting for the mutation rate, which is about 1%. This method of varying the mutation rate is loosely based on the temperature change concept in simulated annealing.

Figure 25 illustrates the behavior that occurs when a mutation rate multiplier of 0.90 is used. A reset count of 50 was used in this example. The solver was run on the Escargot Sudoku puzzle.

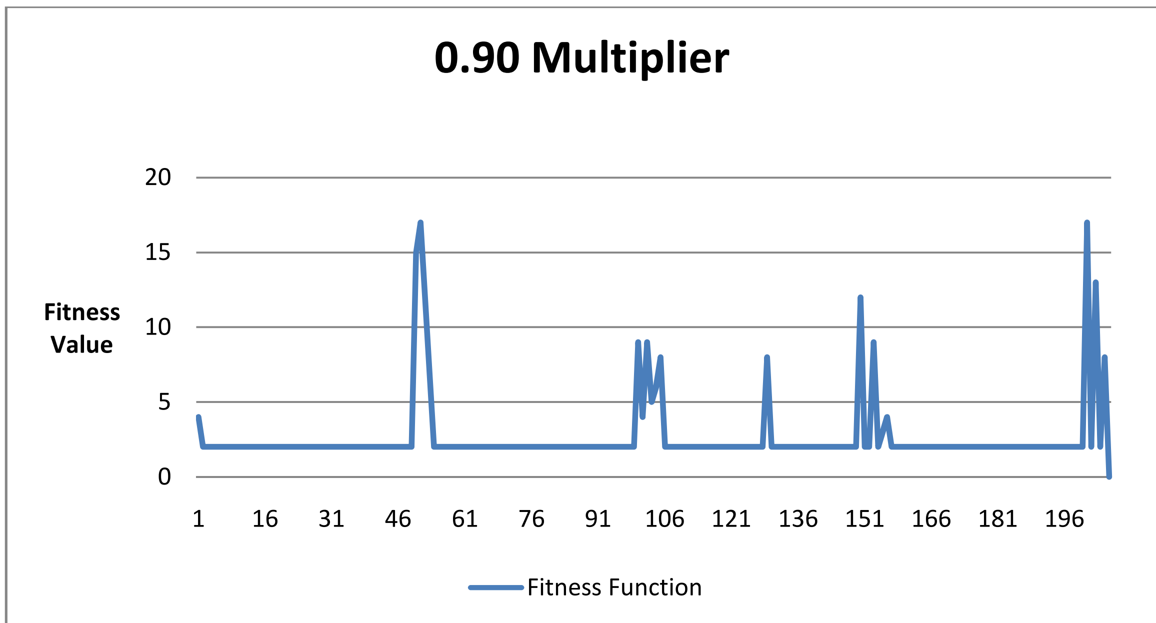


Figure 25: Fitness vs. mutation rate multiplier of 0.90

Figure 26 illustrates the behavior that occurs when a mutation rate multiplier of 0.991 is used. A reset count of 50 was also used in this example. Again, the solver was run on the Escargot Sudoku puzzle.

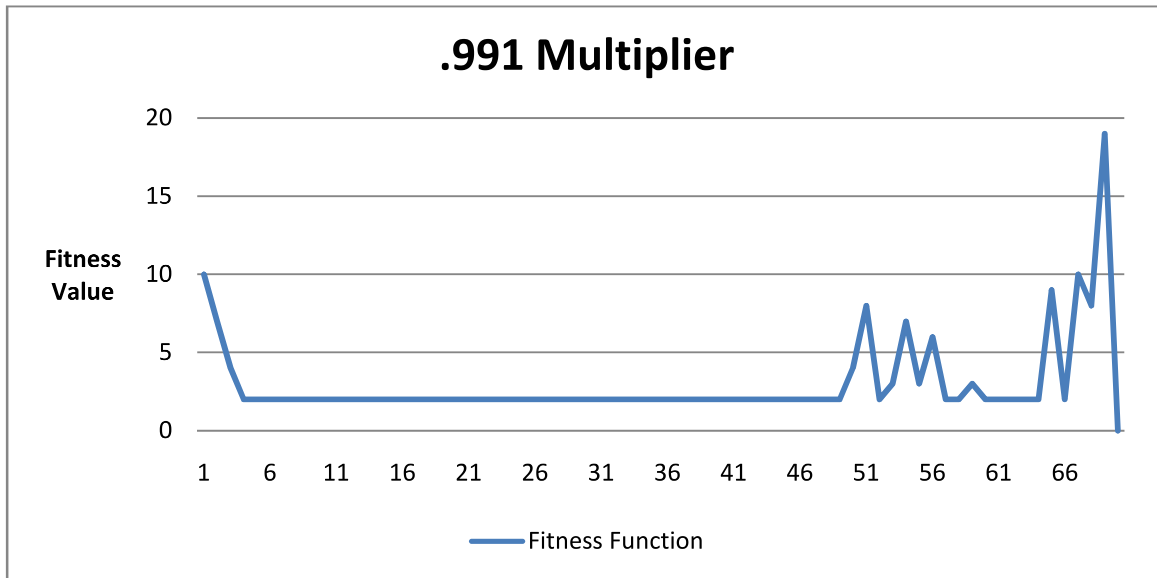


Figure 26: Fitness vs. mutation rate multiplier of 0.991

With a mutation rate multiplier of 0.991, the algorithm maintained a high mutation rate for several iterations. With a mutation rate multiplier of 0.90, the mutation rate decreased too fast to be effective. *Eventually*, the algorithm with a mutation rate multiplier of 0.90 was able to escape the local minimum, but it took the solver four more resets before it was able to find the solution. The final version of the proposed algorithm uses a multiplier of 0.99.

