CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Brock University Digital Repository

The Salmon Algorithm - A New Population Based Search Metaheuristic

John Orth

Computer Science

Submitted in partial fulfillment
of the requirements for the degree of

Masters of Science

Faculty of Computer Science, Brock University
St. Catharines, Ontario

© December, 2011

Abstract

This thesis introduces the Salmon Algorithm, a search meta-heuristic which can be used for a variety of combinatorial optimization problems. This algorithm is loosely based on the path finding behaviour of salmon swimming upstream to spawn. There are a number of tunable parameters in the algorithm, so experiments were conducted to find the optimum parameter settings for different search spaces.

The algorithm was tested on one instance of the Traveling Salesman Problem and found to have superior performance to an Ant Colony Algorithm and a Genetic Algorithm. It was then tested on three coding theory problems - optimal edit codes, optimal Hamming distance codes, and optimal covering codes. The algorithm produced improvements on the best known values for five of six of the test cases using edit codes. It matched the best known results on four out of seven of the Hamming codes as well as three out of three of the covering codes. The results suggest the Salmon Algorithm is competitive with established guided random search techniques, and may be superior in some search spaces.

Acknowledgements

I would first like to thank my supervisor Dr. Sheridan Houghten for her guidance, insight, patience and good humour. I would even like to thank her for her gentle prodding, which I needed on occasion.

Since the experiments in this thesis involved thousands of hours of CPU time, I would like to thank Cale Fairchild for his help setting up the cluster runs and his advice on programming problems. Without him, this thesis would have taken much longer than it did.

To the members of my committee Dr. Brian Ross and Dr. Ke Qiu, I express my gratitude for taking the time to read this lengthy composition and offer your suggestions. To the external examiner Dr. Steven Corns, thank you for traveling all the way from Missouri in February. I am sure you would rather have been going south than north. I would also like to thank my friends Darren Peters, Steve Bergen, Allen Poapst and the other Master's students for accepting me as an equal and not treating me like some strange old guy that wandered in off the street. Finally, thanks to Dr. Dan Ashlock and Joseph Brown for your previous collaborations and for making the trip from Guelph to watch my defence.

All of these people have made a pleasurable experience out of something that could easily have become a chore.

J.W.O

Contents

1 Introduction **1**
1.1 Overview . . . 1
1.2 Problem Statement . . . 2
1.3 Organization . . . 2

2 Overview of Search Techniques **4**
2.1 Evolutionary Strategies . . . 5
2.2 Genetic Algorithms . . . 6
2.3 Artificial Ant Algorithms . . . 8
2.4 Simulated Annealing . . . 11
2.5 Tabu Search . . . 12

3 Salmon algorithm **13**
3.1 Salmon Algorithm Parameter Values for TSP . . . 15
3.2 Experimental Results for the TSP . . . 16

4 Error Correcting Codes **18**
4.1 Hamming Space . . . 23

5 Edit Codes **25**
5.1 Dynamic Programming Algorithm for Edit Distance . . . 27

6 The Optimal Code Problem **31**
6.1 Salmon Algorithm for Finding Maximum Cliques (Optimal Codes) . . . 33
6.1.1 Memory Optimization . . . 35
6.2 Conway's Lexicode Algorithm . . . 36
6.3 Sphere Packing Bound . . . 36

7 Covering Codes 38
7.1 Football Pool Problem . . . 39
7.1.1 Salmon Algorithm for the Football Pool Problem . . . 39

8 Literature Review 42
8.1 Edit Codes . . . 42
8.2 Covering Codes . . . 43
8.3 Hamming Codes . . . 44

9 Experimental Results 46
9.1 Optimal Edit Codes . . . 46
9.1.1 Optimum Initial Water Level Multiplier (IM) . . . 48
9.1.2 Optimum Population Size . . . 51
9.1.3 Tuning the α, ϕ and σ Values . . . 51
9.1.4 Optimum Number of Elite . . . 53
9.1.5 Optimum Parameter Values Summary . . . 53
9.2 Optimal Edit Code Results . . . 54
9.2.1 GC Content . . . 59
9.3 Optimal Hamming Code Results . . . 61
9.4 Covering Code Results . . . 62

10 Conclusion 68
10.1 Future Work . . . 69

Bibliography 70

List of Tables

3.1 Salmon Algorithm Optimum Values for TSP 16
3.2 TSP Results 17

4.1 Hamming (7,4) Code 19

6.1 Compatibility Matrix 32

7.1 Limits on Covering Codes with d = 1 and q = 3 39

9.1 Average clique size vs α for various IM 50
9.2 Best clique size vs α for various IM 50
9.3 α vs Population for (7,3) edit codes 51
9.4 Elite vs. No Elite 53
9.5 Improvements to Edit Codes 59
9.6 Fixed GC Content 60
9.7 Salmon Algorithm Binary Hamming Code Results 61
9.8 Salmon Algorithm Ternary Hamming Code Results 61

List of Figures

2.1 Search Hierarchy [42] 5

4.1 Codeword Spheres 22
4.2 Hamming Space Hypercube 23
4.3 Length 3 Distance 2 Code 24
4.4 Equivalent Code from Rotation or Reflection 24

5.1 Edit space up to length 3. Dotted lines are insertions or deletions. Solid lines are substitutions. λ is the null string. [6] . . 26
5.2 Dynamic Programming Array for finding Edit Distance 27
5.3 Row 0 and Column 0 Filled 28
5.4 Completed Array and Traceback 29

6.1 Codeword Graph 32

9.1 Unimodal Curve 47
9.2 Average Clique Size vs α for $(6,3)$ Codes 49
9.3 Best Clique Size vs α for $(6,3)$ Codes 49
9.4 $\sigma = .5$ Average Clique Size vs α 52
9.5 $\sigma = .333$ Average Clique Size vs α 52
9.6 (6,4) Average Clique Size vs α with 95% CI 54
9.7 (8,4) Average Clique Size vs α 55
9.8 Alpha .30 Results Distribution 56
9.9 Alpha .33 Results Distribution 56
9.10 Alpha .34 Results Distribution 57
9.11 Alpha .35 Results Distribution 57
9.12 Alpha .36 Results Distribution 58
9.13 Alpha .37 Results Distribution 58
9.14 Length 6 Average Cover Size vs ϕ with 95% CI 63

9.15 Complete Covers vs ϕ 64
9.16 Length 6 Average Words Covered vs ϕ with 95% CI 64
9.17 Length 7 Average Cover Size vs ϕ with 95% CI 65
9.18 Length 7 Average Words Covered vs ϕ with 95% CI 66

Chapter 1

Introduction

1.1 Overview

One of the most important quests in computer science today is the search for techniques to find approximate solutions to problems for which exact solutions cannot be obtained in a practical amount of time. One common method for finding good solutions to these NP-hard problems is the use of a metaheuristic such as Simulated Annealing or Genetic Algorithms. This thesis will examine a new algorithm, the Salmon Algorithm, which combines concepts from existing metaheuristics.

There are currently several hundred known NP-hard problems. For example, Pierluigi Crescenzi and Viggo Kann [9] count over 200, and this number only includes problems for which there is known research into approximate solutions. Although there are specialized algorithms for many of these problems (see for example [28]), generalized search techniques such as Genetic Algorithms or Artificial Ant Algorithms are often employed for approximate solutions.

While the popularity of these methods is undeniable, there can be difficulties with their implementations on some problems. Moreover, the No Free Lunch Theorem states that if any algorithm has superior performance in one class of problem, it will necessarily have inferior performance in another class [41]. Thus, a wider variety of search techniques means a greater chance that one of them may perform well in a given search space.

1.2 Problem Statement

The Traveling Salesman Problem (TSP) is one of the best known problems in combinatorial optimization. Given a list of n cities, the object is to find the shortest route that visits each city exactly once, then returns to the first city. This problem has exponential complexity in n. The Salmon Algorithm will be tested on one instance of the TSP.

An $(n, d)_q$ error correcting code is a set of words of length n from an alphabet of size q where all words in the code are at a minimum distance of d. There are a number of possible distance measures that can be used. The two this thesis will be using are edit distance and Hamming distance, which will be explained in greater detail in the following chapters. It is a fundamental problem in coding theory to find the largest possible size of an $(n, d)_q$ code for given q, n, and d. This problem has an exponential complexity in n, hence it is a good candidate for analysis by some type of search metaheuristic.

This thesis will be using the Salmon Algorithm to find the largest possible code for a number of different combinations of n, q, and d. Codes using both edit distance and Hamming distance will be constructed. The results will be compared to the largest codes that have been created by previous researchers.

An $(n, r)_q$ covering code W is a set of words of length n from an alphabet of size q. Each word w in W is said to 'cover' every word that is within a distance r from w. If all q^n words are covered by at least one word in W, then W is a covering code. One of the best known problems in coding theory is the quest to find the smallest covering code using a given n, q and r. This problem also has exponential complexity in word length. This thesis will use the Salmon Algorithm to construct minimum covering codes.

1.3 Organization

Chapter 2 is an overview of search techniques. It will review some of the algorithms that have been used by previous researchers on the maximum or minimum code problems - Genetic Algorithms, Simulated Annealing, and Tabu Search. It will also detail two metaheuristics that the Salmon Algorithm borrows ideas from - Evolutionary Strategies and Artificial Ant Algorithms.

Chapter 3 will introduce the Salmon Algorithm and demonstrate how this algorithm can be applied to the Traveling Salesman's problem. The

results from the *Berlin52* TSP problem will be given and compared to those achieved with a Genetic Algorithm and an Artificial Ant Algorithm.

Chapter 4 will explain noisy channels and show how this creates the need for error correcting codes. It will be demonstrated how codes defined using Hamming distance can be used to correct transmission errors. Hamming distance, Hamming space and the concept of equivalent codes will be explained.

Chapter 5 will introduce the concepts of edit codes, edit distance, and edit space. A dynamic programming algorithm for calculating edit distance will be explained in detail.

Chapter 6 will look at the problem of finding codes of the maximum possible size and show that the problem of finding such a code is the equivalent of finding a maximum clique in a graph. This chapter will also demonstrate how the Salmon Algorithm can be configured to find maximum cliques (codes). It will briefly look at memory optimization, and introduce Conway's Lexicode Algorithm, which is a common algorithm used for finding large codes.

Chapter 7 will explain the concept of covering codes in more detail. It will also introduce the football pool problem, which is a special case of the covering code problem. It will show how the Salmon Algorithm can be set up to find minimum covering codes.

Chapter 8 will give a brief review of previous research on maximum edit codes, maximum Hamming codes, and minimum covering codes, specifically those related to the football pool problem.

Chapter 9 will give the experimental results, beginning with the maximum edit code problem. It will demonstrate how parameter values were optimized. The sensitivity of some parameter values will be shown. Salmon algorithm edit code results will be compared to previous results. The concept of GC content will be explained and results with fixed GC content will be given. Maximum Hamming code results for binary and ternary Hamming codes will then be given and compared to the results from previous research. Finally, the results for the length 5, 6, and 7 football pool problems will be given and compared to previous results.

Chapter 10 will give the conclusions and explore possibilities for future work.

Chapter 2

Overview of Search Techniques

Figure 2.1 shows a hierarchy of search techniques. This chart is by no means complete. It is only intended as a guide so the reader can place the methods that will be discussed in some logical category.

Calculus based techniques are applicable only to functions that are differentiable, and are therefore not useful for the combinatorial optimization problems we will be looking at. Enumerative techniques are probably not the best choice for the problems we will be examining because the search spaces are too large. It is possible that some type of exhaustive search using advanced pruning techniques (such as removing solutions that are isomorphic) might be used, but this is not the direction this thesis will be pursuing because we are concentrating on cases for which this is not computationally feasible. The techniques that we will be exploring in more detail are types of guided random searches: Evolutionary Strategies (often called Evolution Strategies), Genetic Algorithms, Artificial Ant Algorithms, Simulated Annealing and Tabu Search.

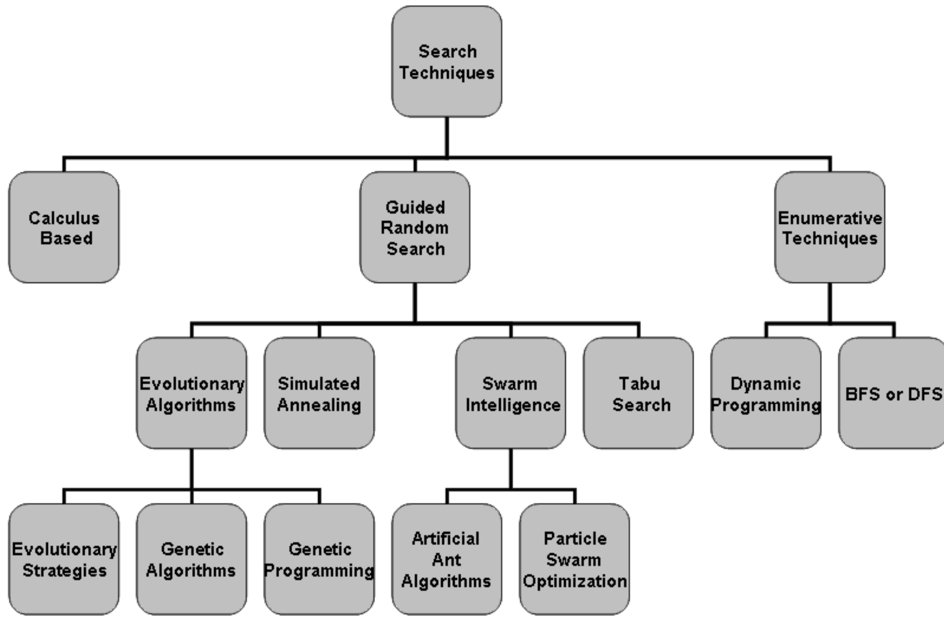

Figure 2.1: Search Hierarchy [42]

2.1 Evolutionary Strategies

Perhaps the simplest evolutionary algorithm is an Evolutionary Strategy (ES). According to Rechenberg [30], Evolutionary Strategies were first developed in 1964 at the Technical University of Berlin. The algorithm is normally used to optimize some function $F()$ in n dimensional space. In an Evolutionary Strategy, there exists a population of chromosomes, each of length n. Each chromosome represents a point in n dimensional space. The algorithm takes each chromosome (the parent) and mutates it to create a child. $F(parent)$ and $F(child)$ are compared, and the point which gives the best value is retained in the population. The mutation normally consists of taking the value of each dimension of the chromosome and selecting a new value from a Gaussian distribution about the old one.

Evolutionary Strategies are normally used to solve engineering problems, since the mutation produces a floating point number. They can be modified

for use on combinatorial optimization problems, but a different scheme for producing mutations must be used.

```
To optimize some function F()
Create random population of chromosomes
    for g generations or until convergence do
        for each chromosome_k in k = 1..numChromosome do
            child = mutate(chromosome_k)
            if F(child) > F(parent)
                chromosome_k = child
            end if
        end for
    end for
```

Algorithm 1 Evolutionary Strategy

2.2 Genetic Algorithms

Like Evolutionary Strategies, Genetic Algorithms (GA) are based on natural selection, mutation and evolution. They differ from ES in that they are normally used for combinatorial optimization problems, so the chromosomes are not composed of floating point values. Instead, a chromosome consists of a string of letters from an alphabet of finite size. A binary alphabet is often used, although other sizes are not uncommon [10].

Each chromosome represents one complete solution to the problem. Initial chromosomes are normally generated randomly, although occasionally the initial chromosomes are filled with known good values. This process is known as seeding.

Unlike an Evolutionary Strategy, a GA employs a crossover operator which simulates the real world process of mating. Crossover consists of taking two chromosomes P_1 and P_2 (the parents), then forming two new chromosomes C_1 and C_2 (the children) by combining parts of P_1 with parts of P_2. There are many different types of crossover operators. The simplest is the single point crossover, which is shown below.

$$P_1 = ACGTTCATGGC$$
$$P_2 = TGCAGGCAATA$$

Suppose we randomly select the crossover point after the fifth gene.

$$P_1 = ACGTT|CATGGC$$
$$P_2 = TGCAG|GCAATA$$

Produce the two children by exchanging the genes after the crossover point.

$$C_1 = ACGTT|GCAATA$$
$$C_2 = TGCAG|CATGGC$$

Some problems, such as the Traveling Salesman Problem, require specialized crossover operators [10]. Using a crossover like the single point on the TSP will probably result in a tour that contains some cities twice and others not at all. This is an illegal tour, so the crossover operator must be designed in such a way that only legal tours are produced.

A Genetic Algorithm must also incorporate a fitness function. The fitness function determines the most fit members of the population for crossover. The fitness function mimics the real world process of Darwinian selection, where members of the population that are best adapted to a particular environment (i.e. the most fit) have a better chance of surviving and producing children.

The selection of parents for reproduction in a GA is usually done with tournament selection. In tournament selection, we set the size of the tournament at some fixed size m. This means we choose m individuals randomly from the population and evaluate each using the fitness function. The most fit of these is retained for reproduction. This process is repeated until the required number of children have been produced. Larger tournament sizes tend to increase the selection pressure (i.e. they favour the most fit).

Like ES, a Genetic Algorithm also employs a mutation operator. However, in GA's the mutation operator is not used every time a new chromosome is created. Instead, it is used randomly on a small proportion of the population after the crossover operation takes place. This small proportion is typically 10% or less.

As with Evolutionary Strategies, a GA is run for a number of generations. This number is normally determined empirically. Alternately, rather than fixing the number of generations, the algorithm can be run until it converges.