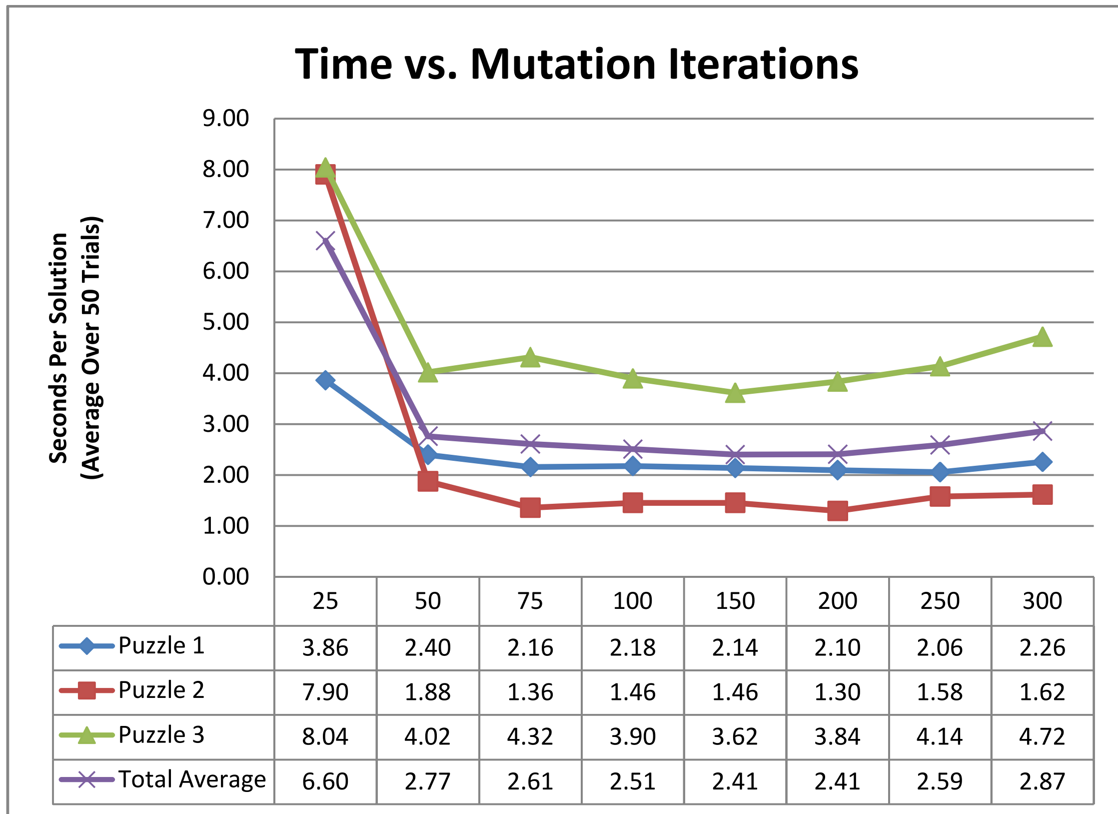
III.E.4.d Maximum Iterations

The maximum iteration setting determines how many generations the proposed algorithm will create. If this number is set too low, it is unlikely that the algorithm will

find the optimal solution to *any* Sudoku puzzle. If this value is set to an infinite number of generations, it would be virtually impossible for the algorithm *not* to find a solution to any valid Sudoku puzzle. However, the time it may take to reach that solution would be unacceptable. Also, many of the other settings in the algorithm become trivial if it is allowed to run indefinitely. Even a random search is able to find a solution if it is given enough time. For the purposes of solving the SudoKubes, 2000 iterations were more than enough to reach the optimal fitness value of zero.

III.E.4.e Mutation Iterations

The maximum iterations setting for the mutation operator determines how far the 'Natural Growth' operator is allowed to take each member of the population. Each mutation iteration represents one application of row/column/mini-grid mutation as described previously. Applying the mutation operator multiple times per population member serves a dual purpose. If the mutation rate is low, the operator eliminates many duplicates throughout the population member. If the mutation rate is high (after it is set to the mutation ceiling), it allows the algorithm to escape a local minima. The proposed algorithm uses a mutation iteration setting of 200, because it yielded the fastest solve time coupled with a 100% solve rate. A plot of solve time versus mutation iterations is shown below in figure 27.

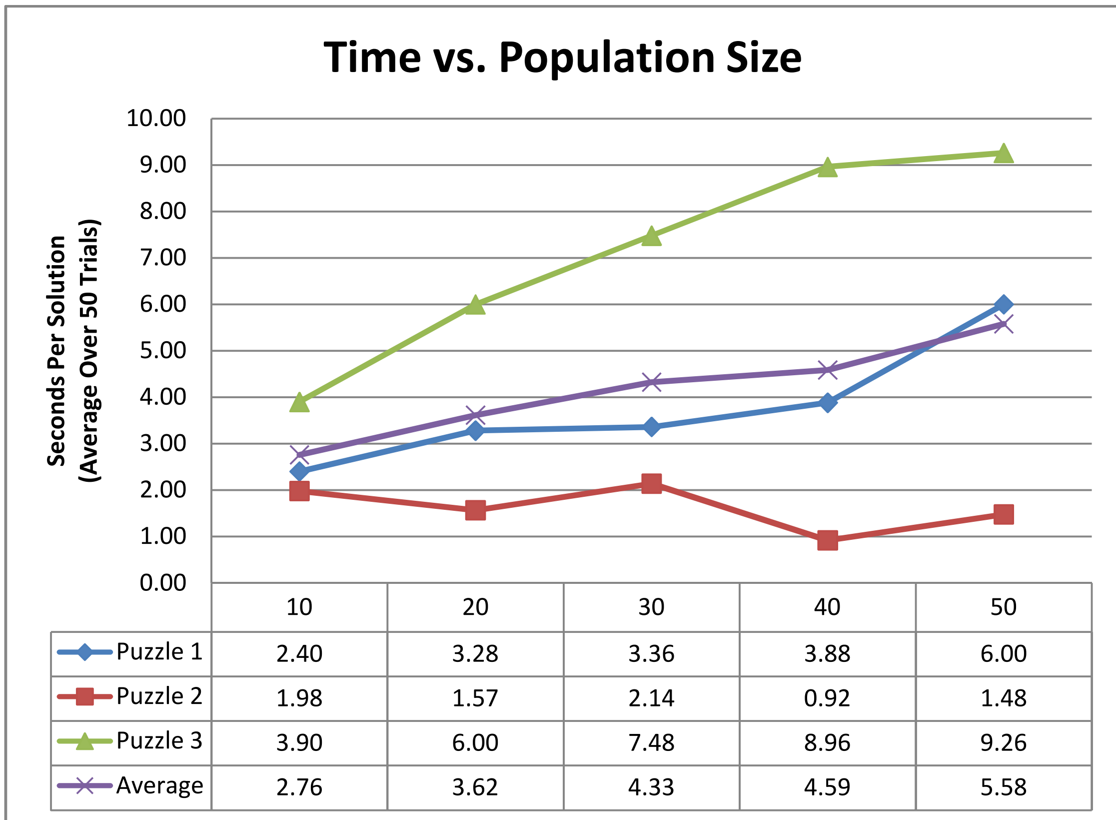


	25	50	75	100	150	200	250	300
Puzzle 1	3.86	2.40	2.16	2.18	2.14	2.10	2.06	2.26
Puzzle 2	7.90	1.88	1.36	1.46	1.46	1.30	1.58	1.62
Puzzle 3	8.04	4.02	4.32	3.90	3.62	3.84	4.14	4.72
Total Average	6.60	2.77	2.61	2.51	2.41	2.41	2.59	2.87

Figure 27: Time vs. mutation iterations

III.E.4.f Population Size

The population size is the number of solutions created by crossover during each iteration (generation). It is also the number of solutions kept at the end of each iteration. Genetic algorithms have the benefit of being able to scale their overall search according to population size. Increasing the population size allows the genetic algorithm to cover more of the search space. However, for each additional population member added to the gene pool, a significant amount of time is added per iteration. The final version of the proposed algorithm uses a population size of 20. A smaller population size did not yield a 100% solve rate. Figure 28 is a plot of solve time versus population size.

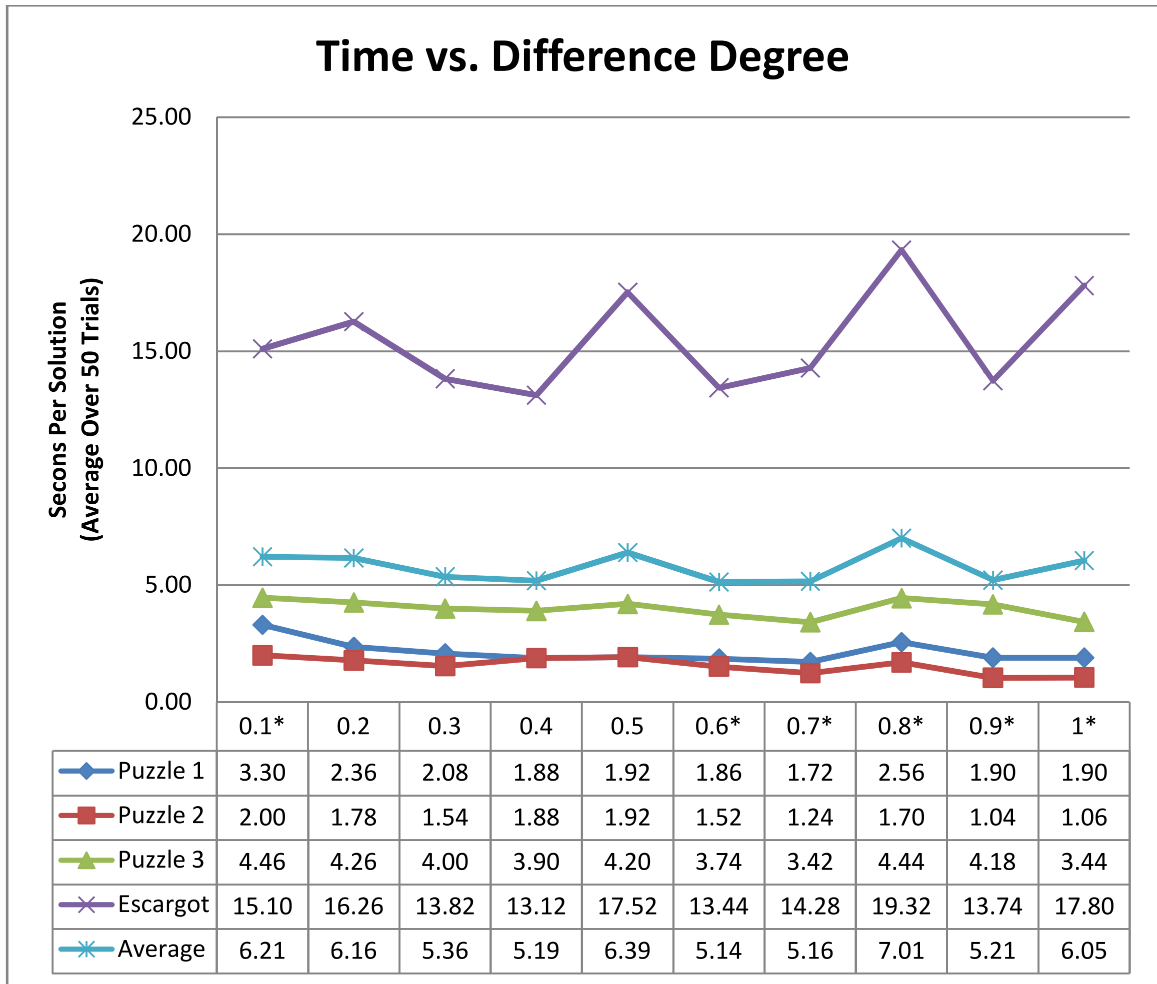


	10	20	30	40	50
Puzzle 1	2.40	3.28	3.36	3.88	6.00
Puzzle 2	1.98	1.57	2.14	0.92	1.48
Puzzle 3	3.90	6.00	7.48	8.96	9.26
Average	2.76	3.62	4.33	4.59	5.58

Figure 28: Time vs. population size

III.E.4.g Difference Degree

The difference degree represents the percentage of elements that are allowed to differ between parents selected for a crossover operation. If the difference degree value is set to 1.0, then it completely negates the difference degree operator, as it would allow mating between solutions that were 100% different. If the value is set to 0.0, then crossover would be completely useless, since the only parents allowed to mate would be exactly alike. This value is best set somewhere between 0.3 and 0.5, as shown in Figure 29.



	0.1*	0.2	0.3	0.4	0.5	0.6*	0.7*	0.8*	0.9*	1*
Puzzle 1	3.30	2.36	2.08	1.88	1.92	1.86	1.72	2.56	1.90	1.90
Puzzle 2	2.00	1.78	1.54	1.88	1.92	1.52	1.24	1.70	1.04	1.06
Puzzle 3	4.46	4.26	4.00	3.90	4.20	3.74	3.42	4.44	4.18	3.44
Escargot	15.10	16.26	13.82	13.12	17.52	13.44	14.28	19.32	13.74	17.80
Average	6.21	6.16	5.36	5.19	6.39	5.14	5.16	7.01	5.21	6.05

Figure 29: Time vs. difference degree – *Indicates settings without a 100% solve rate

Although a difference degree setting of 0.6 had the best overall average times, only four difference degree settings solved the 'Escargot' problem on every run. It is worth noting that the time difference between the best and the worst settings for the Escargot puzzle is 6.2 seconds, and the time difference for a much easier puzzle, Puzzle 2, is 0.94 seconds. This discrepancy between easy and difficult puzzles leads to the conclusion that adjusting settings to account for the possibility of solving a very difficult puzzle is the best approach.

After testing all of these different variables, it became clear that each puzzle has its own set of 'ideal' settings. For example, puzzles of easy to moderate difficulty were solved quickest when using a low value for the 'reset count' setting, but more difficult puzzles needed more iterations between 'resets'. Throughout testing, it was evident that each setting was not entirely independent of every other setting. An example of two intertwined settings would be 'mutation ceiling' and 'mutation multiplier'. With a high mutation ceiling, it is desirable to have a large reduction in the mutation rate per iteration. With a low ceiling, this is not the case. For the final proposed algorithm, the values for each setting that yielded a solve rate of 100% in a reasonable amount of time were used.