Convergence means there has been no improvement in the best solution for a significant number of generations.

Genetic Algorithms sometimes employ an elite chromosome. In this case the most fit member of the population in any generation is saved and directly inserted into the population for the subsequent generation.

Using fitness function $Fit()$
Using crossover operator $Xover()$
Using mutation operator $Mutate()$
Create random *population* of chromosomes
  for g generations or until convergence **do**
    $childPopulation = null$
    for $count = 1$ to size of *population* **do**
      $parent_1 =$ randomly select from *population* using $Fit()$
      $parent_2 =$ randomly select from *population* using $Fit()$
      $child = Xover(parent_1, parent_2)$
      if small probability
        $child = Mutate(child)$
      end if
      $childPopulation = childPopulation + child$
    end for
    $population = childPopulation$
  end for
return the most fit individual in *population*

Algorithm 2 Genetic Algorithm [31]

2.3 Artificial Ant Algorithms

Ant Algorithms were first proposed by Marco Dorigo in his PhD thesis in 1992 [11]. The first problem to be solved using this method was the Traveling Salesman's problem (TSP). Since then Ant Algorithms have been applied to different combinatorial optimization problems such as vehicle routing, the knapsack problem, and job shop scheduling.

Ant Algorithms were inspired by real world ants, which find the shortest path by pheromone communication. Presented with a number of different paths, none of which have been used before, ants will choose randomly. As

it makes its way along a path, an ant will deposit pheromone. Since the ants on shorter paths will travel back and forth more quickly, they will make more trips in a given period of time. Since the ants are making more trips, these routes will have a greater amount of pheromone deposited. In general, ants will prefer a trail with higher pheromone levels, so that over time short paths will come to be preferred by the majority of ants. Pheromones evaporate slowly, which means that eventually the older, longer paths lose their pheromones and become less likely to be selected, while the newer, shorter paths will tend to predominate.

In 1997 Dorigo and Gambardella [12] introduced an improvement to the Ant Algorithm they called Ant Colony System. This algorithm differed from the basic Ant Algorithm in three ways. i) A *pseudo-random-proportional* rule is used for deciding which edge to select (state transition rule). ii) Only the best ant has a global pheromone update. iii) A local pheromone updating rule is added. The Ant Colony Algorithm is given as Algorithm 3.

```
for g generations or until convergence do
    Each ant is positioned on a starting node.
    for count = 1 to number of ants do
        Each ant builds a solution as follows
        for count = 1 to number of vertices do
            Apply state transition rule
            Apply local pheromone update
        end for
    end for
    Apply global pheromone update
end for
Return the shortest path
```

Algorithm 3 Ant Colony Algorithm [12]

The *pseudo-random-proportional* rule, given below, allows the programmer to bias the algorithm towards exploration or exploitation by adjusting the parameter q_0. An ant positioned on node r chooses the node s to move to by using this state transition rule. Here, $\tau(r,s)$ is the amount of pheromone on edge (r,s), $d(r,s)$ is the length of edge (r,s), q is a random number between 0 and 1, *candidates* are all vertices that have not yet been visited, and $u \in candidates$. The rule is as follows:

$$s = \begin{cases} argmax[\tau(r,u)] * [1/d(r,u)]^{\beta} & \text{if } q < q_0\,, \\ S & \text{otherwise}\,. \end{cases}$$

Here S is a random variable selected according to the formula given below. The following formula gives the probability an ant on node r will move to node s.

$$p(r,s) = \frac{[\tau(r,s)] * [1/d(r,s)]^{\beta}}{\displaystyle\sum_{u \in Candidates} [\tau(r,u)] * [1/d(r,u)]^{\beta}}$$

This formula is termed roulette selection. The probability an edge will be selected is proportional to its fitness. (For this reason it is also sometimes called fitness proportionate selection.)

Thus, if $q < q_0$ the algorithm uses a greedy approach and selects the edge with the best combination of length and pheromone. Otherwise, it uses roulette selection to find the next edge (r,s).

As each individual ant constructs a path, it applies the local pheromone update given below.

$$\tau(r,s) \leftarrow \tau(r,s) * (1-\rho) + \rho * \tau_0$$

This rule tends to decrease the amount of pheromone on an edge. This might seem counterintuitive, but there is a logical reason for this reduction. Consider what would happen if each ant added pheromone after it had constructed a path. Since ants favour paths with more pheromone, the next ant would tend to build the same solution as the first. It would then reinforce this path with more pheromone, and so on. The reduction in pheromone forces the ants into a wider exploration.

After all ants have constructed a solution the global pheromone update is applied.

$$\tau(r,s) \leftarrow \tau(r,s) * (1-\alpha) + \alpha * \Delta\tau(r,s)$$

where $\Delta\tau(r,s)$ is given by

$$\Delta\tau(r,s) = \begin{cases} (L_{gb})^{-1} & \text{if } (r,s) \in \textit{global best tour} \\ 0 & \text{otherwise} \end{cases}$$

Here, L_{gb} is the length of the global best tour and $0 < \alpha < 1$ is the pheromone decay parameter. α simulates pheromone evaporation.

The primary problem with the basic Ant Algorithm is that it gets trapped too easily in local optima. The Ant Colony System has better performance, especially on TSP problems with more than 30 cities [12].

2.4 Simulated Annealing

Simulated Annealing was introduced by Kirkpatrick et al.[24] in 1983. It is based on the process of annealing in metals in which a heated metallic object is allowed to slowly cool. This enables the molecules to crystallize into the lowest energy state. The Simulated Annealing algorithm is given as Algorithm 4.

function $Neighbour(C)$ produces a random neighbour of a solution C
function $Fitness(C)$ returns the fitness of a solution C
Start from $T = T_0$ and an initial solution $C = C_0$
while stop criteria not satisfied **do**
  $C' = Neighbour(C)$
  $E = Fitness(C') - Fitness(C)$
  if $E > 0$
    $C = C'$
  else
    $C = C'$ with probability $e^{E/T}$
  end if
  Decrease T
end while
return C

Algorithm 4 Simulated Annealing [27]

The key to understanding Simulated Annealing is to examine how the algorithm decides whether or not to accept a successor C' to a solution C. If C' has higher fitness than C, then it is always accepted. If C' has a lower fitness than C, then it is accepted with probability $e^{E/T}$. This has two effects. First, very bad moves will have large negative values for E, and thus will be accepted with low probability. Second, early in the algorithm, when T is high,

bad moves are more likely to be accepted than late in the algorithm when T is low. This allows the algorithm to find its way out of local optima early, then concentrate its search in the area of the global optimum (hopefully) as the algorithm nears completion.

Note that the algorithm is normally not population based. Of course, it would be possible to create a population of solutions, but this would essentially be no different than running a number of single solution algorithms in parallel.

2.5 Tabu Search

A Tabu Search begins with a solution C, then generates a candidates list of all neighbours of C. The fitness of all the candidates is evaluated, and the most fit is retained as the current solution C'. Note that this is true even if the fitness of C' is less than the fitness of C. If we were to simply repeat this cycle, the most likely outcome is that the algorithm would return to C on the next iteration. To prevent this from happening the algorithm maintains a list of the n most recent solutions in a tabu list. These solutions are forbidden, so the algorithm cannot simply go back to its previous position. Note that the tabu list can be overridden if the resulting solution satisfies some special aspiration condition. Tabu Search is given as Algorithm 5.

function $Neighbour(C)$ produces S, the set of all neighbours of C
function $MaxFitness(S)$ returns the most fit of a set of solutions S
Start from an initial solution $C = C_0$
while stop criteria not satisfied **do**
    $S = Neighbour(C)$
    $C = MaxFitness(S)$
    while C is on tabu list **and** C does not satisfy aspiration criteria
        $S = S - C$
        $C = MaxFitness(S)$
    end while
    update tabu list, aspiration criteria, and global best solution
end while
return global best solution

Algorithm 5 Tabu Search [15]

Chapter 3

Salmon algorithm

The Salmon Algorithm was inspired by the behaviour of salmon swimming upstream to spawn. This thesis makes no claim that real world salmon exhibit the same characteristics as the software salmon. The behaviour of the simulated salmon is idealized. However, there are commonalities.

We know real salmon are attracted to flow. In fact, salmon will not begin spawning if there is insufficient water level in the river [26]. We also know salmon have a memory. Salmon will return to spawn in the same location where they were born. These two facts have been used to create a search meta-heuristic called the Salmon Algorithm. The following is a description of the Salmon Algorithm for the Traveling Salesman's Problem.

The algorithm begins by creating a population of salmon and a water level manager. A water level manager is a 2D array of all edges in the graph, where the value for each edge equals the water level on that edge. The water levels are initialized to some positive value.

Each salmon contains two lists. The first is a tabu list, which is a list of all vertices visited thus far on the current tour. The second is a memory list. The memory list is a copy of the completed tabu list from the salmon's parent. In other words, each child remembers the path its parent took the previous generation. To begin the algorithm a random path is generated and placed in each salmon's memory list.

The salmon then begins its circuit at the first vertex in its memory list. At each vertex in its path, the salmon will tend to follow the path that its parent used with probability ϕ. If it does not select the edge that its parent used, it will select an edge from a list of candidate edges. Candidate edges are those that connect to a vertex that is currently not in the tabu list. This

selection is done using roulette selection on water level values. After each selection, the candidates list is updated.

After all salmon have built a complete path, the water level manager is updated. Each salmon[i] adds an amount of water to each edge in its path equal to $bestknown/pathlength[i]$. Here, *bestknown* is the length of the shortest path known to exist for this problem. This is similar to the way an ant deposits pheromone in an Ant Algorithm. The primary difference is that pheromones evaporate, while simulated water does not.

The most fit σ of the salmon spawn and produce offspring. For algorithmic simplicity, σ is a fraction $1/n$, for an integer n. This allows a constant population to be maintained if each surviving salmon produces n children. The spawning process is essentially cloning. There is no crossover between salmon as there would be with a Genetic Algorithm. The tabu list of the parent salmon is copied to the memory list in the baby salmon. The most fit of the parent salmon is saved to an elite salmon. The parents then "die" and the cycle repeats. Pseudocode for the Salmon Algorithm is presented as Algorithm 6.

```
Initialize water levels
for each salmon_k do
    place random solution in memory_k
end for
for g generations do
    for each salmon_k construct a solution as follows
        construct candidates list
        for each item_m in memory
            select item_m with probability φ
            if item_m not selected then select from candidates using roulette
            update candidates list
        end for
    end for
    update water levels
    produce children
    save elite
end for
return elite
```

Algorithm 6 Salmon Algorithm

The probability p_n of selecting $item_n$ using roulette selection is given by:

$$p_n = \frac{(waterLevel_n)^{\alpha}}{\sum_{x \in Candidates} (waterLevel_x)^{\alpha}}$$

A high value for the roulette selection exponent α will bias the algorithm more towards exploitation, since the formula will show a strong preference for known good values. Low α values bias the algorithm more towards exploration, since the formula then has only a weak preference for higher water levels.

In summary, each child inherits about ϕ of its path from a parent, just as in a GA. Instead of getting the remainder from a second parent as would occur in a GA, the algorithm builds the rest of the path using a method similar to an Artificial Ant Algorithm. Note that there is no use of tournament selection in the reproductive process, as is common in GA's. The least fit of the salmon are completely excluded from the reproductive pool.

3.1 Salmon Algorithm Parameter Values for TSP

The values shown in Table 3.1 were found to give good results with the Berlin52 Traveling Salesman data. This data set, which consists of 52 locations in Berlin Germany, was used because it is a well studied problem instance. The locations are stored as a list of (x,y) coordinates which have integer values. However, the distances between locations will generally not be integer values because of the Pythagorean theorem.

These parameter values were obtained by experimentation. Since the Salmon Algorithm needs to be tuned for each problem instance, not just each problem type, these values may not work well for other Traveling Salesman data sets.

parameter	optimum
σ	0.1
ϕ	0.5
$number\ of\ salmon$	400
α	3.0

Table 3.1: Salmon Algorithm Optimum Values for TSP

3.2 Experimental Results for the TSP

Table 3.2 shows how the results of the Salmon Algorithms on the Berlin52 Traveling Salesman data compare to the results from a Genetic Algorithm and an Ant Colony Algorithm. The GA and ant algorithm results were obtained from research that was conducted prior to this thesis. Ten runs of each algorithm were used.

The Ant Colony Algorithm was run with the parameter values suggested in [12]. They are $\alpha = .1$, $\beta = 2$, $q_0 = .9$, $\rho = .1$ and τ_0 equal to $(n * L_{nn})^{-1}$ where L_{nn} is the tour length produced by the nearest neighbour heuristic and n is the number of cities. The algorithm was run for 300 iterations.

The Genetic Algorithm was run with crossover = 100%, mutation = 10%, one elite, tournament size 2, and cycle crossover. These values were obtained via experimentation. The algorithm was run with a population size of 4000 for 1000 generations. The population and number of generations were chosen so that the run time of the GA would be close to the run time of the Ant Colony Algorithm. Note that cycle crossover is a specialized crossover that is used on problems that have ordered chromosomes, such as the TSP. For a complete description see [10].

We say the algorithm has converged when there is no further improvement in the best value for that run. The time to converge is the average number of seconds the algorithm took to reach convergence. The optimal path for this problem has a length of 7544.37 (with floating point values). The Salmon Algorithm found either this value or 7544.66 on 4 out of 10 runs.

Algorithm	**best**	**average**	**worst**	**time to converge (seconds)**
Salmon	7544.37	7635	7860	6.28
Ant	7548	7694	7830	21.7
GA	7819	8147	8560	26.1

Table 3.2: TSP Results

Chapter 4

Error Correcting Codes

Anyone who has listened to AM radio during a thunderstorm is aware that communication channels are not perfect. Extra information in the form of noise may be added to the transmitted signal, with the result that the received signal differs from the original transmission.

Prior to 1948 communications was considered to be an engineering discipline, not a scientific one [8]. Engineers approached the problem of noise from a purely technical standpoint. For example, they might choose frequency modulation over amplitude modulation or increase transmitter strength to improve the signal to noise ratio. In that year Claude Shannon, a scientist working at the Bell Telephone Labs, published a landmark paper titled "A Mathematical Theory of Communication"[32]. Shannon identified a characteristic of communication channels called capacity, and proved that even noisy channels have the ability to transmit reliable information, provided this information is sent at a rate below the channel's capacity.

No codes were actually produced in Shannon's proof. Shannon's theorem only guarantees that they exist [20]. The construction of usable error correcting codes was pioneered by a contemporary of Shannon's at the Bell Labs, R.W. Hamming. In 1950 Hamming published a paper titled "Error Detecting and Error Correcting Codes" [18] which introduced the well known Hamming(7,4) code. This code has a word length of 7 and each word contains 4 bits of information. The four information bits can transmit a value from 0 to 15 i.e. one hexadecimal digit. The Hamming (7,4) code is given in Table 4.1.

Hamming Code Word	Value
0000000	0
1101001	1
0101010	2
1000011	3
1001100	4
0100101	5
1100110	6
0001111	7
1110000	8
0011001	9
1011010	10
0110011	11
0111100	12
1010101	13
0010110	14
1111111	15

Table 4.1: Hamming (7,4) Code

Hamming constructed this code by building on the concept of parity. Parity, within the context of error detection, means that the number of 1 bits in a word is always either even (even parity) or odd (odd parity). We will restrict ourselves to a discussion of even parity, since this is what Hamming used. Adding a parity bit to a binary number allows us to detect a single error. The parity bit is set to either 0 or 1, so that the total number of 1's in the number is always even.

Suppose we wish to transmit the 7 bit binary number 0110001. Begin by appending a parity bit to the end of this number. Our number, 0110001, has three 1 bits so the parity bit will be 1, giving us 01100011. This makes the total number of 1's equal to four, which is even. If any single bit gets flipped during transmission the total number of 1 bits will be odd, indicating that an error has occurred.

Hamming used three parity bits in his (7,4) code. The parity bits are placed where the position number is a power of 2. Thus, position numbers 1, 2, and 4 are parity bits. (Note that we are counting from 1, not from 0.)

The first parity bit checks any position number where there is a 1 as the

last digit in its binary representation. We see that:

$$\begin{aligned} 1 &= 001 \\ 3 &= 011 \\ 5 &= 101 \\ 7 &= 111 \end{aligned}$$

Thus, the first parity check uses positions 1, 3, 5, and 7. Similarly, the second parity check uses positions where the second digit of its binary representation is a one.

$$\begin{aligned} 2 &= 010 \\ 3 &= 011 \\ 6 &= 110 \\ 7 &= 111 \end{aligned}$$

Finally, the third parity check uses positions

$$\begin{aligned} 4 &= 100 \\ 5 &= 101 \\ 6 &= 110 \\ 7 &= 111 \end{aligned}$$

Let us examine how Hamming used this code to correct a transmission error. Suppose the transmitted word is 0111100, or 12 from Table 4.1. This corresponds to the original message (without check bits inserted) 1100. Suppose also that an error has occurred in transmission which changes the fifth bit from 1 to 0. The received word will be 0111000.

The first parity bit checks positions 1, 3, 5, and 7. It should be 1, but it is 0. Therefore, the error is in either position 1, 3, 5, or 7.

The second parity bit checks positions 2, 3, 6, and 7. It has the correct value. Thus the error is not in any of these positions. Combining this with the result of the previous check, we see the error can only be in position 1 or 5.