CHAPTER IV

RESEARCH FINDINGS

Chapter III of this thesis outlined the standard 2D Sudoku solver, the 3D SudoKube solver, and puzzle generation for the 3D SudoKube. This chapter will describe the performance of both the 2D and the 3D solvers.

The 2D solver within the genetic algorithm being tested was applied to puzzles taken from the same periodicals as **[5]** in order to have analogous results. The genetic algorithm described in this thesis had an overall performance that was better than the comparable genetic algorithms that were described in similar research.

The first and most important factor in measuring the effectiveness of a Sudoku solver is its rate of success. The speed of a solver is only significant if it can find solutions on virtually every run. Below, Table 4 shows the results of the 2D solver described in this thesis as compared to other research. The algorithm attempted to solve each puzzle 30 times. In order to match the testing done in **[5]**, 3 easy puzzles, one medium puzzle, and one hard puzzle were selected for testing.

Table 4: Solutions found out of 30 attempts – Puzzles taken from [30]

Puzzle	Proposed Algorithm (2,000 iterations)	Mantere/Koljonen (Unlimited Iterations) [5]	Mantere/Koljonen (5,000 Iterations) [5]	Hamming Space Crossovers [4]	Swap Space Crossovers [4]	Hill Climbers [4]
Easy 1	30	30	29	5	28	30
Easy 2	30	30	30	8	21	30
Easy 3	30	30	30	14	30	30
Medium	30	30	10	0	0	0
Hard	30	30	2	0	15	0
Total	150	150	101	27	94	90

Regardless of the challenge rating, the proposed algorithm solved every given puzzle with no difficulties. The large difference between maximum iterations allowed between this genetic algorithm and the one described in **[5]** can be attributed to the implementation of the mutation and natural growth operator in the proposed algorithm. With the natural growth operator in place, there is no need to run the algorithm for so many generations, because each generation is greatly improved by applying natural growth.

The algorithm designed by Mantere and Koljonen also solved each puzzle 30 times out of 30 attempts, but only when allowed to run indefinitely. When limited to 5,000 iterations, it solved 10 out of 30 on the medium difficulty and 2 out of 30 on the hard difficulty. The puzzles were taken from **[30]**, which has a free downloadable Sudoku game.

The following table compares the results of this solver and solvers from **[4]** and **[5]**. The puzzles rated 1 Star through 5 Star were taken from **[2]**, and the puzzles rated 'Easy', 'Challenging', 'Difficult', and 'Super Difficult' were taken from **[31]**. While it is

unlikely that the exact same puzzles were used for testing, the testing method used was the same. The solver was applied to three puzzles from each difficulty level 100 times each. The results in Table 5 show the mean number of times the solver worked for each difficulty.

Table 5: Solutions found out of 100 attempts – Puzzles taken from [31] & [2]

	1 Star	2 Star	3 Star	4 Star	5 Star	Easy	Challenging	Difficult	Super Difficult
Proposed GA	100	100	100	100	100	100	100	100	100
Old GA*	100	69	46	26	23	100	30	4	6
GA w/Unlimited Iterations*	100	100	100	100	100	100	100	100	100
GA w/5,000 Iterations*	100	100	96	63	47	100	60	10	8
* Indicates results from [5]									

Again, the genetic algorithm designed for the research in this thesis performed very well. When allowed unlimited iterations, the algorithm from **[5]** also solved the puzzles 100% of the time. However, when limited to only 5,000 iterations, the solve rate dropped to approximately 22%. According to **[5]**, the largest number of iterations required to solve a puzzle when allowed to run indefinitely was approximately 203,300 – more than 40 times the 5,000 limit.

The difference in solver effectiveness indicates that the genetic algorithm described in this thesis is a better solver in terms of solving percentage. It also compared favorably when measuring both solution time and required solution generations.

In the tables below, (Table 6 & Table 7) the mean solve times for each individual puzzle is listed along with the mean solve iterations. Included in the tables are the maximum and minimum numbers of iterations required per solution. Each puzzle was

solved exactly 100 times. Solve time was calculated by taking the difference between the DateTime.Now value before the solver was run and the DateTime.Now value after the solver had finished working. The DateTime.Now value is accessible in C# when 'using system' is included in the 'using' statements.

In the tables below, 'Solve Time' is the mean solve time over 100 solve attempts. 'Solve Iterations' is the mean number of generations required to solve the puzzle over 100 attempts. 'Max' and 'Min' represent the maximum and minimum number of generations required to solve the puzzle out of the 100 attempts. 'Fit Calc Per Puzzle' is the mean number of fitness calculations performed per solve attempt.

Table 6: Results on puzzles from [31]

Puzzle Type	Solve Time	Solve Iterations	Max	Min	Fit Calc Per Puzzle
5 Star - A	2.87	73.34	300.00	2.00	97786.67
5 Star - B	3.55	88.68	376.00	1.00	118240.00
5 Star - C	4.61	108.64	379.00	6.00	144853.33
4 Star - A	2.89	72.61	301.00	2.00	96813.33
4 Star - B	12.49	273.26	1620.00	21.00	364346.67
4 Star - C	3.13	66.46	279.00	2.00	88613.33
3 Star - A	1.55	32.13	128.00	1.00	42840.00
3 Star - B	2.75	68.60	225.00	2.00	91466.67
3 Star - C	3.18	69.66	201.00	1.00	92880.00
2 Star - A	1.91	39.60	128.00	1.00	52800.00
2 Star - B	1.60	33.49	121.00	1.00	44653.33
2 Star - C	2.01	42.63	131.00	1.00	56840.00
1 Star - A	0.56	9.11	76.00	1.00	12146.67
1 Star - B	0.27	2.14	37.00	1.00	2853.33
1 Star - C	1.78	38.67	137.00	1.00	51560.00
Helsingin Sanomat	3.01	67.93	295.93	2.93	90579.56

For the Helsingin Sanomat online newspaper publication (one of the main sources for Sudoku puzzles in **[5]**), the overall mean solve time for this genetic algorithm was 3.01 seconds, and the mean for the solve generations was 67.93. The approximate mean number of fitness calculations over 100 solve attempts was 90,580.

Table 7: Results on puzzles from [2]

Puzzle Type	Solve Time	Solve Iterations	Max	Min	Fit Calc Per Puzzle
V. Hard - A	3.30	80.89	296.00	2.00	107853.33
V. Hard - B	4.84	107.14	370.00	2.00	142853.33
V. Hard - C	3.69	80.87	256.00	2.00	107826.67
Hard - A	2.92	87.95	320.00	4.00	117266.67
Hard - B	2.24	66.70	334.00	2.00	88933.33
Hard - C	6.70	201.29	943.00	4.00	268386.67
Normal - A	0.91	16.74	79.00	1.00	22320.00
Normal - B	0.46	6.13	77.00	1.00	8173.33
Normal - C	3.17	70.66	215.00	2.00	94213.33
Easy - A	0.17	1.13	2.00	1.00	1506.67
Easy - B	0.16	1.12	2.00	1.00	1493.33
Easy - C	0.17	1.14	2.00	1.00	1520.00
Aamulehti	2.39	60.15	241.33	1.92	80195.56
Total Averages	2.70	64.04	268.63	2.43	85387.56

For the Aamulehti online publication (another main source for Sudoku puzzles in **[5]**), the overall mean solve time for this genetic algorithm was 2.70 seconds, and the mean for the solve generations was 64.04. The approximate mean number of fitness calculations over 100 solve attempts was 80,196.

Overall, the algorithm in **[5]** solved all of the puzzles with an average of 4.11 seconds per Sudoku grid. It was run on a 3GHz Pentium 4 processor, and programmed in Java. The algorithm described in this thesis solved all of the puzzles with an overall

average of 2.7 seconds per Sudoku grid, but both the processor and programming language differ from **[5]**. The processor for the machine that tested the proposed algorithm was a Pentium Core 2 – 2.0 GHz, and the programming language was C# in Visual Studio 2005. The system also had 2GB of RAM installed.

As a result of using different systems on which to test the algorithms, a better benchmark for comparable effectiveness is the mean number of generations required to solve these puzzles. In **[5]**, the overall mean was approximately 9745 generations per solution, which corresponds to approximately 195,000 fitness calculations per solution. For the testing on the algorithm described in this thesis, the overall mean was approximately 64 generations, which corresponds to about 85,388 fitness calculations per solution – less than half the calculations as the algorithm in **[5]**.

The following pages illustrate the solver's effectiveness in rapidly reducing the number of duplicates within a potential solution (Figures 30, 31, & 32). It is evident that more difficult puzzles take many more iterations to solve. The puzzles for the following charts were taken from **[30]**.

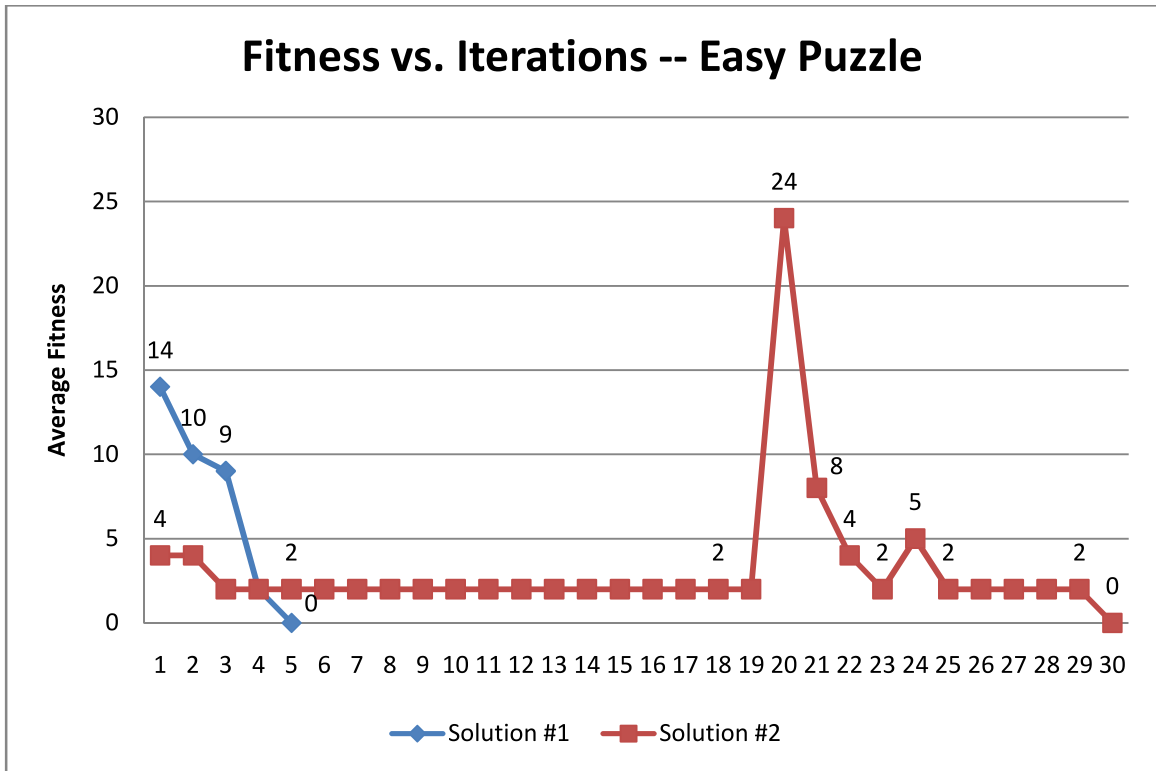


Figure 30: Fitness vs. iterations (easy)

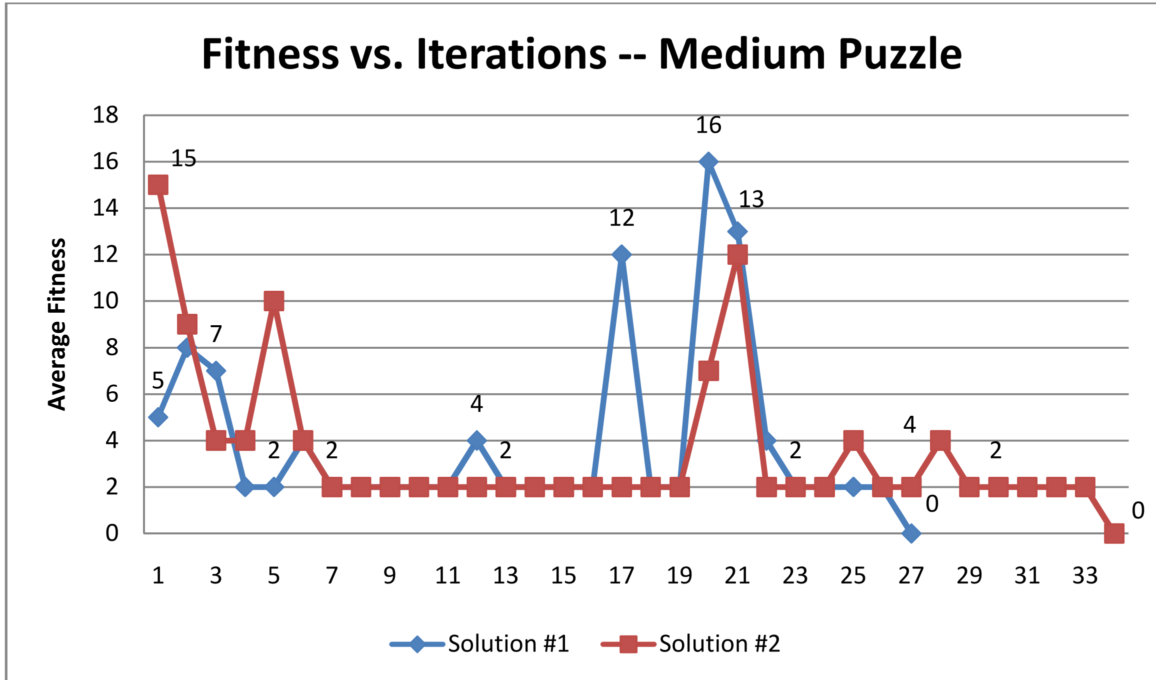


Figure 31: Fitness vs. iterations (medium)