The last parity bit checks positions 4, 5, 6, and 7. It has the incorrect value. Thus the error is in position 4, 5, 6, or 7. Combining this with the results of the previous two checks, we can see that the error must be in position 5. The correct value is obtained by changing the 0 in the fifth position to a 1.

In his 1950 paper, Hamming also introduced a new distance metric which has since come to be known as Hamming distance. Given two strings of length n, $S_1 = x_1, x_2...x_n$ and $S_2 = y_1, y_2...y_n$, the Hamming distance d is defined as the number of substitutions required to change S_1 into S_2. Viewed a different way, this value is the same as the number of positions in which the two strings have different characters. All of the words in the Hamming (7,4) code are at a minimum distance of three.

The algorithm to compute the Hamming distance is a very simple order n algorithm that counts the number of positions in which two strings differ.

Read $S_1 = \{x_1, x_2...xn\}$ and $S_2 = \{y_1, y_2...y_n\}$
$distance = 0$
for i = 1 to n do
  if $x_i \neq y_i$ **then** $distance = distance + 1$
end for
return $distance$

Algorithm 7 Hamming Distance

Let us examine how we use the concept of Hamming distance to detect and correct an error in a transmission, using the same example that was used above, 0111100 being transmitted and 0111000 received.

We first check all words in Table 4.1 and see that 0111000 does not appear. This means an error has occurred. We then compute the Hamming distance between 0111000 and all the words in the table. We find that the Hamming distance between 0111000 and 0111100 (12) is one, while the distance to all other words is 2 or more. Thus, the received word decodes to 12. This type of decoding is referred to as maximum likelihood decoding. We make the assumption that a single error will occur more often than two or more, and we consequently choose the closest word in Hamming space.

It is possible to view Hamming space as a collection of spheres of radius $d/2$ centred about each code word, as shown in Figure 4.1. Every word contained in a sphere decodes to the code word at the centre. A code can correct up to $t = \lfloor (d-1)/2 \rfloor$ errors. For this reason the spheres are frequently assigned a radius of t rather than $d/2$.

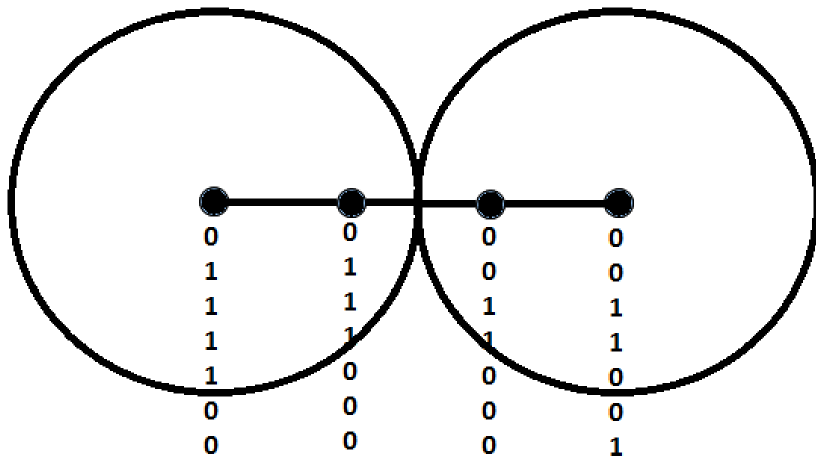

Figure 4.1: Codeword Spheres

It is possible in some codes that a word will be equidistant to two or more codewords. In this case decoding is ambiguous. It is also possible that some words will be more than distance t from every codeword. These also cannot be corrected. If neither of these problems exists then every word is contained uniquely in one codeword sphere and the code is said to be *perfect*.

If a noisy transmission results in 2 errors in a given word, then the Hamming (7,4) code will decode incorrectly. Using Figure 4.1, if the transmitted word is 0111100 and the received word is 0011000 (2 errors), then the received word will decode to 0011001. If the transmission medium is so noisy that 2 errors are a frequent possibility, then we should use a distance 5 code. A distance 5 code can correct 2 errors. If we wish to correct 3 errors, then a distance 7 code must be used. But a length 7 Hamming code with distance 7 has only 2 codewords - $\{0000000, 1111111\}$ is one possibility. This demonstrates a fundamental trade off with error correcting codes, greater redundancy means less information. If no error correcting was required a 7 bit word could transmit $2^7 = 128$ unique pieces of information. If we need to correct one error then only 16 unique words can be sent. The ability to correct three errors limits us to only two distinct words.

4.1 Hamming Space

The Hamming space for binary strings of length n can be viewed as an n dimensional hypercube as shown in Figure 4.2. Edges connect words that are at a distance of one. The distance between two words w_1 and w_2 is the minimum number of edges that must be traversed to move from w_1 to w_2. This allows us to visualize equivalent codes as rotations or reflections. The term equivalent codes means that the codes are isomorphic.

Equivalent codes will have the same characteristics: length, alphabet, distance, and number of words. If a code is perfect, then any equivalent code will also be perfect. If a code is not perfect, then any equivalent code will share this trait. Equivalent codes have the same shape in Hamming space.

Let us look at a binary length 3 distance 2 code. We see that such a code {000, 011, 101, 110} can be created by selecting words that are at opposite corners of every two dimensional square. This is shown in Figure 4.3. Rotation of the cube about any axis, or reflection through any plane, will produce the equivalent code of {111, 100, 010, 001}, shown in Figure 4.4. These are the only two binary length 3 distance 2 codes of size 4.

The above two codes are equivalent. One can be converted to the other by a permutation of the letters. (In a binary code the only possible permutation of letters is $0 \to 1$ and $1 \to 0$.) In the Hamming space equivalent codes are created by permuting the letters and/or permuting the positions in the words.

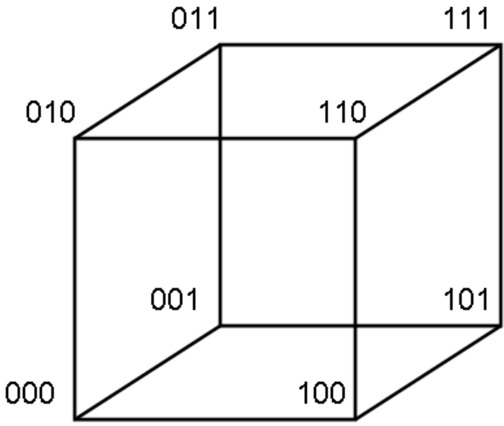

Figure 4.2: Hamming Space Hypercube

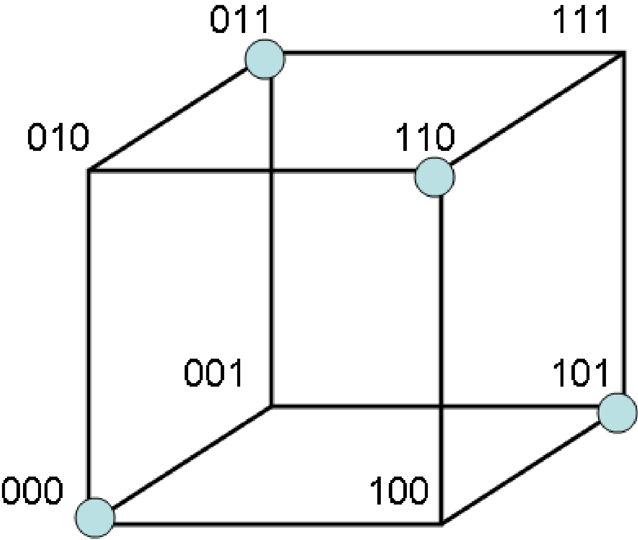

Figure 4.3: Length 3 Distance 2 Code

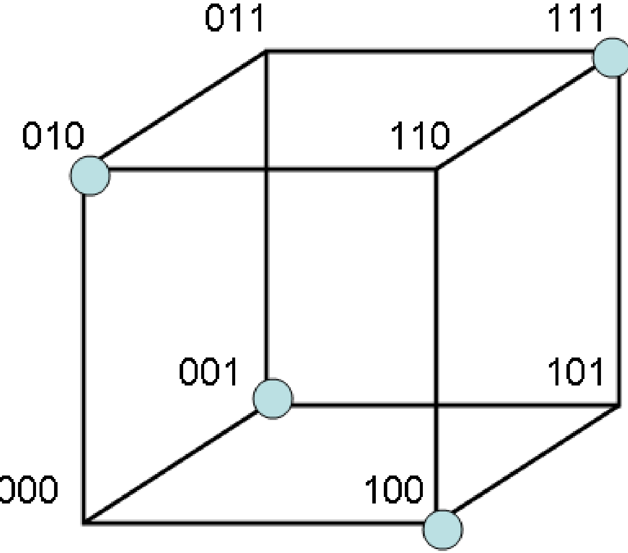

Figure 4.4: Equivalent Code from Rotation or Reflection

Chapter 5

Edit Codes

The edit distance between two words is defined as the minimum number of any combination of single character insertions, deletions, or substitutions required to change one word into the other. Edit distance is also known as *Levenshtein* distance. Two words can be far apart in Hamming space but close in edit space. For example, the words 01010101 and 10101010 are at a distance of 8 in Hamming space but only a distance 2 in edit space (The first word can be changed to the second by inserting a leading 1 then removing the trailing 1.)

While Hamming codes are normally associated with error correction in data transmission or storage, edit codes have applications in bioinformatics because errors in DNA can occur as either substitutions, deletions or insertions. For this reason, all experimental work on edit codes in this thesis will be done using an alphabet of size 4, which is the number of letters in the DNA alphabet.

As can be seen from Figure 5.1, the edit space is much more complex than Hamming space. This graph can be viewed as a series of layers of Hamming spaces joined vertically by insertions and deletions. Equivalent edit codes can be created only by simultaneously permuting the letters in all the words or by the reversal of all the words[6]. Thus, there are much smaller numbers of equivalent edit codes than there are equivalent Hamming codes.

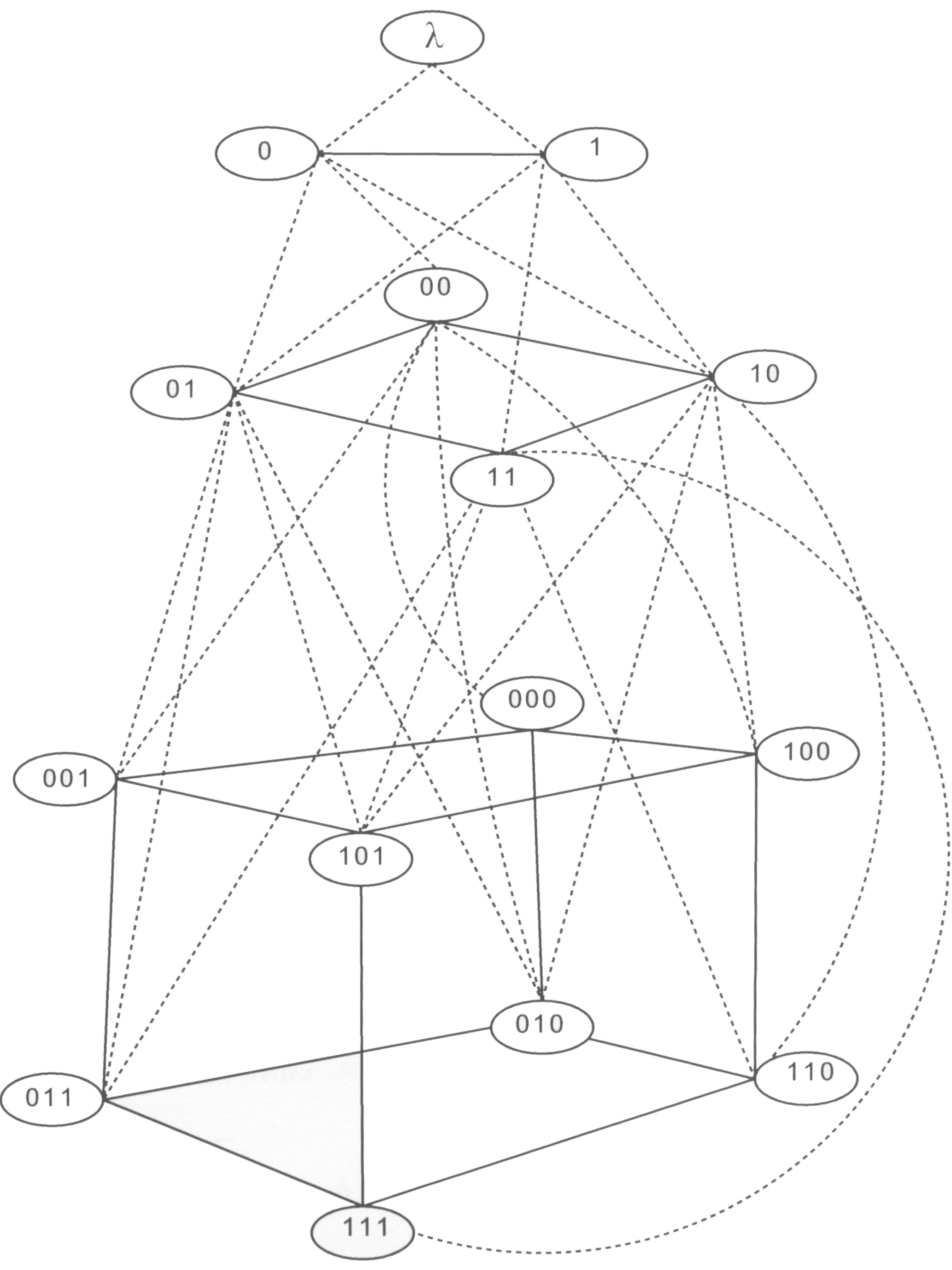

Figure 5.1: Edit space up to length 3. Dotted lines are insertions or deletions. Solid lines are substitutions. λ is the null string. [6]

5.1 Dynamic Programming Algorithm for Edit Distance

Calculation of edit distance is typically done with a dynamic programming algorithm. This algorithm will now be explained by way of an example.

Begin by constructing an array L with the rows corresponding to the letters of the source word and the columns corresponding to the letters of the destination word as shown in Figure 5.2. λ represents the null string.

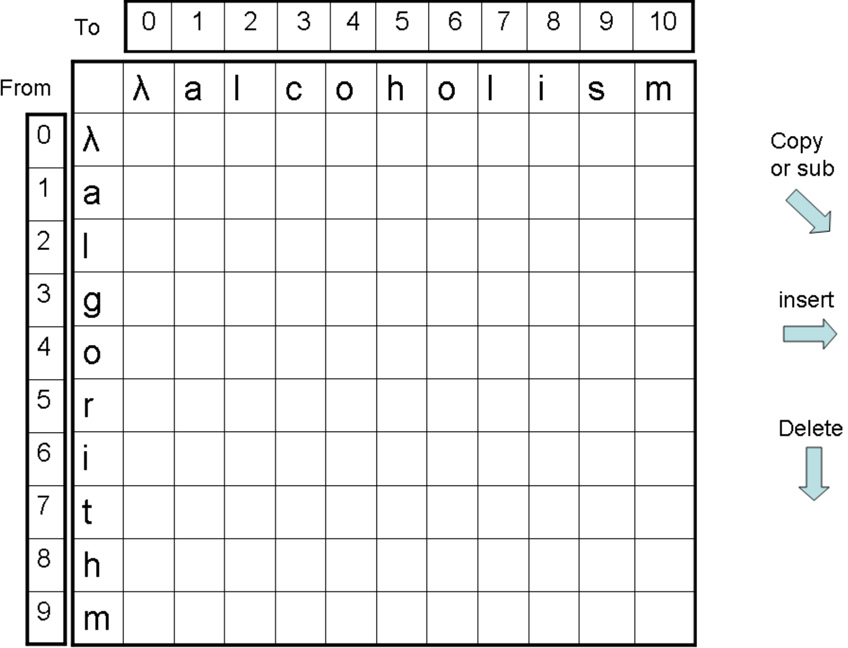

Figure 5.2: Dynamic Programming Array for finding Edit Distance

Moving to the right in the array represents an insertion. Moving down represents a deletion. Moving diagonally represents a substitution or a copy. A copy can only be performed in $L[i,j]$ if $source[i] = destination[j]$. The cost of a substitution, deletion or insertion is one. The cost of a copy is zero. We will determine the edit distance from 'algorithm' to 'alcoholism'.

The value at any point in the array $L[i][j]$, will be the edit distance between $source[i]$ and $destination[j]$. Thus, the value at $L[4][5]$ is the edit

distance from ‘algo’ to ‘alcoh’. The value at $L[6][7]$ is the edit distance between ‘algori’ and ‘alcohol’.

The recursive formula for this algorithm is given below.

If $source[i] = destination[j]$

$$L[i][j] = Min \begin{cases} L[i-1][j-1] + 0 \\ L[i][j-1] + 1 \\ L[i-1][j] + 1 \end{cases}$$

If $source[i] \neq destination[j]$

$$L[i][j] = Min \begin{cases} L[i-1][j-1] + 1 \\ L[i][j-1] + 1 \\ L[i-1][j] + 1 \end{cases}$$

From / To		0	1	2	3	4	5	6	7	8	9	10
		λ	a	l	c	o	h	o	l	i	s	m
0	λ	0	1i	2i	3i	4i	5i	6i	7i	8i	9i	10i
1	a	1d										
2	l	2d										
3	g	3d										
4	o	4d										
5	r	5d										
6	i	6d										
7	t	7d										
8	h	8d										
9	m	9d										

Copy or sub ↘

insert →

Delete ↓

Figure 5.3: Row 0 and Column 0 Filled

Begin by filling $L[0][j] = j$ and $L[i][0] = i$ as in Figure 5.3. Also place in each array element an 'i' if the value was arrived at via an insert, a 'c' if a copy was used, an 's' for substitution, and a 'd' if the element was reached via a delete. This will assist in the traceback. If only the distance is required, and not the sequence of steps needed to convert the source to the destination, then this step can be omitted.

Fill the remaining elements in row major order according to the recursive formula. For example, $L[1][1]$ will have a value of 0, since $source[1] = destination[1]$, $L[1][1]$ is filled with a copy from $L[0][0]$. Figure 5.4 shows the completed array including the traceback.

From \ To		0	1	2	3	4	5	6	7	8	9	10
		λ	a	l	c	o	h	o	l	i	s	m
0	λ	0	1i	2i	3i	4i	5i	6i	7i	8i	9i	10i
1	a	1d	0c	1i	2i	3i	4i	5i	6i	7i	8i	9i
2	l	2d	1d	0c	1i	2i	3i	4i	5i	6i	7i	8i
3	g	3d	2d	1d	1s	2i	3i	4i	5i	6i	7i	8i
4	o	4d	3d	2d	2d	1c	2i	3i	4i	5i	6i	7i
5	r	5d	4d	3d	3d	2d	2s	3i	4i	5i	6i	7i
6	i	6d	5d	4d	4d	3d	3d	3s	4i	4c	5i	6i
7	t	7d	6d	5d	5d	4d	4d	4d	4s	5i	5s	6i
8	h	8d	7d	6d	6d	5d	5c	5s	5d	5s	6i	6s
9	m	9d	8d	7d	7d	6d	6d	6d	6d	6d	6s	6c

Figure 5.4: Completed Array and Traceback

Note that there may be multiple ways of filling an element. For example, $L[8][9]$ could have been reached by an insertion, a deletion, or a substitution. Since all would give a value of 6, it does not matter which one is used.

The pseudo code for the edit distance dynamic programming algorithm is given as Algorithm 8.

```
Read S1 = {x1, x2...xn} and S2 = {y1, y2...ym}
int E[n][m]
for i = 1 to n do
    E[i][0] = i
end for
for i = 1 to m do
    E[0][i] = i
end for
for i = 1 to n do
    for j = 1 to m do
        if xi = yj
            cost = 0
        else
            cost = 1
        end if
        E[i][j] = MIN(E[i-1][j]+1, E[i][j-1]+1, E[i-1][j-1] + cost)
    end for
end for
return E[n][m]
```

Algorithm 8 Edit Distance [39]

Chapter 6

The Optimal Code Problem

An $(n, M, d)_q$ code is defined as a set of M codewords (strings) each having a length n where the characters are taken from an alphabet of size q. In addition each word must be at a minimum distance d from every other word. If the code is an unspecified size, then it is referred to as an $(n, d)_q$ code. Typically one of two different distance measures will be used, Hamming distance or edit distance. The optimal Hamming code problem will be described in detail. The analysis for the edit code problem is exactly the same, with the obvious exception that a different distance measure is used.

The maximum possible number of Hamming codewords of length n from an alphabet of q characters having a minimum distance d is referred to as $A_q(n, d)$. It is a fundamental problem in coding theory to find these maximums for given values of n, q, and d. A code with $M = A_q(n, d)$ is called *optimal*.

If we wish to use the Salmon Algorithm to find optimal codes, it is helpful to convert the problem into a graphical representation. This is easily done, since the problem of finding an optimal code is equivalent to the problem of finding a maximum clique.

A graph $G(V, E)$ is defined by a set of vertices V and a set of edges E. A clique C is defined as a subset of V such that every pair of vertices in this subset is connected by an edge. A maximum clique is the largest possible clique in a graph G. Consider a binary error correcting code of length 3 with a minimum distance of two. This is a $(3, 2)_2$ code, where the subscript refers to the alphabet size. First, construct the compatibility matrix shown in Table 6.1. This matrix will have the value one in any position where the two codewords differ by two or more in Hamming space, and the value zero

if they differ by less than two.

	000	001	010	011	100	101	110	111
000	0	0	0	1	0	1	1	1
001	0	0	1	0	1	0	1	1
010	0	1	0	0	1	1	0	1
011	1	0	0	0	1	1	1	0
100	0	1	1	1	0	0	0	1
101	1	0	1	1	0	0	1	0
110	1	1	0	1	0	1	0	0
111	1	1	1	0	1	0	0	0

Table 6.1: Compatibility Matrix

This codeword compatibility matrix is also the adjacency matrix for the graph given below in Figure 6.1. Thus, we see that the problem of finding the largest number of codewords at a distance 2 or more is the equivalent of finding the maximum clique in the following graph. In this example, the size of the maximum clique is 4, and the vertices in this clique are {000, 011, 101, 110} or {111, 100, 010, 001}. These same values are the codewords in the optimal code.

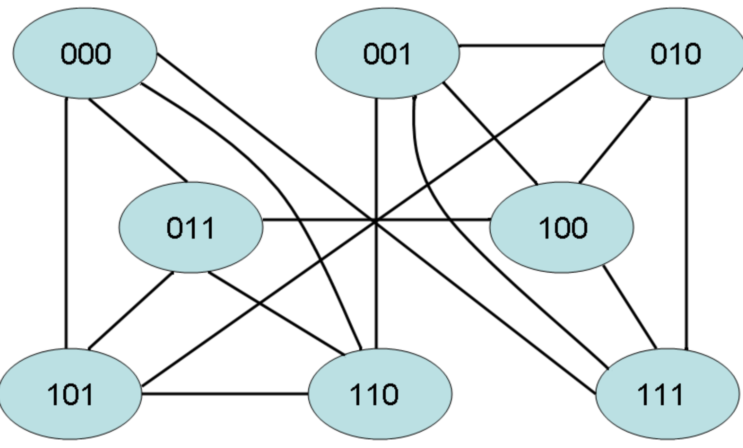

Figure 6.1: Codeword Graph

The maximum clique problem is a known NP-complete problem. In fact, it is one of Karp's original 21 NP-complete problems [22]. This means that exact solutions will be difficult for all but the smallest problem instances. Since codeword graphs can have tens of thousands of vertices and hundreds of millions of edges, it is best to use some type of approximation algorithm for this problem. Previous works have used Genetic Algorithms [16]. This thesis will now demonstrate how the Salmon Algorithm can be configured to find large cliques.

6.1 Salmon Algorithm for Finding Maximum Cliques (Optimal Codes)

The maximum clique problem requires some changes to the algorithm. A solution to the Traveling Salesman's problem is an ordered set of edges. The algorithm adds water to edges to distinguish 'good' edges from 'bad' edges. A solution to the maximum clique problem is a set of vertices. Thus, instead of tracking the water level on each edge, the algorithm tracks the water level on each vertex.

Another subtle but important difference is that the clique cannot be constructed one vertex at a time in a manner analogous to the way the TSP algorithm adds each vertex in succession. Instead, we build a partial clique from the memory vertices first. The remainder of the vertices are then added to the clique using roulette selection on the water levels.

To understand why this is necessary, consider what would happen if we added each vertex in succession, with probability ϕ that the vertex be selected from memory and $1 - \phi$ that it be selected based on water level. Any vertex selected on the basis of water level will probably not be connected (share an edge) with most of the memory vertices. Thus, after selection of a vertex based on water level, most of the vertices in memory will not be candidates for further addition to the clique. The salmon's memory will have been rendered useless.

The pseudo-code for the clique-finding Salmon Algorithm is given as Algorithm 9. It is based on the pseudo-code for constructing cliques using an ant algorithm in [33].

Initialize water levels
for each $salmon_k$ in $k = 1..numSalmon$ **do**
  place random clique in $memory_k$
end for
do
  for each $salmon_k$ in $k = 1..numSalmon$ construct a max clique C_k in graph $G = (V, E)$ as follows:
    for each v_i in $i = 1..numMemory$
      $r = randomDouble(0, 1)$
      if $r < \phi$ **then** $C_k \leftarrow C_k \cup \{v_i\}$
    end for
    for each $v_j \in (V - C_k)$
      if $(v_i, v_j) \in E \, \forall \, v_i \in C_k$ **then** $Candidates \leftarrow Candidates \cup v_j$
    end for
    while $Candidates$ is non-empty
      Choose a vertex $v_i \in Candidates$ with probability $p(v_i)$
      $C_k \leftarrow C_k \cup \{v_i\}$
      $Candidates \leftarrow Candidates \cap \{v_j | (v_i, v_j) \in E\}$
    end while
  end for
  update water levels
  produce Children
  if best salmon clique size $>$ $elite.cliquesize$ **then** update elite
until maximum number of cycles reached
return largest constructed clique found

$$\text{Where } p(v_i) = \frac{(waterLevel_i)^\alpha}{\displaystyle\sum_{v_j \in Candidates} (waterLevel_j)^\alpha}$$

Algorithm 9 Salmon Algorithm for Maximum Cliques (Codes)

6.1.1 Memory Optimization

Looking at Algorithm 8, we can see that every time a vertex is evaluated for possible addition to the candidates list the algorithm must compute the distance between the potential candidate vertex and all the vertices currently in the clique. Because of the large number of times this must be done, and the fact that the edit distance algorithm is $O(n^2)$, it is best to pre-compute and store the edit distances between all possible codewords (vertices) to reduce run times. This may not be true with Hamming distance which can be calculated with an $O(n)$ algorithm. In fact Hamming distance calculations can be optimized further by the use of bitwise exclusive-or statements. (Note: Strictly speaking the edit distance algorithm is $O(mn)$ where m and n are the lengths of the source and destination words. However, in the codes we will be constructing all words have the same length i.e. the number of insertions equals the number of deletions, so the algorithm in this case is $O(n^2)$.)

Because the distance between any two words S_1 and S_2 equals the distance between S_2 and S_1 the adjacency matrix is symmetrical. We can use this fact to cut our memory requirements in half. Note also that since the distance from a word to itself is zero, we can eliminate the diagonal of the adjacency matrix. Finally, be aware that because a byte is the smallest addressable unit of memory, C++ only stores one boolean value per byte. If we represent the adjacency matrix with a boolean array, only one bit of every byte will be utilized. Memory requirements will be 8 times what a simple counting of the number of entries in the adjacency matrix would predict. To maximize memory utilization it is necessary to use bit masking to store information in all 8 bits per byte and to recover individual bit information.

Even with memory optimization, for large values of n it will not be possible to store the adjacency matrix because memory requirements increase exponentially with word length. With a binary alphabet the adjacency matrix quadruples in size every time the word length increases by one character. Using a quaternary alphabet, storage requirements increase 16 fold for each character added. For a length 8 quaternary code the adjacency matrix with memory optimization requires about 256 MB of storage. A length 9 code would require 4 GB.

6.2 Conway's Lexicode Algorithm

One algorithm that is often used for constructing large codes is Conway's Lexicode Algorithm [7]. This is a simple greedy algorithm that does not usually produce an optimum code, but frequently produces a good code in a relatively short time. Conway's Lexicode Algorithm is given as Algorithm 10.

Given an alphabet of size q, minimum distance d, and word length n
$dist(x, y)$ returns the distance between x and y.
S is the ordered set of all q^n words
$R = \{\}$
for each $s \in S$ in lexicographic order **do**
  if $dist(r, s) \leq d \; \forall r \in R$
    $R = R \cup s$
  end if
end for
return R

Algorithm 10 Conway's Lexicode Algorithm

The algorithm can also be run with a seed, which means that the set R is not initially null, but instead contains a small, incomplete code. Since every word in the seed blocks any other word within distance d from being included in the code, even a small seed can have a large effect on the composition of the code.

6.3 Sphere Packing Bound

The *Sphere Packing Bound* [20] is used to set an upper bound on the size of a code, whereas the various search techniques described previously set a lower bound. Recall from Chapter 4 that a code can correct up to $t = \lfloor (d-1)/2 \rfloor$ errors. For Hamming distance codes the maximum number of words in each codeword sphere is given by

$$1 + \binom{n}{1}(q-1)^1 + ... + \binom{n}{t}(q-1)^t$$

where the first term is the codeword itself, the second term is the number of possible words that can have 1 error in an alphabet of q symbols, and the last term is the number of words than can have t errors. The total number of words is q^n. Thus, the maximum number of codewords is given by

$$A_q(n,d) \leq \frac{q^n}{\displaystyle\sum_{i=0}^{t} \binom{n}{i} (q-1)^i}$$

If a code is perfect, it will achieve the Sphere Packing Bound, i.e. the number of codewords will be equal to the above upper bound.

Chapter 7

Covering Codes

Let W be the set of all q^n words of length n that can be formed using an alphabet of size q. A *covering code* C of radius d is a subset of W such that for all words in W there exists at least one word in C at a Hamming distance of d or less. Stated in mathematical terms, $C \subseteq W$ such that $\forall w \in W, \exists c \in C$ with $dist(w, c) \leq d$.

For example, if $n = 2$ and $q = 3$ then $W = \{00, 01, 02, 10, 11, 12, 20, 21, 22\}$. One possible cover with $d = 1$ is $\{00, 01, 02\}$.

Recall from Chapter 4 that an error correcting code is said to be perfect if every word is contained uniquely in one codeword sphere. A perfect $(n, r)_q$ covering code has the same definition using spheres of radius r. However, a covering code that is not perfect will differ from an error correcting code that shares the same characteristic.

An error correcting code that is not perfect will have words that exist outside of all codeword spheres. Thus, the number of codewords will be less than the number in a perfect code. By definition, a covering code cannot have any words outside of all codeword spheres. Instead, a covering code that is not perfect will have words that appear in multiple spheres. This code will therefore be larger than a perfect code.

Exact bounds on covering codes are known for only a few of the smaller cases plus $n = 13$, which is a perfect code. Known limits for codes with $q = 3$ and $d = 1$ are shown in Table 7.1. For a more complete list see the on line site, Tables on Bounds for Covering Codes [23].

Length	**Lower**	**Upper**
1	1	1
2	3	3
3	5	5
4	9	9
5	27	27
6	71	73
7	156	186
8	402	486

Table 7.1: Limits on Covering Codes with d = 1 and q = 3

The minimum size for a cover using an alphabet of size q, word length n and radius d is known as $K_q(n, d)$.

7.1 Football Pool Problem

Some believe the football pool problem is one of the most famous problems in coding theory [17]. Consider a hypothetical football (or, in North American parlance, soccer) pool where the objective is to correctly predict the outcome of n games as either a win, loss, or tie for the home team, with the restriction that the winner of the pool can make at most one wrong prediction. The football pool problem asks this question: What is the smallest number of tickets that must be purchased to ensure that the holder of those tickets will win the pool regardless of the outcome of the games? Since there are three possible outcomes for each game, this makes $q = 3$. Since the winner can pick at most one wrong, this makes $d = 1$. The football pool problem for n games is the equivalent of finding the smallest possible covering code of length n with $d = 1$ and $q = 3$.

7.1.1 Salmon Algorithm for the Football Pool Problem

The pseudo code for the Salmon Algorithm for the minimum covering code problem is given as Algorithm 11.

W is the set of all q^n words of length n and alphabet size q.
Given $S \subseteq W$ the function $coveredBy(S)$ returns the set of all words covered by S

Initialize water levels
for each $salmon_k$ in $k = 1..numSalmon$ **do**
  place random cover in $memory_k$
end for
do
  for each $salmon_k$ in $k = 1..numSalmon$ construct a min cover C_k as follows:
    $C_k = \{\}$
    for each w_i in $i = 1..numMemory$
      $r = randomDouble(0, 1)$
      if $r < \phi$ **then** $C_k \leftarrow C_k \cup \{w_i\}$
    end for
    while $coveredBy(C_k) \neq W$
      $Candidates = \{\}$
      for each $w_j \in (W - C_k)$
        if $\exists w_k \in (W - coveredBy(C_k)$ and $w_k \in coveredBy(w_j)$
        then $Candidates \leftarrow Candidates \cup w_j$
      end for
      Choose a word $w_i \in Candidates$ with probability p_i
      $C_k \leftarrow C_k \cup \{w_i\}$
    end while
  end for
  update water levels
  produce Children
  if best salmon cover size $<$ $elite.cliquesize$ **then** update elite
until maximum number of cycles reached
return smallest constructed cover found

The probability p_i of selecting a given word w_i is given by:

$$p_i = \frac{(waterLevel_i)^\alpha}{\sum_{w_j \in Candidates} (waterLevel_j)^\alpha}$$

Algorithm 11 Salmon Algorithm for Minimum Covering Codes

Until this point we have viewed the Salmon Algorithm as a population of salmon swimming from vertex to vertex in a graph. This may help with visualization of some problems, but it is not necessary for a problem to be in graphical format for the Salmon Algorithm to be used. It is only necessary to have discrete components on which we can deposit water. In the football pool problem, these components are the words in W.

Since we are attempting to build the smallest possible cover, this is a minimization problem. However, it is possible to view it as a maximization problem. To do this we do not fix the number of words covered at q^n as in Algorithm 10. Instead, we fix the number of words in the cover at an amount slightly greater than the best known cover size, then attempt to maximize the number of words covered. If a cover for all q^n words is found, then the number of words in the cover is reduced by one, and the process repeated. This was the strategy used by Van Laarhoven et al.[36]. Both versions of the algorithm are used in this thesis.