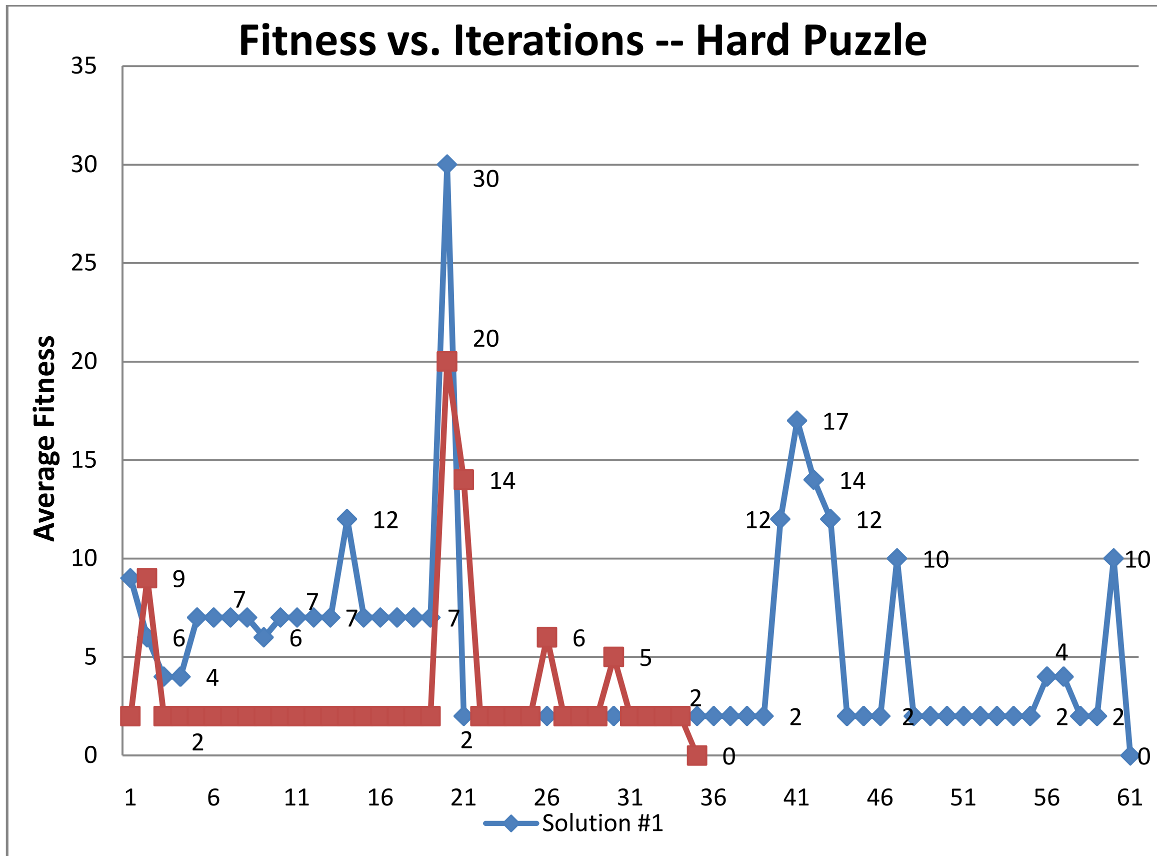


Figure 32: Fitness vs. iterations (hard)

The solver was applied to the same puzzle twice, and the two runs are represented in the plots by the names 'Solution #1' and 'Solution #2'. It is obvious that the algorithm solved the puzzle in a different way on each run. The first run on the easy puzzle took 5 iterations to solve, but the second run took only 30. For the medium puzzle, the first run took 27 iterations, and the second run took 32 iterations. However, the path that the algorithm took to get to the solution was quite different between the two runs. The hard puzzle had the largest difference between the two runs. The first run took 35 iterations, and the second run took 61 iterations. The spike in fitness at iteration multiples of 20 corresponds to a 'reset count' of 20, where the mutation rate is greatly increased for a time.

The 2D solver was quite successful, and the 3D solver was modeled on the 2D type. Using the method described in the previous section, 50 puzzles of each difficulty (Easy, Medium, Hard, and Very Hard) were generated. Each one was solved 100 times for a total of 20,000 SudoKubes solved. Each SudoKube (Figure 33) contains six valid Sudoku grids, which gives a total of 80,000 Sudoku grids solved.

Figure 33: Screen shot of a SudoKube

In order to determine whether or not solving six puzzles simultaneously is an effective approach to this type of combinatorial problem, the tests were run on both the 3D solver and also the 2D solver. The 2D solver was run sequentially on each individual side. The 2D sequential solver was not required to have the edges of the cube match. Below is a table that outlines the overall results of the testing.

Table 8: Comparison of sequential solver vs. 3-D solver (based on averages of solving 100 puzzles)

Puzzle Representation	Average Time to Solve 1 Cube	Shortest Time Average	Longest Time Average	Average Variance
2D Easy	0.2962	0.2952	0.3076	0.0017
3D Easy	0.2515	0.2363	0.3135	0.0008
2D Med	1.5247	0.7260	3.7757	0.6489
3D Med	1.3308	0.6481	4.3675	1.6800
2D Hard	1.7234	0.6481	4.4117	1.0390
3D Hard	0.9687	0.4273	3.5490	0.6970
2D VH	2.0540	0.8804	4.9988	1.2069
3D VH	2.0385	0.6291	6.9425	3.2402
Based on 50 puzzles per difficulty for both 2D and 3D representations				

Population variance (σ^2) was calculated with the following formula with N representing the number of times the algorithm was run and X representing the solve time:

$$\sigma^2 = \frac{\sum(X^2) - \sum(X)^2/N}{N} \tag{6}$$

For every difficulty setting, the average time to solve one six-sided puzzle was shorter for the 3D representation than it was for the 2D representation. However, for three out of the four difficulty levels (Easy, Medium, and Very Hard), the average 'longest' time was higher for the 3D representation. In the case of the Very Hard puzzle, the difference between the 'longest' averages was approximately 2 seconds, in favor of the 2D solver. Also in the case of the Very Hard difficulty level, the 3D solver had a *smaller* 'shorter' time average than the 2D sequential. The full range of the solve times as well as the average solve times for each difficulty can be seen below (Figure 34). From top to bottom, each column displays the maximum solve time, the average solve time, and the minimum solve time.