Chapter 8

Literature Review

This chapter will give a brief review of the research that has been done on maximum edit and Hamming codes, and minimum covering codes.

8.1 Edit Codes

The study of edit codes was pioneered by Russian scientist Vladimir Levenshtein in the mid 1960's. As indicated in Chapter 5, edit distance is often referred to as Levenshtein distance. The dynamic programming algorithm for edit distance presented in Chapter 5 is usually attributed to Levenshtein[25], although many other researchers discovered the same algorithm independently[34].

In 2002 Ashlock et al.[1] used a genetic algorithm in conjunction with Conway's lexicode algorithm to find optimal edit codes. Each chromosome consisted of a small seed for the lexicode algorithm. The fitness of a chromosome was determined by the size of the code that was produced when the lexicode algorithm was run with the given seed. This process was termed a greedy closure evolutionary algorithm, since the greedy algorithm (Conway's) was used to finish or close the code. Crossover consisted of swapping a given word in the two parents with a probability of 50%. Mutation consisted of taking one of the words in the seed and replacing it with a new word. Both crossover and mutation had the potential to create seeds that violated the minimum distance requirement. Such an illegal chromosome was given a fitness of zero and removed from the population by selection.

The problem with this algorithm is that it must step through all possible

words every time the fitness of a chromosome is evaluated. Since the number of possible words is exponential in the length of the code words, run times also increase exponentially. In 2006 Houghten et al.[19] provided a faster method for finding codes. In this algorithm the completed codes are stored as chromosomes. A binary variation operator was used to produce one new code from two parents. The two parent codes were shuffled together and one random word was added. Conway's algorithm was then used to turn this list of words into a code. This method produced codes that were smaller than those produced in [1]. However, run times were much shorter. This allowed codes using longer codewords to be created.

In 2009 Ashlock et al.[3] improved on the work in [1] by demonstrating that crossover was detrimental to the process of finding large codes. The reason crossover was ineffective can be explained with the concept of infertility. If two parents have a high probability of producing offspring with a fitness good enough for them to survive, the parents are said to be fertile. If the fitness of the children is low, meaning they will probably not survive, then the parents are said to be infertile. The problem with the crossover operator used in [1] is that it frequently produced children that violated the minimum distance requirement and hence had a fitness of zero. The crossover free algorithm used in [3] is a type of evolutionary strategy (ES).

In [3] the authors also demonstrated that the optimum seed size is roughly three. The larger the seed size, the more likely it is that two of the words will violate the minimum distance requirement.

All of the previous research into optimal edit codes has been restricted to codes comprised of words of equal length, i.e. the number of insertions equals the number of deletions. All edit codes in this thesis will be similarly restricted.

8.2 Covering Codes

Recall that the most complete table of covering codes can be found on line at [23].

The $K_3(4,1)$ and $K_3(13,1)$ codes are perfect[37]. Thus, using the Sphere Packing Bound formula given in Chapter 6, and substituting the value 1 for the radius, we see that $K_3(4,1) = 9$ and $K_3(13,1) = 3^{10}$. By successively extending the $K_3(4,1)$ result it is easily shown that $K_3(n,1) \leq 3^{n-2}$ for $5 \leq n \leq 12$.

Simulated annealing was first applied to the covering codes problem by Wille [40] who in 1987 established a new upper bound of 74 for the $K_3(6,1)$ problem. The current best known upper bounds for the $K_3(6,1)$, $K_3(7,1)$, and $K_3(8,1)$ cover codes were found by Van Laarhoven et al.[36] in 1989. These authors used simulated annealing directly to find the length 6 and 7 codes, and simulated annealing in conjunction with the Blokhuis and Lam [4] matrix construction method for the length 8 code.

In 1997 Patric Östergård [27] was able to improve the then best known value for the $K_3(9,1)$ problem to 1341 using a tabu search.

8.3 Hamming Codes

Andries Brouwer[5] maintains tables of best known Hamming distance codes on line, along with references for the papers in which these bounds were established.

The *Gilbert Bound*[14] uses a construction similar to the Sphere Packing Bound to set a lower bound. The Gilbert Bound is

$$A_q(n,d) \geq \frac{q^n}{\displaystyle\sum_{i=0}^{d-1} \binom{n}{i}(q-1)^i}$$

Begin the proof of the Gilbert Bound by observing that the covering radius r of a code C with $A_q(n,d)$ codewords has $r \leq d-1$. Proof of this observation is by contradiction. Assume the code C has a covering radius of d or more. If this is the case, then there is a word w in q^n at a distance $\geq d$ from every codeword. This means $C \cup \{w\}$ is also a code with minimum distance d, and this code has one more codeword than C. But this is not possible, since we assumed C was optimal.

The minimum number of codewords is the total number of words q^n divided by the maximum possible number in each sphere, which gives us the Gilbert Bound.

Another lower bound is the *Varshamov Bound*. This bound is similar to the Gilbert Bound, in fact, asymptotically they are the same [20]. For this reason the two bounds are sometimes treated as one, and referred to as the *Gilbert-Varshamov Bound*. The only other analytical lower bound is the *Algebraic Geometry Bound*[38]. The problem with both of these bounds is

they are very low quality[35].

On the other hand, a number of analytical upper bounds are known. Some are the *Plotkin Bound*, the *Johnson Bound*, the *Singleton Bound*, the *Elias Bound*, the *Linear Programming Bound*, the *Griesmer Bound*, and the Sphere Packing Bound, which was discussed in Chapter 6. Derivations for all of these bounds can be found in [20].

Because only two analytical lower bounds are known to exist, and these are poor quality, lower bounds are usually established with some type of search. Haas and Houghten [16] used a variety of methods to search for large Hamming codes. These authors tested beam search, hill climbers, tabu search, simulated annealing, Conway's lexicode, and two variations of genetic algorithms in conjunction with the lexicode algorithm. They determined the most successful strategy was to use a GA to produce an incomplete code, then finish it with Conway's lexicode algorithm.

Chapter 9

Experimental Results

This chapter gives the experimental results for optimal edit, Hamming, and covering codes. Most of the parameter optimization was performed on edit codes first, so the edit code section is more detailed. The initial work on Hamming codes used the same parameter values that were found to perform well on edit codes. Since the problems differ only in their distance metric, it was thought the search spaces might be similar. Further optimization on Hamming codes was only attempted if the initial results were poor.

As seen in Chapter 8, previous researchers obtained the best results for covering codes using Simulated Annealing and Tabu Search, while good results for edit and Hamming codes were achieved with GA's or variations of Conway's lexicode algorithm. Based on the fact that these problems have demonstrated the need for completely different search strategies, we might expect the covering code parameter values to be significantly different from those used in the edit and Hamming code problems.

All experiments were conducted on a 30 core cluster made up of a mixture of 4 and 6 core AMD and Pentium chips running at between 2.6 and 3.2 Ghz. All experiments used a total of 30 runs for each test case.

9.1 Optimal Edit Codes

The following is a list of parameters that this thesis will attempt to optimize:

1. Initial Water Level Multiplier IM

2. Population Size
3. Memory Probability ϕ
4. Reproduction Fraction σ
5. Roulette Selection Exponent α
6. Number of Elite

A complete examination involving only 5 distinct values for each of the 6 parameters would result in $5^6 = 15625$ test cases. Of course, each test case would need to be run a minimum of 30 times for statistical significance. This process would then need to be repeated for each of the six data sets that were examined. This is simply not feasible. In order to reduce the amount of computing to a reasonable level we need to make some assumptions about parameter independence and the shape of the curves.

We will assume that the curves are unimodal. If we have a clique size vs α curve similar to Figure 9.1, then we assume that the curve does not tend upwards again at α values less than .5 or greater than 1. Testing on the $(6,3)_4$ code with $\alpha = 0$ to $\alpha = 1.0$ in increments of .1 revealed only one optimum, so this assumption is fairly reasonable.

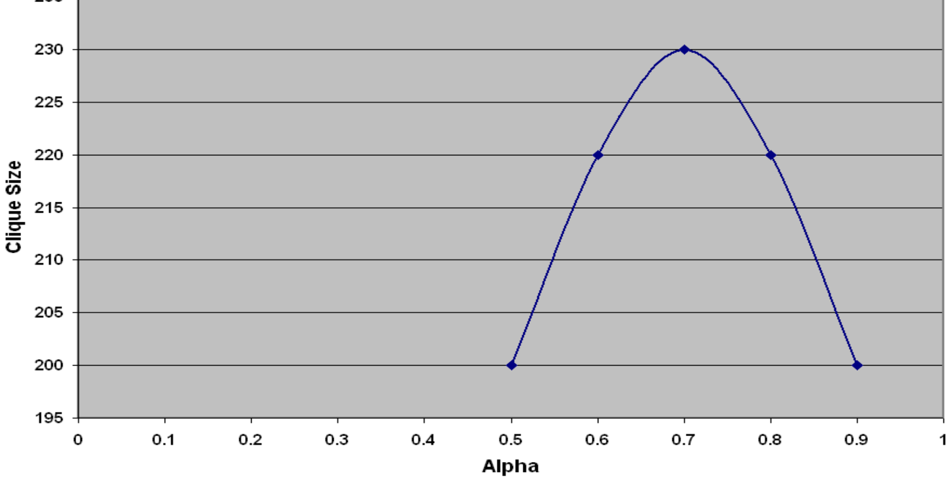

Figure 9.1: Unimodal Curve

We will assume that the optimum IM value is constant for all cases. The algorithm attempts to compensate for the fact that larger cliques deposit larger quantities of water by setting an initial water level equal to $BK * IM$, where BK is the largest clique known to exist for this problem.

We will assume that the optimum number of elite is only related to the population size and not to any other parameters. We will select a population size that gives good results without taking an excessive amount of time to run. This population will be used for all the data sets. Finally, preliminary testing showed that there was some degree of dependence between ϕ, α, and σ, so we will attempt to optimize these parameters together.

Most of the parameter adjustments were made on the (6,3) code first. This is a well studied case. An application for this code is described in [29]. Codewords were used to identify DNA sources in an expressed sequence tag project. The codes used were length 6 and distance 3, which allowed for single error correction. Additionally, run times for this code are significantly shorter than the length 7 and length 8 codes. Using a population of 100 the time to complete a run of the (6,3) code was about 1400 seconds, while the (7,3) code required 12 000 seconds, and the (8,3) code needed 3 days.

9.1.1 Optimum Initial Water Level Multiplier (IM)

The first attempts at setting the initial water level [2] involved a method of allocating more water to vertices with a higher degree in the codeword graph. Vertices with the highest degree received an amount of initial water equal to $BK * 2$. Vertices with the lowest degree received an initial water level equal to BK. (Recall that BK is the best known clique size for a given data set.) Vertices between these two extremes received a water level proportional to their degree. The intention was to bias the initial water level towards those vertices that had a greater chance of being members of a large clique.

This strategy was found to be non-productive. The algorithm produced superior results by simply setting the initial water level to a constant. Figures 9.2 and 9.3 show the results on the (6,3) data. Here, "complex" refers to the biased initial water level while "simple" is an initial water level equal to BK. As can be seen, the simple initial water level was no better than the complex value on average clique sizes, but it more consistently found the largest (at the time) size of 112. The other parameters used to produce these graphs were $\phi = 0.7$, $\sigma = 0.5$, $population = 400$, and $number\,of\,elite = 1$.

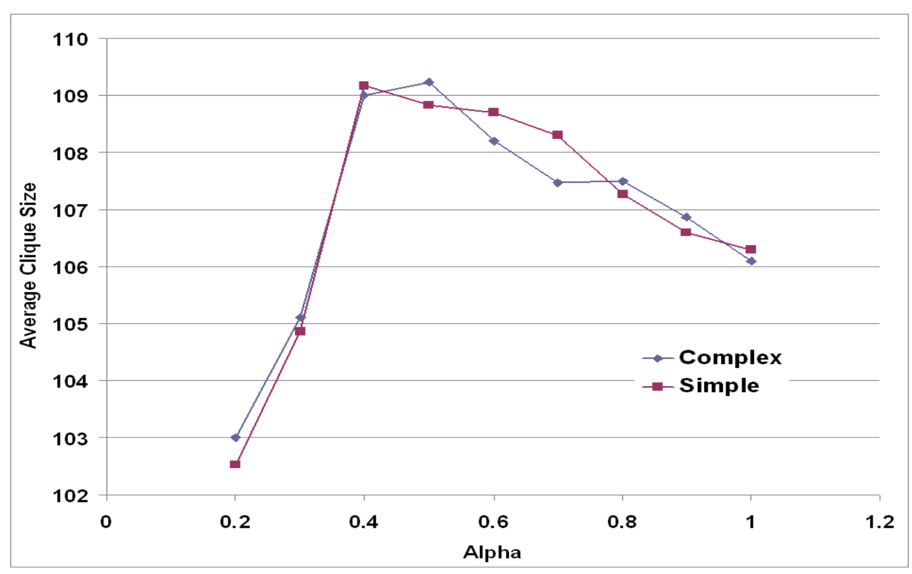

Figure 9.2: Average Clique Size vs α for $(6, 3)$ Codes

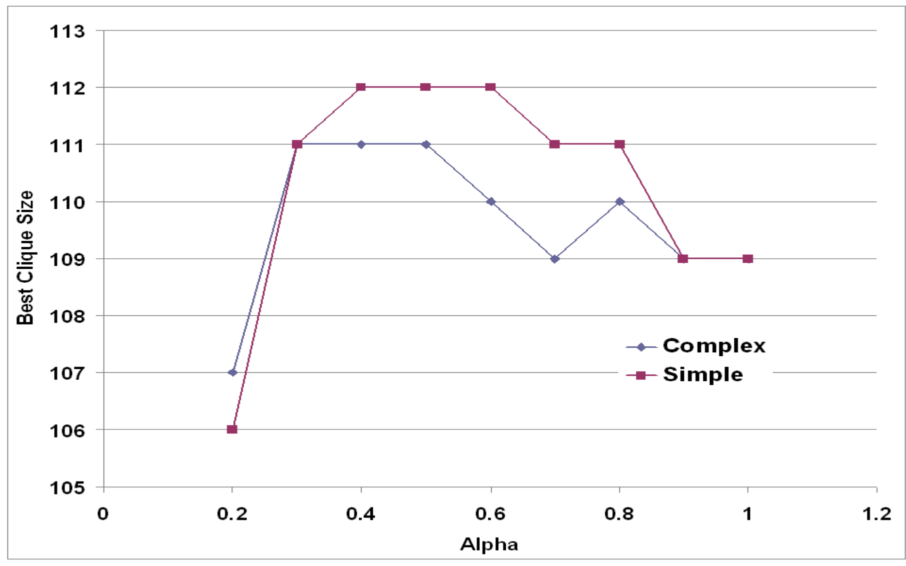

Figure 9.3: Best Clique Size vs α for $(6, 3)$ Codes

Alpha	**.3**	**.4**	**.5**	**.6**
$IM = 1$	104.87	109.17	108.83	108.70
$IM = 10$	105.23	109.43	109.17	108.50
$IM = 100$	105.03	109.07	108.80	108.23
$IM = 1000$	104.30	107.70	108.63	108.47

Table 9.1: Average clique size vs α for various IM

Alpha	**.3**	**.4**	**.5**	**.6**
$IM = 1$	111	112	112	112
$IM = 10$	109	112	111	111
$IM = 100$	109	111	111	110
$IM = 1000$	108	111	111	111

Table 9.2: Best clique size vs α for various IM

Notice that although the complex initial water level produced an average clique size that was slightly higher than the simple initial level, it never found a clique of size 112. We cannot optimize only for best average clique size, but need to also look at the best value obtained for a given set of parameter values. Since our final goal is to produce the largest possible code, it can be argued that the highest value obtained is more important than the average value.

Having decided upon using an evenly distributed initial water level, we must now find the optimum initial value. Tables 9.1 and 9.2 show the average and best clique sizes obtained for $IM = 1$ to $IM = 1000$ over a range of α values for (6,3) codes. The other parameters used were $\phi = 0.7$, $\sigma = 0.5$, $population = 400$ and $number\,of\,elite = 1$.

The results suggest that an optimal IM value lies in the range of 1 to 10. Running the algorithm with an IM value of 100 or higher did not produce a single instance of a clique size of 112, the best result we obtained for this data set early in the testing. Using an IM value of 1 or 10 produced six runs with the optimum result. Higher IM values also cause the algorithm to take longer to converge. We used an IM value of 5 for most of the experiments on all data sets.

Population	Optimum α
100	.30
200	.33
400	.35 to .36

Table 9.3: α vs Population for (7,3) edit codes

9.1.2 Optimum Population Size

Population size was found to have only a slight effect on the quality of the results. Using the (7,3) data, a population of 100 returned a high average of 347.6 and a best of 352. A population of 400 gave a high average of 348.6 and a best of 352. Thus, although higher populations yield slightly better clique sizes, the gain is not worth the extra run time. Better results could be obtained with 4 runs of population 100 over one run with population 400. Populations of less than 100 were not tested, since all parameter sets up to (8,3) ran within a reasonable amount of time with this population.

Optimum α values were found to be sensitive to population size. See Table 9.3 for this result.