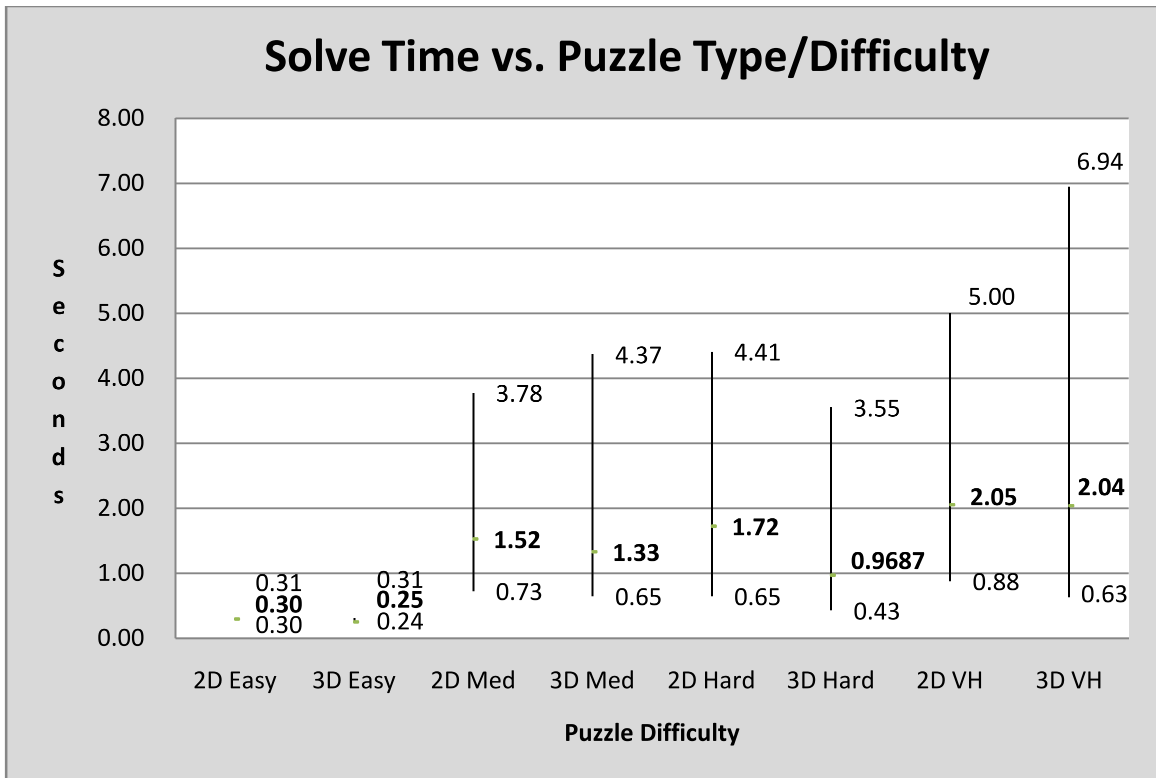


Figure 34: Solve time vs puzzle type/difficulty

Both versions of the solver solved six Sudoku puzzles quickly, but the use of the 2D solver did not guarantee that the sides of the SudoKube would match. The 3D solver worked all six sides *simultaneously,* which allowed it to find equivalent sides as it worked towards the global optimum.

During the testing, an attempt was made to use the 2D sequential solver to find a valid solution for a whole SudoKube. However, since most of the sides had more than one solution when solved alone, this option was not feasible. The 2D solver would solve the first side with no problem, but subsequent sides often had the problem of containing conflicting constraints. The first sides solved created contradictions when their sides were copied to adjacent puzzles.

CHAPTER V

CONCLUSION

After running tests on puzzles taken from many different sources, it is reasonable to state that the solver is effective. As an extension of the 2D solver, the 3D solver was also effective.

V.A 2D Representation – Success

The Sudoku solver was a success. It performed better than the genetic algorithm based Sudoku solvers presented in similar research by a fair margin. The solver provided an excellent foundation for the 3D representation.

V.B 3D Representation – Success

The solver was a success. It performed better than the 2D solver operating sequentially, and the 2D solver itself worked better than its predecessors.

When solving a problem in which constraints are linked to the solutions of other problems, it can be beneficial to solve every problem simultaneously. The case of 3D Sudoku illustrates this. If a different representation was developed for network routing problems, scheduling problems, or the traveling salesman problem, this method of solving multiple combinatorial problems could be applied easily.

Also, the developed program allows a user to play a challenging new Sudoku-type game. In this respect, the algorithm is also a success (Figure 35).

V.C Future Work

Future work would include using a genetic algorithm to find *every* possible solution to a given Sudoku grid. The goal with such an exercise would be to maintain population diversity while still finding optimum solutions.

Due to the large number of different settings within the proposed algorithm, using another evolutionary algorithm in order to optimize the settings has the potential to be worthwhile.

Figure 35: Solved SudoKube

REFERENCES

[1] Wei-Meng Lee, *Programming Sudoku*. New York, NY: Springer-Verlag New York, Inc., pp. 47-170, 2006.

[2] Aamuulehti.fi, "Sudoku", http://www.aamulehti.fi/sudoku/

[3] T. K. Moon and J. H. Gunther, "Multiple constraint satisfaction by belief propagation: an example using Sudoku," Proceedings of IEEE Mountain Workshop on Adaptive and Learning Systems, pp 122-126, 2006.

[4] A. Moraglio, J. Togelius, and S. Lucas, "Product geometric crossover for the Sudoku puzzle," *Proceedings of IEEE Congress on Evolutionary Computation,* pp. 470-476, 2006.

[5] T. Mantere and J. Koljonen, "Solving, rating and generating Sudoku puzzles with GA," *Proceedings of IEEE Congress on Evolutionary Computation*, pp. 1382-1389, 2007.

[6] M. F. Tasgetiren, P. N. Suganthan, Q. –K. Pan, and Y. –C. Liaing, "A genetic algorithm for the generalized traveling salesman problem," *Proceedings of IEEE Congress on Evolutionary Computations,* pp. 2382-2389, 2007.