9.1.3 Tuning the α, ϕ and σ Values

The parameters ϕ and α both have similar effects on the algorithm. A high α value causes the algorithm to bias its roulette selection to known good vertices which results in fast convergence. A high ϕ value means each salmon is building its clique based primarily on the vertices in its parent's clique and not exploring significantly for new possibilities. This also tends to cause rapid convergence. For the algorithm to produce good results a high α value must be offset by a low ϕ value. The reverse is also true. However, there are limits beyond which it is not possible to compensate. Using the (6,3) data we found that good results could be obtained with either $\phi = 0.7$ or $\phi = 0.8$ by adjusting the α value up or down accordingly. However, if ϕ was set too low ($\phi = 0.6$) or too high ($\phi = 0.9$) then the results deteriorated regardless of the α value selected. This characteristic is shown in Figs. 9.4 and 9.5. Other parameter values used to produce these graphs were $population = 400$, $IM = 5$ and $number\,of\,elite = 1$.

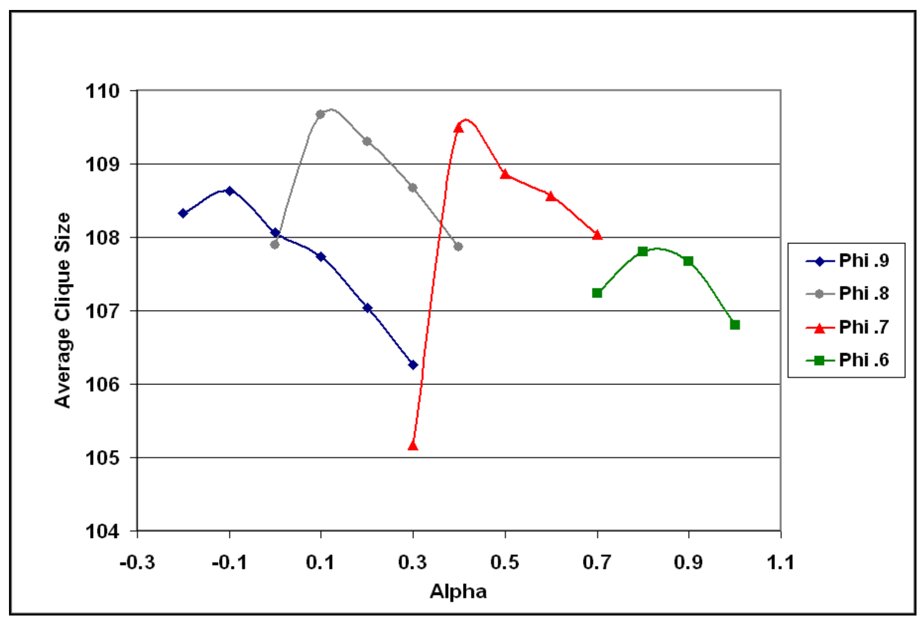

Figure 9.4: $\sigma = .5$ Average Clique Size vs α

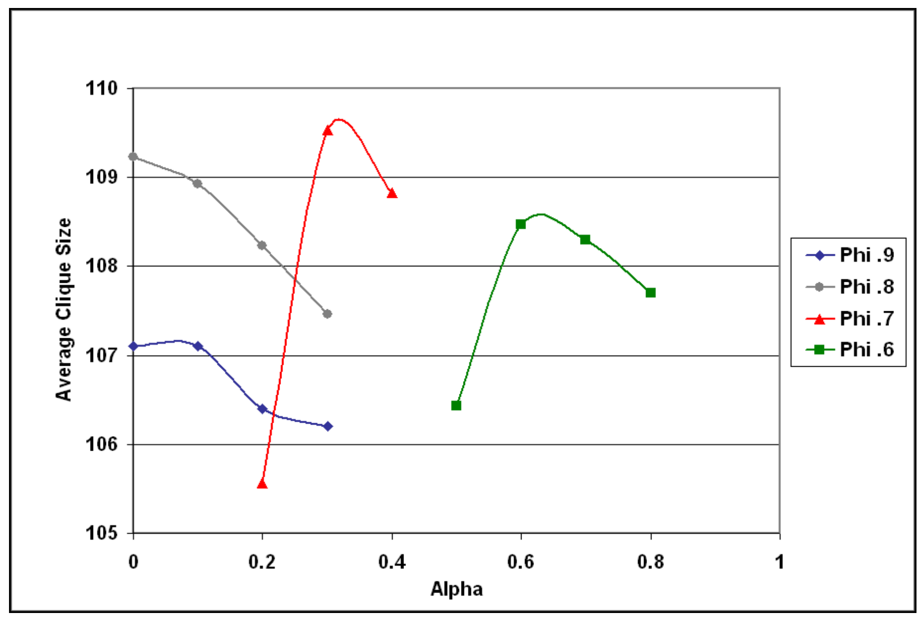

Figure 9.5: $\sigma = .333$ Average Clique Size vs α

Viewing these graphs it is not immediately clear whether the $\sigma = 0.5$ or $\sigma = 0.333$ results are superior. In both groups the $\phi = 0.7$ and $\phi = 0.8$ values have best averages of over 109. However, with $\sigma = 0.333$ there was only one run that resulted in a clique of size 112, while with $\sigma = 0.5$ there were six runs with this value. For this reason, we chose to use $\sigma = 0.5$ for further experiments.

Although good results can be obtained by using ϕ values of either 0.8 or 0.7, it is clear from the $\sigma = 0.5$ graph that the results from using 0.8 are slightly better. The same optimum value for ϕ was found when similar measurements were performed on both the (7,3) and (6,4) codes. Note that the Salmon Algorithm was previously demonstrated to work well when applied to the problem of finding codes of lengths 5–8 and minimum edit distance of 3 or 4 [2]. The results from current tests suggest that $\phi = 0.8$ will produce good (optimal or near optimal) results for all codes within this previously demonstrated viable range.

9.1.4 Optimum Number of Elite

The optimum number of elite was found to be zero. Results for three data sets are given in Table 9.4. The results with an elite salmon were obtained with a population of 400 and one elite, which produced 2 elite children. The results with no elite were produced with a population of 100. In spite of the smaller population, the results with no elite were consistently superior to those achieved with an elite salmon.

Data Set	1 Elite	No Elite
(6,3)	112	114
(7,3)	350	356
(8,4)	173	176

Table 9.4: Elite vs. No Elite

9.1.5 Optimum Parameter Values Summary

The final parameter values selected are $population = 100$, $IM = 5$, $number\ of\ elite = 0$, $\sigma = 0.5$, and $\phi = 0.8$. The α value varies with the data set and will be determined with further experimentation.

9.2 Optimal Edit Code Results

Figure 9.6 shows average clique size versus α for the (6,4) data set. Notice that the best average comes at an α value of -.9. Initial attempts to produce large (6,4) codes were disappointing. The best known size for such a code is 28, and the Salmon Algorithm at first struggled to produce a code of 25 words. It was not until the algorithm was tested with negative α values that it was able to match the best known value for this code.

This was an unexpected and counterintuitive result. The Salmon Algorithm is based on the idea that components that have been part of previous good solutions can be combined to make better solutions. However, in the (6,4) code, instead of showing a preference for vertices that have been part of previous good solutions, α needs to be biased to show a preference for vertices that have previously been part of poor solutions. There are strong local optima at sizes 24 and 25 for this case. Perhaps the algorithm needs to be oriented towards exploration to find its way past such situations.

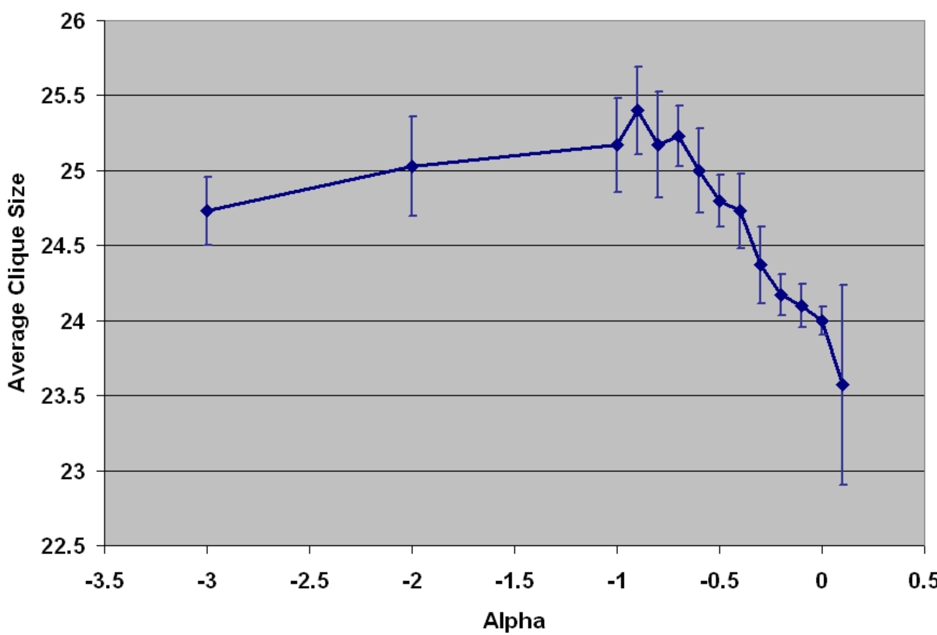

Figure 9.6: (6,4) Average Clique Size vs α with 95% CI

More typically, the clique size vs α results were distributed like the (8,4) data shown in Figure 9.7. There are two curious points regarding this graph. First, the best results (largest cliques) were not obtained where the graph reaches its maximum value at $\alpha = .37$. Instead, they came at $\alpha = .35$. In the area around the inflection point the results have a bimodal distribution, with the lower group clustered around 153 and the upper group around 173. The bimodal distribution is why no 95% confidence interval is shown. The distribution of the results from $\alpha = .30$ to $\alpha = .37$ is shown in Figures 9.8 to 9.13.

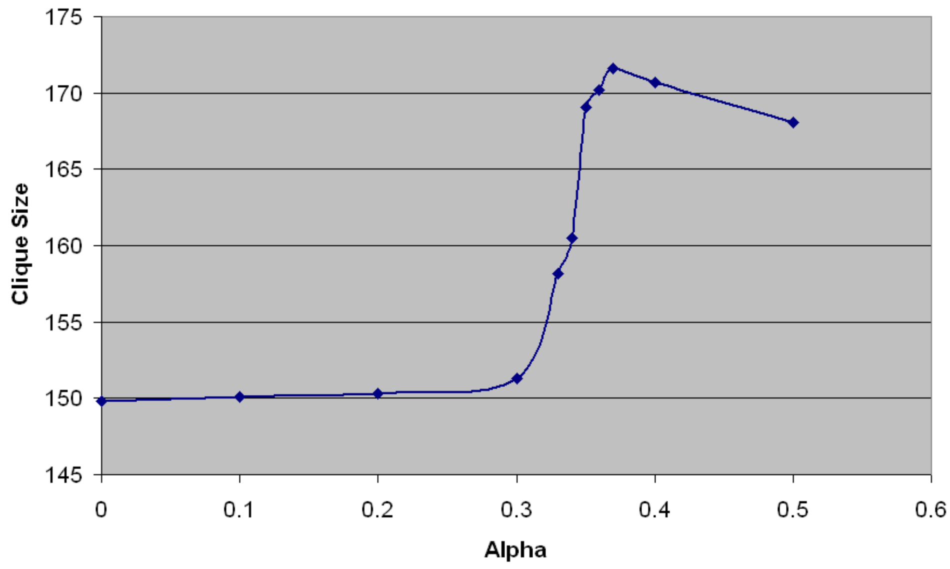

Figure 9.7: (8,4) Average Clique Size vs α

The second point is that the algorithm is very sensitive to small changes in the α value for this problem. The largest cliques were obtained with $\alpha = .35$. The distribution of results at this value is shown in Figure 9.12. Seven (of 30) runs found cliques of size 174, four runs resulted in 175, and 2 reached 176. For $\alpha = .36$, shown in Figure 9.13, there was only one run of 174, none of 175 and none of 176. Thus an increase of only .01 in α value resulted in a significant deterioration in the quality of the results.

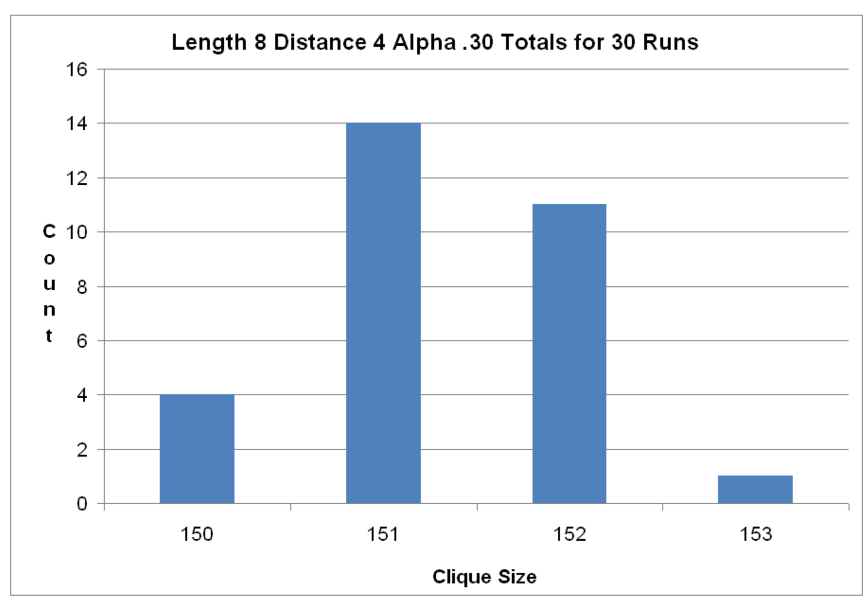

Figure 9.8: Alpha .30 Results Distribution

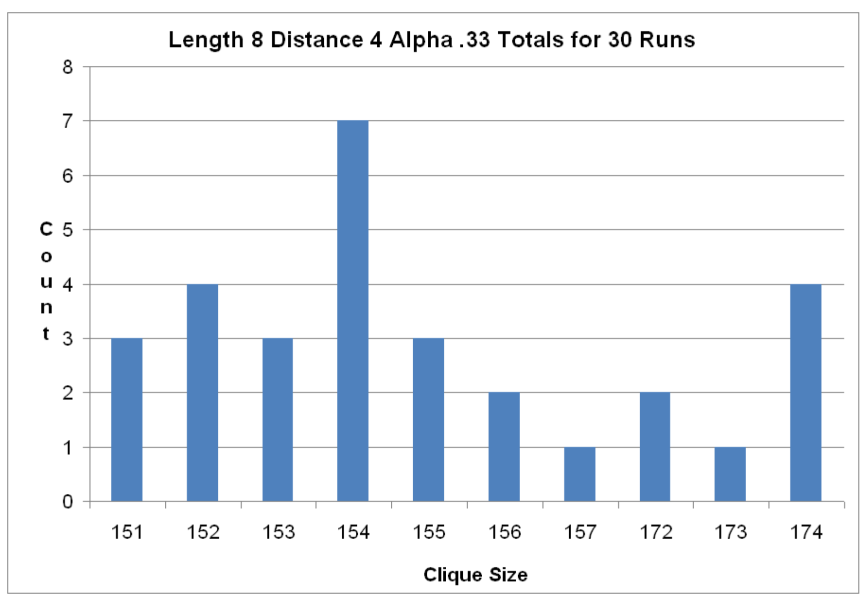

Figure 9.9: Alpha .33 Results Distribution

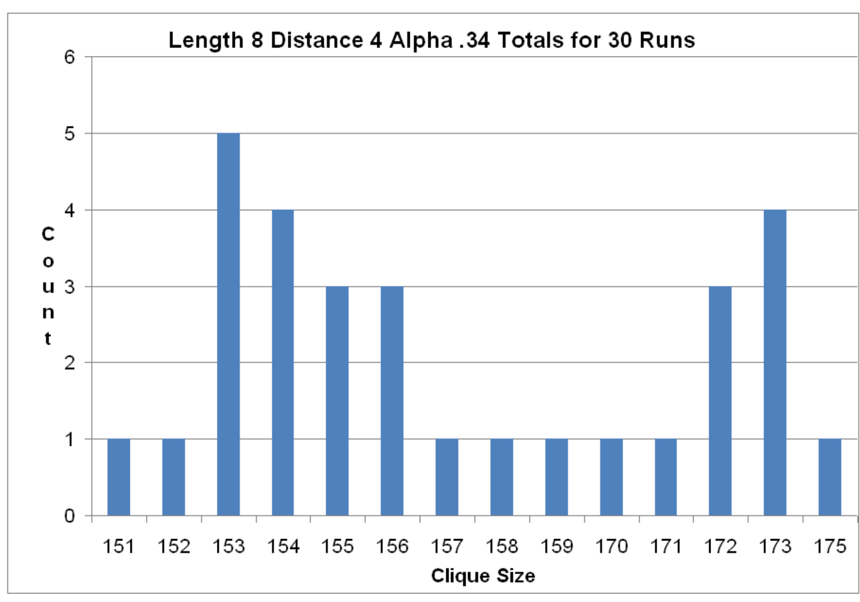

Figure 9.10: Alpha .34 Results Distribution

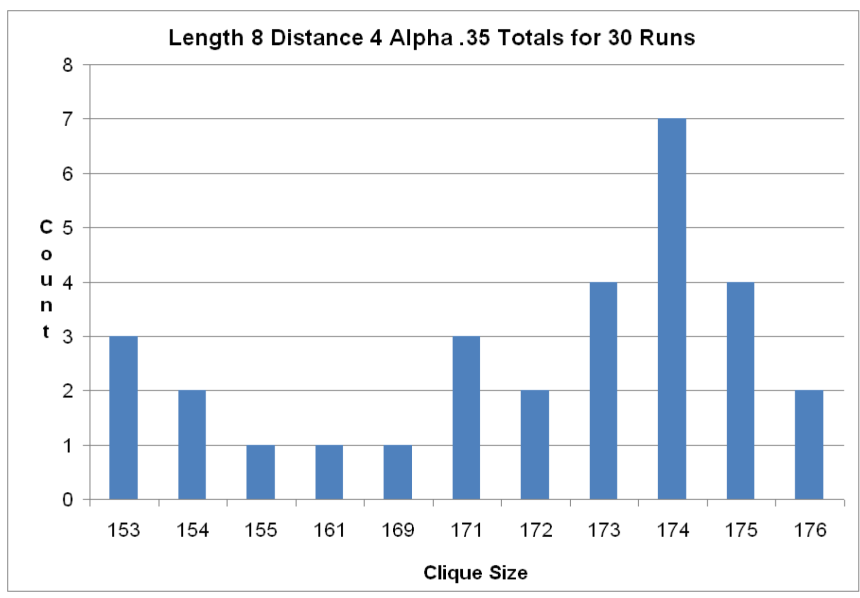

Figure 9.11: Alpha .35 Results Distribution

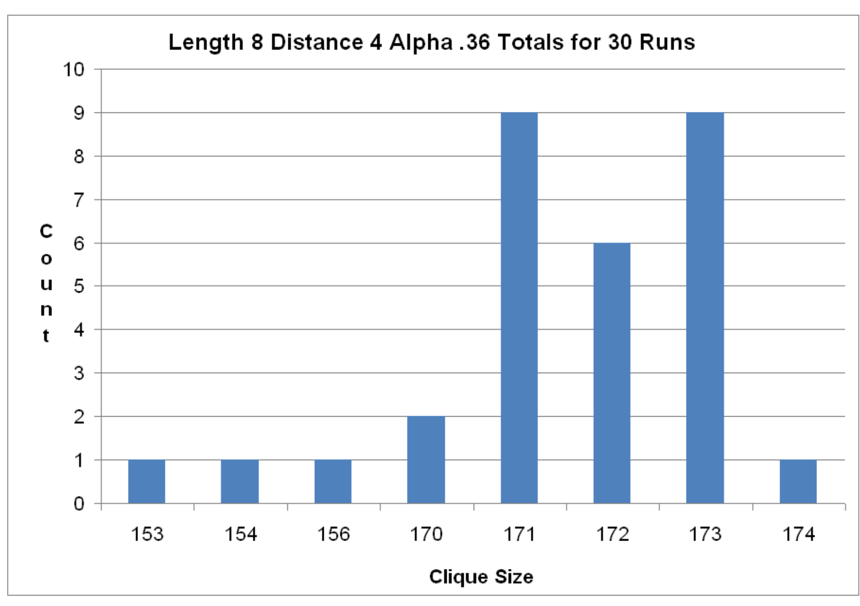

Figure 9.12: Alpha .36 Results Distribution

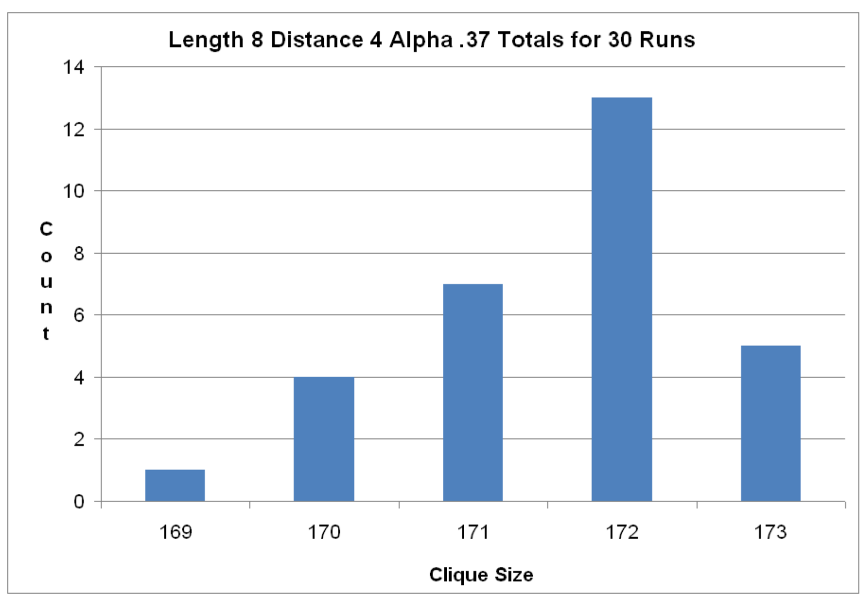

Figure 9.13: Alpha .37 Results Distribution

The maximum number of edit codewords of length n from an alphabet of q characters having a minimum distance d is referred to as $E_q(n, d)$. The Salmon Algorithm was able to obtain improvements in the size of five of the codes. These results are given in Table 9.5.

Code	**Previous Best**	**New Best**
$E_4(6, 3)$	108	114
$E_4(7, 3)$	329	356
$E_4(8, 3)$	1025	1132
$E_4(6, 4)$	28	28
$E_4(7, 4)$	63	65
$E_4(8, 4)$	164	176

Table 9.5: Improvements to Edit Codes

The algorithm does not scale well for this problem because both memory and time requirements increase exponentially with word length. An increase in word length of one character causes a 16 fold increase in the size of the compatibility matrix, which is normally stored in memory. It was also observed that run time increased by roughly a factor of 10 for each increase of one in word length. Of course, it would be possible to compute edit distances 'on the fly' and eliminate the memory constraints, but this would further exacerbate the run time problem. For these reasons it is unlikely the algorithm will be useful for anything larger than the $E_4(9, 3)$. Even this code would probably require several weeks of run time, and use most of the memory on a standard desktop computer.

It was also found that the algorithm needs to be tuned for every different code. Each code has a different search space and parameter values need to be adjusted for best results. This consisted primarily of altering the α value in the roulette selection formula.

9.2.1 GC Content

The GC content of a DNA sequence is defined as the fraction of the bases which are either C or G. The GC content of a DNA molecule determines its melting point, and for this reason it is frequently desirable to have all codewords with the same, or close to the same, GC content. Moreover, it

is common to have the GC content set as closely as possible to 50%. For a more complete explanation of GC content see Frutos et al.[13].

Thus, while the codes in Table 9.5 may be interesting from a mathematical perspective, they are of limited use for biological applications since their GC content can range from zero to one hundred percent. One strategy for producing codes with constant GC content would be to simply take the codes created above and remove any words that violate the required GC content. While this will produce a valid code, there is no guarantee that the code created this way will be as large as possible.

A superior strategy is to remove any words without the required GC content at the very start of the Salmon Algorithm, when the codeword adjacency matrix is created. This was the technique used to produce the results given in Table 9.6. Here, GC refers to the number of letters in the word that are either C or G.

Code	**GC**	**Size**
$E_4(6,3)$	2	63
$E_4(6,3)$	3	81
$E_4(6,3)$	2 or 3 or 4	102
$E_4(6,4)$	2	18
$E_4(6,4)$	3	22
$E_4(6,4)$	2 or 3 or 4	28
$E_4(7,3)$	3	220
$E_4(7,3)$	3 or 4	272
$E_4(7,4)$	3	49
$E_4(7,4)$	3 or 4	55
$E_4(8,3)$	3	585
$E_4(8,3)$	4	704
$E_4(8,3)$	3 or 4 or 5	952
$E_4(8,4)$	3	110
$E_4(8,4)$	4	132
$E_4(8,4)$	3 or 4 or 5	159

Table 9.6: Fixed GC Content

Words with a GC content of less than one third were not considered. Equivalent codes were also not considered. For example $E_4(6,3)$ GC 2 is

given, but not $E_4(6,3)$ GC 4, because these two codes are equivalent. Recall that an equivalent edit code can be created by reversing all words or by permuting the symbols in all words. Having found an optimal $E_4(6,3)$ GC 2 code, there is no need to perform a separate search for an optimal $E_4(6,3)$ GC 4 code. The $E_4(6,3)$ GC 4 code can be created from the GC 2 code by interchanging A for C and G for T.

9.3 Optimal Hamming Code Results

Tables 9.7 and 9.8 show the results of the Salmon Algorithm on several different Hamming codes. For comparison, the results from Haas and Houghten [16] are shown as well. Haas and Houghten obtained these results using a Genetic Algorithm to create an incomplete code which was then finished with Conway's Lexicode Algorithm.

Code	$A_2(12,6)$	$A_2(13,6)$	$A_2(17,6)$	$A_2(17,4)$
Best Known	**24**	**32**	**256**	**2720**
Haas and Houghten	**24**	**32**	**256**	**2238**
Salmon	**24**	**26**	**171**	**na**

Table 9.7: Salmon Algorithm Binary Hamming Code Results

Code	$A_3(5,3)$	$A_3(6,3)$	$A_3(7,3)$	$A_3(8,3)$
Best Known	**18**	**38**	**99**	**246**
Haas and Houghten	**18**	**36**	**88**	**219**
Salmon	**18**	**38**	**99**	**234**

Table 9.8: Salmon Algorithm Ternary Hamming Code Results

Clearly, the Salmon Algorithm did not perform well on the binary Hamming Codes. These codes have a regularity to their structure, and it may be that algorithms incorporating some variation of the lexicode algorithm are able to exploit this regularity. Note that the $A_2(17,4)$ case was not attempted because of time constraints. Based on the code size it was estimated a single run for this case would take about three days. Since several such runs might

be necessary to tune the parameters, it was decided that this time might be better spent on other problems.

The Salmon Algorithm performed better on the ternary codes, matching the best known on three out of four. Even here however, the algorithm gets stuck in a local optimum on the $A_3(8, 3)$ code. Extensive testing of parameter values was done on this code, but the Salmon Algorithm could not exceed a code size of 234, while the best known value is 246.

The Hamming codes displayed the same general shape as the α versus code size graph for edit codes shown in Figure 9.7. They also displayed the same tendency to give the best results near the inflection point in the graph and the same bimodal distribution at this point.

9.4 Covering Code Results

The Salmon Algorithm easily matched the known best (27) for the length 5 football pool problem. Using the maximization version of the algorithm resulted in a complete cover (243 words covered) on 30 out of 30 runs. The results were sometimes achieved in as little as 200 generations which took only about 3 seconds of run time. These results were found using $\sigma = 1/100$, ϕ values from .870 to .900 and $\alpha = 0$. Note that these values are strikingly different than those used for the Hamming and edit code problems.

Testing the minimization version of the Salmon Algorithm on the $K_3(6, 1)$ covering code problem it was found that the optimum σ was 1/100, the optimum ϕ was $.963 \pm .002$, and that any α value in the range of 0 to .2 functioned well.

It should be noted that $\alpha = 0$ is preferable to any other value in terms of the speed of the algorithm. Setting $\alpha = 0$ means that the algorithm is treating all water levels the same. If this is the case, we can dispense with the roulette selection and choose our candidate word by simply generating a single random number. We can also eliminate the section of the algorithm that deals with updating water levels, since with $\alpha = 0$ they are irrelevant. For this reason $\alpha = 0$ was used for all experiments on covering codes. However, by treating all water levels the same the algorithm has essentially been converted into a type of Evolutionary Strategy.

Figure 9.14 shows the results of the minimization version of the Salmon Algorithm run on the length 6 football pool problem. The best known size of 73 was achieved twice, one using $\phi = .962$ and once using $\phi = .964$.

The algorithm was run for 60 000 generations. The number of generations required was very large compared to the Hamming and edit code problems. In spite of this the algorithm ran fairly quickly, taking about 2400 seconds to finish. This is primarily because the ϕ value is quite high, which means the algorithm is selecting all but a few of the cover words from its memory. The selection of words from memory is much less computationally expensive than selection from the candidates list. This is because when the algorithm selects a word from memory, the candidates list does not need to be updated after each selection.

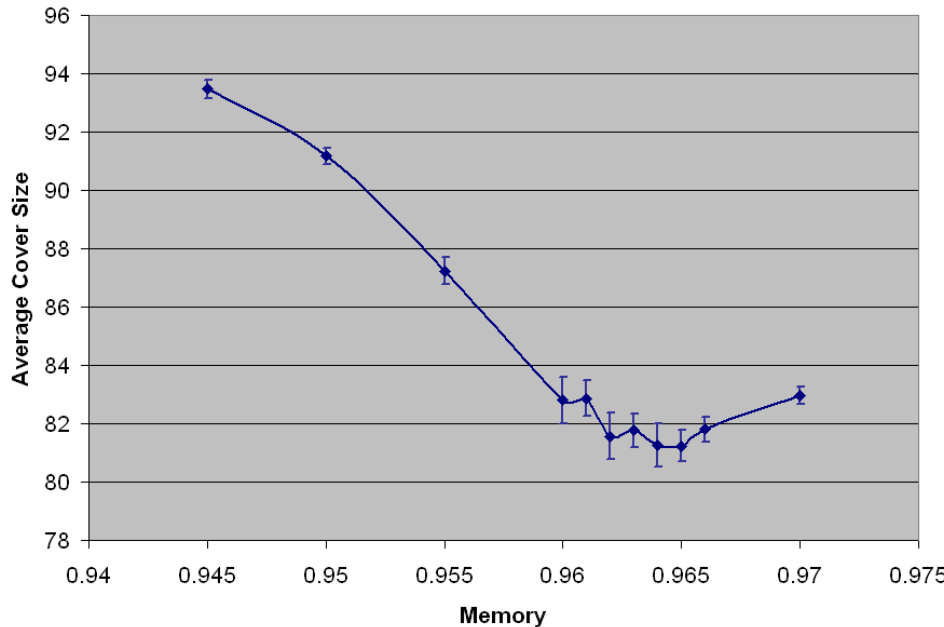

Figure 9.14: Length 6 Average Cover Size vs ϕ with 95% CI

The maximization version of the cover codes algorithm was then run on the length 6 data. This version was able to match the best known cover size of 73 much more reliably. Figure 9.15 shows the number of runs out of 30 that achieved a complete cover i.e. all $3^6 = 729$ words covered versus ϕ. Figure 9.16 shows the average number of words covered versus ϕ using a cover size of 73. The algorithm was run for 60 000 generations with $\alpha = 0$.

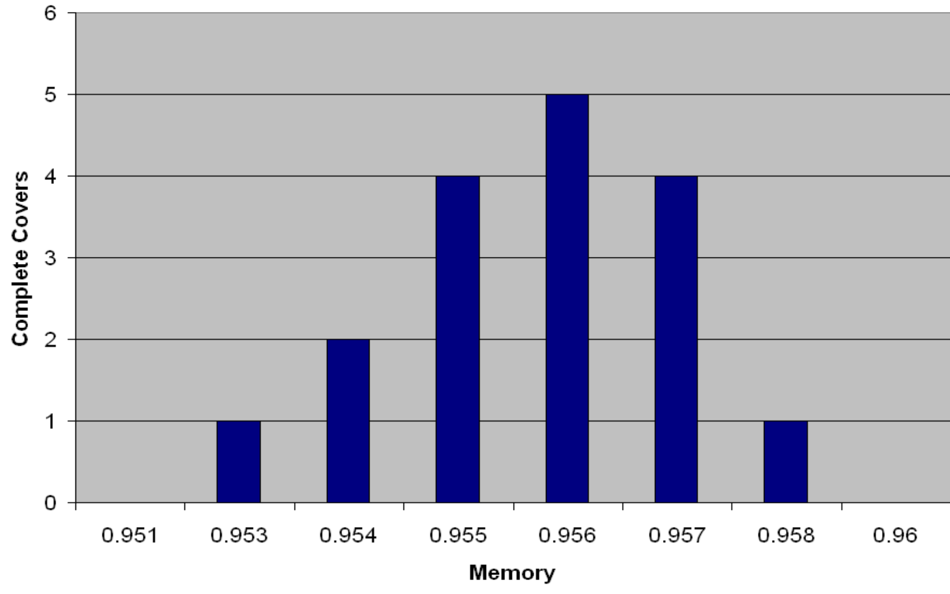

Figure 9.15: Complete Covers vs ϕ

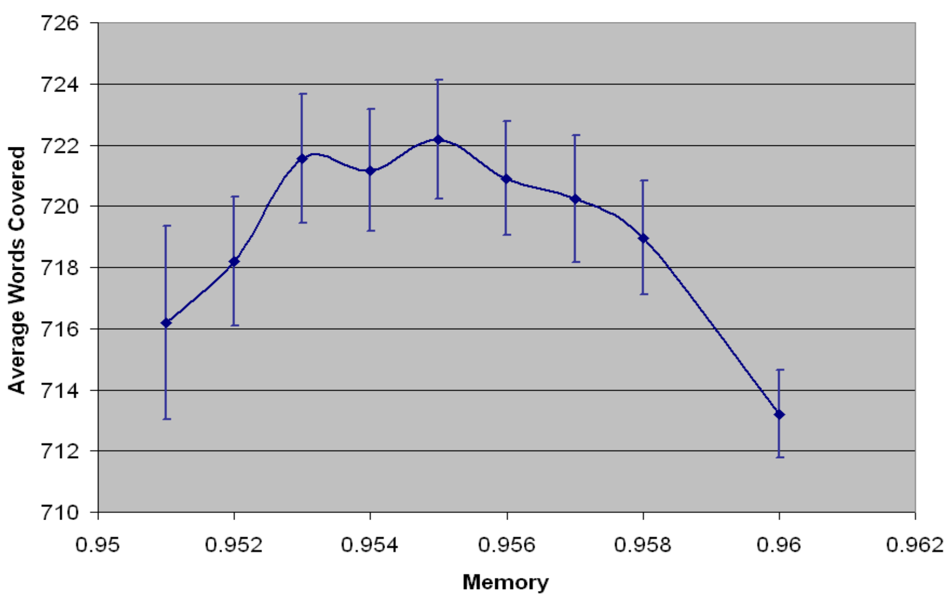

Figure 9.16: Length 6 Average Words Covered vs ϕ with 95% CI

The maximization version of the algorithm was then run with a cover size of 72 i.e. one less than the current best known. Despite extensive tuning of parameters and over 1000 total runs the largest covers found were two words short of a complete cover. This is the same result found by Van Laarhoven et al.[36].