[7] R. Takahashi, “Solving the traveling salesman problem through genetic algorithms with changing crossover operators,” *Proceedings of the Fourth International Conference on Machine Learning and Applications*, pp. 6-12, 2005.

[8] S. A. Mulder, “Computational intelligence and the traveling salesman,” Ph.D. dissertation, University of Missouri-Rolla, Rolla, MO, USA, 2004.

[9] A. Homaifar, J. Turner, and S. Ali, “The N-Queens problem and genetic algorithms,” Proceedings of Southeastcon ’92, pp. 262-267, 1992.

[10] C. R. Reeves, “A genetic algorithm for flowshop sequencing,” *Computers Ops Res.,* vol. 22, pp. 5-13, 1995.

[11] C. R. Reeves and T. Yamada, “Genetic algorithms, path relinking, and the flowshop sequencing problem,” *Evolutionary Computation,* vol. 6, pp. 45-60, 1998.

[12] Q. Wang, K. L. Yung, and W. H. Ip, “A pattern-based evolving mechanism for genetic algorithm to solve combinatorial optimization problems,” *Proceedings of IEEE International Workshop on Soft Computing in Industrial Applications,* pp. 97-101, June 2003.

[13] Wikipedia.org, “NP-Complete”, http://en.wikipedia.org/wiki/Np_complete

[14] M. W. S. Land, “Evolutionary algorithms with local search for combinatorial optimization,” Ph.D. dissertation, University of California - San Diego, San Diego, CA, USA, 1998.

[15] Ivica Martinjak, Marin Golub, “Comparison of Heuristic Algorithms for the N-Queen Problem,” University of Zagreb, Faculty of Electrical Engineering and Computing.

[16] Wikipedia.org, “Genetic algorithm”, http://en.wikipedia.org/wiki/Genetic_algorithm

[17] Y. Yusong, “Genetic-combinatorial algorithm of 0-1 programming,” Proceedings of the Fourth International Conference on Parallel and Distributed Computing, Applications and Technologies, pp. 698-701, 2003.

[18] C. –M. Lin and M. Gen, “Multi-criteria human resource allocation for solving multistage combinatorial optimization problems using multiobjective hybrid genetic algorithm,” *Expert Systems with Applications,* vol. 34, pp. 2480-2490, 2008.

[19] L. Aaronson, “Sudoku science: a popular puzzle helps researchers dig into deep math,” *IEEE Spectrum*, pp. 16-17, Feb 2006.

[20] J. –L. Lin, "An analysis of genetic algorithm behavior for combinatorial optimization problems," Ph.D. dissertation, University of Oklahoma, Norman, OK, USA, 1993.

[21] A. Tuson and P. Ross, "Adapting operator settings in genetic algorithms," *Evolutionary Computation,* vol. 6, pp. 161-184, 1998.

[22] ScanRaid.com, Sudoku Solver, http://www.scanraid.com/sudoku.htm

[23] R. –L. Wang, S. Fukuta, J. –H. Wang, and K. Okazaki, " A genetic algorithm with conditional crossover and mutation operators and its application to combinatorial optimization problems," *IEICE Trans. Fundamentals,* vol. E90-A, pp. 287-294, 2007.

[24] Wikipedia.org, "Sudoku", http://en.wikipedia.org/wiki/Sudoku

[25] Minimum intrusion grid, N-Queen picture, http://mig-1.imada.sdu.dk/MiG/Mig/user_tutorial/images/nqueen.png

[26] Mathworks.com, "Genetic algorithm and direct search toolbox", http://www.mathworks.com/access/helpdesk/help/toolbox/gads/dejong5fcn.gif

[27] J. –W. Dang, Y. –P. Wang, and S. –X. Zhao, "Study on a novel genetic algorithm for the combinatorial optimization problem," *Proceedings of International Conference on Control, Automation and Systems*, pp. 2538-2541, 2007.

[28] B. Felgenhauer and F. Jarvis, "Enumerating possible Sudoku puzzles," http://www.afjarvis.staff.shef.ac.uk/maths/felgenhauer_jarvis_sudoku1.pdf, June 2005.

[29] A. M. Herzberg and M. R. Murty, "Sudoku squares and chromatic polynomials," *Notices of the AMS,* vol. 54, pp. 708-717, 2007.

[30] WayneGouldPuzzles.com, "Download", http://www.waynegouldpuzzles.com/sudoku/download

[31] HS.fi, "Sudoku", http://www2.hs.fi/extrat/sudoku/

VITA

David Isaac Waters

Candidate for the Degree of

Master of Science

Thesis: SUDOKUBE – USING GENETIC ALGORITHMS TO SIMULTANEOUSLY SOLVE MULTIPLE COMBINATORIAL PROBLEMS

Major Field: Electrical Engineering

Biographical:

Personal Data:

David goes by his middle name, 'Isaac'.
Isaac is from St. Louis, MO.
He attended Oklahoma Christian University in Edmond, OK from the fall of 2000 until the spring of 2002, when he transferred to Oklahoma State University.

Education:
Completed the requirements for the Bachelor of Science in Electrical Engineering at Oklahoma State University, Stillwater, Oklahoma in May, 2005.
Completed the requirements for the Master of Science in Electrical Engineering at Oklahoma State University, Stillwater, Oklahoma in May, 2008.

Experience:

Isaac worked for Interstates Engineering from 2005 to 2006 before returning to Oklahoma State to work towards his masters degree.
Isaac has completed two summer internships with The Benham Companies in their St. Louis office.
He has completed another 2 year internship with The Benham Companies in their Oklahoma City office.

Name: David Isaac Waters | Date of Degree: May, 2008

Institution: Oklahoma State University | Location: Stillwater, Oklahoma

Title of Study: SUDOKUBE – USING GENETIC ALGORITHMS TO SIMULTANEOUSLY SOLVE MULTIPLE COMBINATORIAL PROBLEMS

Pages in Study: 83 | Candidate for the Degree of Master of Science

Major Field: Electrical Engineering

Scope and Method of Study:

This thesis proposes a novel genetic algorithm to solve Sudoku puzzles. It also proposes a new algorithm for generating and solving six linked Sudoku puzzles. The six puzzles form a box, which is called a SudoKube.

Findings and Conclusions:

Sudoku is a difficult and complex combinatorial problem. Several genetic algorithms have been developed to solve Sudoku puzzles, but none have been tested to see if they are capable of solving a *set* of six Sudoku puzzles in the form of a cube. This thesis details the development of a standard Sudoku solver that outperforms its genetic algorithm predecessors, and it goes on to apply the same solver concepts to a 3D puzzle of the author's creation. This thesis also demonstrates that an algorithm meant to solve a set of six puzzles can outperform a standard solver run six times in succession to solve the same set.

ADVISER'S APPROVAL: Dr. Gary Yen