The results from the minimization version of the algorithm on the $K_3(7, 1)$ code were disappointing. Figure 9.17 shows the average cover size obtained versus ϕ for length 7. The algorithm was run with a population of 20 for 500 000 generations to find the optimum ϕ value. It was then run for 10 000 000 generations using this value. While this may seem like an improbably high number of generations, it is comparable to the number of iterations of simulated annealing used by [36]. The smallest cover obtained had a size of 210 words which compares poorly to the best known of 186.

The population was reduced because it was found that slightly better results could be obtained using a population of 20 for 500 000 generations than using a population of 100 for 100 000 generations. Since the run times for these two cases are nearly the same, the smaller population was used. Note that additional improvements were not realized by reducing the population further to a size of 10.

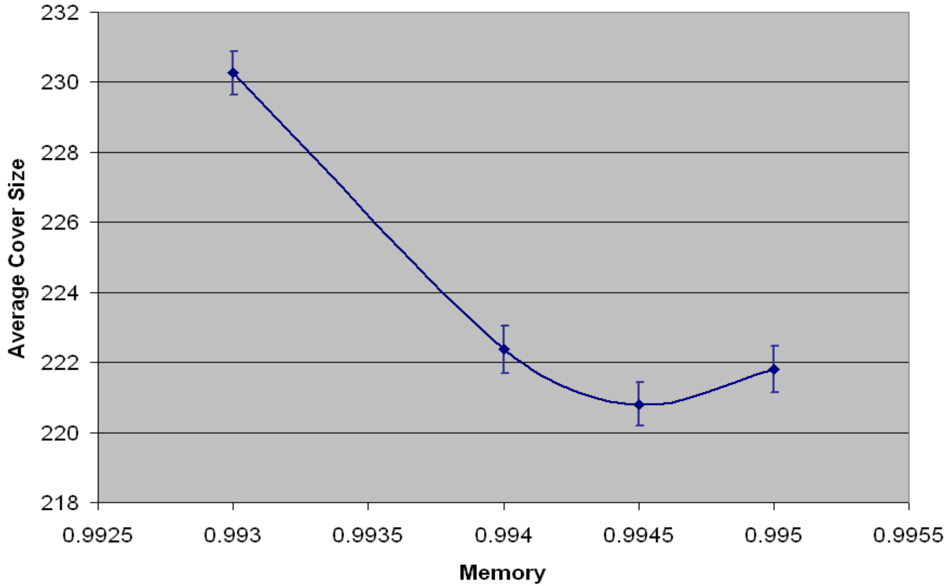

Figure 9.17: Length 7 Average Cover Size vs ϕ with 95% CI

Figure 9.18 shows the average number of words covered versus ϕ for length 7 using the maximization version of the algorithm for 100 000 generations with a population of 20 and a cover size of 186. After determining that the optimum ϕ value was .9925, the algorithm was run for 10 000 000 generations using this value. This resulted in two runs out of 30 that achieved a complete cover (2187 words covered), matching the best known value. Each individual run for this case took about 40 hours. It should be noted that although the maximization version proved to be more effective, it is more cumbersome and time consuming to use since the algorithm needs a separate run for each cover size attempted.

The cover size was then reduced to 185 and the algorithm was run 90 times using ϕ values of .9924, 9925 and .9926. The best cover found covered 2184 words, three short of a complete cover.

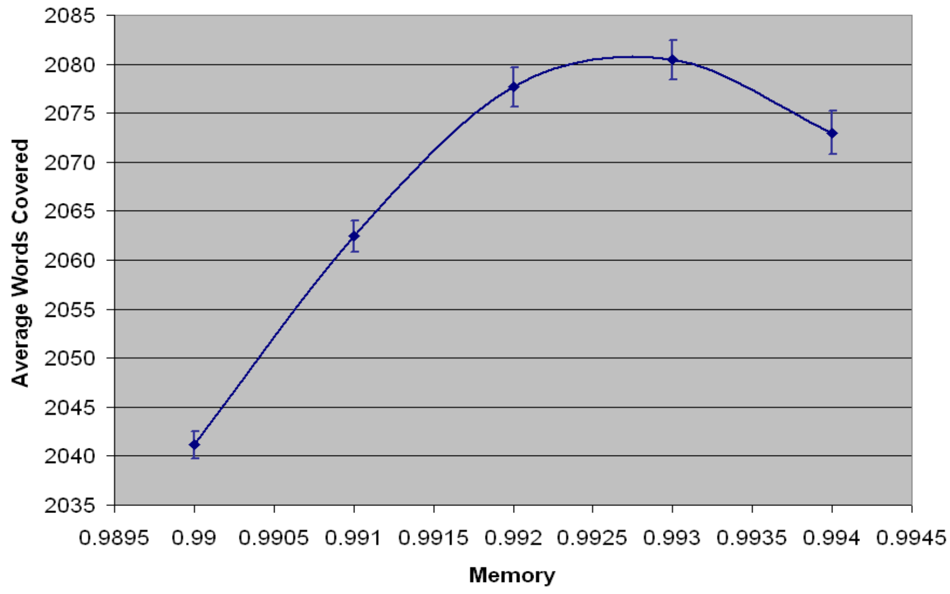

Figure 9.18: Length 7 Average Words Covered vs ϕ with 95% CI

The $K_3(8,1)$ case was not attempted because the existing best known cover of 486 was not found with a direct search, but with a search using the Blokhuis and Lam matrix construction method [4] to reduce run times. It was estimated that a direct search for 10 000 000 generations with a population of 20 would require about 7 days for this case. This was not feasible.

Any realistic attempt at the length 8 or larger cases would require reconfiguring the salmon algorithm to incorporate the Blokhuis and Lam method. Indeed, most researchers have used this method on the larger codes. See for example [27] and [4].

Chapter 10

Conclusion

The Salmon Algorithm proved to be adept at finding optimal edit codes. The algorithm exceeded the previous best known code on five cases and matched the best known on the sixth. It was also effective at finding covering codes up to length 7, matching the best known for lengths 5, 6, and 7. Its performance on Hamming codes was inconsistent. The ternary data sets functioned well, while the binary code results were poor. It was somewhat surprising, given the consistency of the edit code results, that simply increasing the length of the word by one could cause the quality of the Hamming code results to collapse. See for example the $A_2(12, 6)$ and $A_2(13, 6)$ results, or the $A_3(7, 3)$ and $A_3(8, 3)$ results.

Of course, it is possible the problem instances that gave poor results did so because the parameter values selected caused the algorithm to converge to a local optimum. As stated in Chapter 9, this thesis made the assumption that the parameter optimization curves were unimodal. In other words, if we find a clear optimum for a particular parameter value, there is no point in checking significantly above or below this value. This assumption was necessary because of time constraints, but it may not be true.

We also made some assumptions about parameter independence to save time. The parameters ϕ, α, and σ were optimized together as much as time permitted, but *population*, IM, and $number\ of\ elite$ were treated, for the most part, as independent variables. This allowed them to be optimized separately, which saved a considerable amount of time, but this may also not be true.

The Salmon Algorithm demonstrated an extreme sensitivity to small changes in some parameter values. On the edit and Hamming codes, changes

to the α value as small as .01 had a noticeable impact on the results. The cover code test cases were relatively immune to changes in α, but reacted drastically to changes in ϕ. In the $K_3(7, 1)$ case, alterations to the ϕ value as small as .0005 visibly affected the code size. This sensitivity means that it takes a long time to fine tune parameter settings. This problem is exacerbated by the fact that σ, ϕ, and α have a considerable degree of dependence on each other, and consequently should be optimized together.

10.1 Future Work

Given that much of the work in this thesis involved the tedious task of parameter optimization, it would be logical to suggest that future work might incorporate some method of mechanizing this job.

As stated at the beginning of Chapter 9, an exhaustive search of parameter values is not possible, simply because of the size of the search space and the fact that many of the parameters do not take discrete values. The strategy that most people adopt when faced with this problem is to begin with a particular parameter configuration based on past experience, then alter values one at a time, accepting the change if it produces an improvement in the results. These one at a time modifications are continued until no single change produces an improvement. This describes a hill climbing strategy. The problem with a hill climber is that it gets trapped in the first local optimum it encounters [21].

Hutter et al.[21] describe a program they call iterative local search (ILS). ILS is a stochastic search that builds a list of local optima by iterating through a loop that performs the following functions - (i) perturb solutions to escape from local optima, (ii) run subsidiary local search procedure, (iii) evaluate and retain new solutions based on some acceptance criteria. This strategy produces results superior to those achieved with the hill climbing method, and takes much less time than an exhaustive search.

One other obvious area of future work would be to apply the algorithm to other known NP-hard problems. Because augmenting water levels in the Salmon Algorithm is similar to depositing pheromone in an ant algorithm, it should be possible to use the Salmon Algorithm on any problem where ant algorithms have previously been successful. Some suggestions are the vehicle routing problem, the job shop scheduling problem, or the unbounded knapsack problem.

Bibliography

[1] D. Ashlock, Ling Guo and Fang Qiu. Greedy Closure Evolutionary Algorithms. *Proceedings of the 2002 Congress on Evolutionary Computation*, pages 1296 - 1301.

[2] D.Ashlock, S.Houghten, J.A.Brown and J.Orth, On the Synthesis of DNA Error Correcting Codes, preprint 2010.

[3] Daniel Ashlock and Sheridan Houghten. DNA Error-Correcting Codes: No Crossover. *Proceedings of the 2009 IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology*, pages 38-45.

[4] A Blokhuis, C.W.H Lam. More Coverings by Rook Domains. *Journal of Combinatorial Theory*, Series A, Volume 36, Issue 2, March 1984, pages 240-244.

[5] Andries E. Brouwer. Tables of Bounds for Hamming Codes. http://www.win.tue.nl/ aeb/, accessed February 10, 2012.

[6] Joseph Alexander Brown. Decoding Algorithms Using Side-Effect Machines. Master's Thesis, Brock University, 2009.

[7] Richard A. Brualdi and Vera Pless. Greedy Codes. *Journal of Combinatorial Theory(A)* volume 64, 1993 pages 10-30.

[8] Eugene Chiu, Jocelyn Lin, Brok Mcferron, Noshirwan Petigara, Satwiksai Seshasai. Mathematical Theory of Claude Shannon. 6.933J / STS.420J *The Structure of Engineering Revolutions*, No Date. Available on line at http://web.mit.edu/6.933/www/Fall2001/Shannon1.pdf, accessed November 27, 2011.

[9] Pierluigi Crescenzi and Viggo Kann. How to Find the Best Approximation Results − a Follow Up to Garey and Johnson. *ACM SIGACT News*, volume 29, number 4, 1998, pages 90-97.

[10] S.N.Sivanandam and S.N.Deepa. Introduction to Genetic Algorithms. Springer Publishing, 2008.

[11] M. Dorigo. Optimization, Learning and Natural Algorithms, PhD thesis. Politecnico di Milano, Italie, 1992.

[12] Marco Dorigo and and Luca Maria Gambardella. Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem. *IEEE Transactions on Evolutionary Computation*, Vol. 1, No. 1, April 1997, pages 53 - 66.

[13] A. G. Frutos, Q. Liu, A. J. Thiel, A. M. W. Sanner, A. E. Condon, L. M. Smith and R. M. Corn. Demonstration of a Word Design Strategy for DNA Computing on Surfaces. *Nucleic Acids Research*, vol. 25 1997, pages 47484757.

[14] E. N. Gilbert. A Comparison of Signalling Alphabets. *Bell System Tech. J* 31, 1952, pages 504-522.

[15] Fred Glover. Tabu Search: A Tutorial. Center for Applied Artificial Intelligence, University of Colorado, 2001.

[16] Wolfgang Haas and Sheridan Houghten. Evolutionary Algorithms for Optimal Error Correcting Codes. Brock University 2005.

[17] Hämäläinen, H., I. Honkala, S. Litsyn, P. Östergård. Football pools - A Game for Mathematicians. *American Mathematical Monthly* 102, 1995, pages 579588.

[18] R. W. Hamming, Error Detecting and Error Correcting Codes. *The Bell System Technical Journal*, April 1950 pages 147-160.

[19] Sheridan K. Houghten, Dan Ashlock, and Jessie Lenarz. Construction of Optimal Edit Metric Codes. *Proceedings of the 2006 IEEE Workshop on Information Theory (ITW 20006)*, pages 259-263.

[20] W. Cary Huffman and Vera Pless. Fundamentals of Error Correcting Codes. Cambridge University Press, 2003.

[21] F. Hutter, H. H. Hoos, and T. Stützle.. Automatic Algorithm Configuration Based on Local Search. In Howe, A. and Holte, R. C. (Eds.), *Proceedings of the Twenty-second National Conference on Artificial Intelligence* (AAAI07), AAAI Press / The MIT Press, Menlo Park, CA, USA, pages 11521157.

[22] Richard M. Karp. Reducibility Among Combinatorial Problems. University of California at Berkeley, 1972.

[23] Gerzson Keri. Tables on Bounds for Covering Codes. http://www.sztaki.hu/ keri/codes/, accessed September 16, 2011.

[24] S. Kirkpatrick, C.D. Gelatt Jr., and M.P. Vecchi. Optimization by Simulated Annealing. *Science* 220, 1983, pages 671-680.

[25] Levenshtein, V. I. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics-Doklady* 10, 1966, pages 707-710.

[26] Paul C. Marriner. Atlantic Salmon A Fly Fishing Reference. Gales End Press, Nova Scotia, Canada. No date, p 19.

[27] Patric R. J. Östergård. Constructing Covering Codes by Tabu Search. Helsinki University of Technology, 1997.

[28] Oriana Ponta, Falk Huffner, and Rolf Niedermeier. Speeding up Dynamic Programming for Some NP-hard Graph Recoloring Problems. Proc. 5th TAMC, 2008.

[29] F.Qiu, T.J.Wen, D.A.Ashlock and P.S.Schnable. DNA Sequence Based Bar-Codes for Tracking the Origins of ESTs from a Maize CDNA Library Constructed Using Multiple MRNA Sources. *Plant Physiology* 133, 2003, pages 475481.

[30] I. Rechenberg. Cybernetic Solution Path of an Experimental Problem. Royal Aircraft Establishment, Library translation NO. 1122, Farnborough, Hants., UK, August 1965.

[31] Stuart Russell, Peter Norvig. Artificial Intelligence, a Modern Approach, second edition. Pearson Education, 2003, page 119.

[32] C Shannon, A Mathematical Theory of Communication, *Bell System Technical Journal 27* 1948, pages 379-423 and 623-656.

[33] C.Solnon and S.Fenet, A Study of ACO Capabilities for Solving the Maximum Clique Problem, 2005, manuscript available on line at `http://liris.cnrs.fr/Documents/Liris-1847.pdf`.

[34] R. William Soukoreff and I. Scott MacKenzie. Measuring Errors in Text Entry Tasks: An Application of the Levenshtein String Distance Statistic, *Extended Abstracts of CHI 2001*, pages 319-320.

[35] R.J.M. Vaessens, E.H.L. Aarts, J.H. van Lint. Genetic Algorithms in Coding Theory. Eindhoven University of Technology, Department of Mathematics and Computing Science, September 1991.

[36] P.J.M. Van Laarhoven, E.H.L. Aarts, and J.H. Van Lint. New Upper Bounds for the Football Pool Problem for 6, 7 and 8 Matches, *Journal of Combinatorial Theory*, Series A 1989, pages 304-312.

[37] J.H. van Lint. Recent Results on Covering Problems, Department of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, Netherlands. No date.

[38] J.H. van Lint and T.A. Springer, Generalized Reed-Solomon Codes from Algebraic Geometry, *IEEE Trans. Info. Theory* 33, 1987, pages 305-309.

[39] Robert A. Wagner and Michael J. Fischer. The String to String Correction Problem. *J. ACM*, 21(1), 1974, pages 168-173.

[40] L.T. Wille. The Football Pool Problem for 6 Matches: A New Upper Bound Obtained by Simulated Annealing. *Journal of Combinatorial Theory*, Series A, Volume 45, Issue 2, July 1987, pages 171-177.

[41] David H. Wolpert and William G. MacReady. No Free Lunch Theorems for Optimization. *IEEE Transactions on Evolutionary Computation*, April 1997, pages 67-82.

[42] Assaf Zaritsky, An Introduction to Genetic Algorithms. Ben-Gurion University, Israel. No date. Available on line at www.cs.bgu.ac.il/ sipper/courses/ecal051/assaf-ga.ppt.