New Lower Bounds for the Snake-In-The-Box Problem: A Prolog Genetic Algorithm and Heuristic Search Approach

by

D. Scott Bitterman

(Under the direction of Walter D. Potter)

Abstract

This project establishes new lower bounds (new longest snakes and coils) for the Snake-In-The-Box problem in a hypercube of nine dimensions. The lengths obtained exceed any reported in the current literature. Three important methods or tools were used: the Prolog programming language, a Genetic Algorithm, and traditional search including depth limited and a heuristic search called the Narrowest Path Heuristic. These methods were used in conjunction and expand on others previously tried.

The Snake-In-The-Box problem consists of finding the longest simple non-cyclical path (a snake) or the longest simple cyclical path (a coil) in an d-dimensional hypercube without chords. The longer the Snake the better.

The Snake-In-The-Box problem has a range of practical applications including error detection in analog-to-digital conversion and encryption technology. Studying computational approaches to this constraint satisfaction problem and attempting to find good solutions (long snakes or coils) has theoretical value as well. The Snake Problem belongs to a set of problems known to be non-deterministically polynomial (NP). Due to exponential and explosive growth in the search space, finding solutions to such problems is notoriously difficult. Attempts to find long snakes in hypercubes of dimensions higher than seven have

since required non-traditional or heuristic search. Evaluating new approaches to solving a constraint satisfaction problem in NP has theoretical import to other similar problems, and perhaps all problems that are NP.

INDEX WORDS: Snake-In-The-Box, Evolutionary Programming, Genetic Algorithm, Heuristic Search, Constraint Satisfaction, Prolog, Hypercube, Graph Theory

New Lower Bounds for the Snake-In-The-Box Problem: A Prolog Genetic Algorithm and Heuristic Search Approach

by

D. Scott Bitterman

B.A., Spring Hill College, 1995

A Thesis Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

Master of Science

Athens, Georgia

2004

New Lower Bounds for the Snake-In-The-Box Problem: A Prolog Genetic Algorithm and Heuristic Search Approach

by

D. Scott Bitterman

Approved:

Major Professor: Walter D. Potter

Committee: Michael A. Covington
Charles Cross

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
December 2004

Acknowledgments

This thesis was very very long in the making. I thank Dr. Potter for his patience and encouragement over the years. Special thanks to Dr. Covington for dealing with my Prolog questions/issues, and Dr. Cross for tolerating my handicaps in logic. Despite not being on my committee, Dr. Rasheed was also helpful and generous with his advice on Genetic Algorithms. It was quite literally only through the faith and support of my professors, fellow students and friends that I was able to complete this project.

Much appreciation goes out to classmates and friends in the AI center (both current and long-gone) – especially Fred John, Mose Chalom, Cartic Ramakrishnan, Vassi Deltcheva, Anil Bahuman, Ronald Gonsher, Heather Silvio, David Crouch, Nelson Rushton, Darren Casella, Rajesh Kommineni, and Uli Bubenheimer (who also got me a job).

Fred Maier deserves extra-special thanks for his invaluable support and patience with my LaTeX and computer science questions and for actually reading my entire thesis. Fred has been a great friend over the years. Whether he likes it or not, he is an academic through and through!

And then there are my close friends and family who have truly born the brunt of my complaining and my seemingly empty promises over the years ("I'm going to finish my thesis this time. I swear!"): Aaron Dallas, Greg Schuler, Natalie Perrin (who won't believe I'm done until she sees a diploma), Jason Bitterman, Mom, Dad, and Diane.

Thanks to Bridget Metzger who has supported me in all the quiet ways. She fills the vast space in between and makes me laugh hard everyday. Her faith in me was equanimous and absolute. I love her deeply.

And finally, thanks to my little sister, Amy Lawson, for teaching me how to draw a better cube.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction to Snake-In-The-Box Problem

1.1 Overview

In the most general sense, this project exists as a study of the Snake-In-The-Box Problem (hereto referred to as the Snake Problem). The Snake Problem consists of finding the longest simple non-cyclical path (a snake) or the longest simple cyclical path (a coil) in an d-dimensional hypercube without chords. It was first proposed by Kautz in 1958 [Kautz58], with respect to coding theory and is still an open problem.

Thus far, attempts to verify the longest snake, i.e., the optimal snake, in dimensions higher than seven have not been successful. Other classic constraint satisfaction problems such as the Traveling Salesperson Problem (TSP) and the Hamiltonian Circuit Problem (HCP) are similar, in that attempts to find optimal or near optimal paths are notoriously difficult. As with all of these problems, the search space grows exponentially with added dimensions.

Studying and applying computational techniques to the Snake Problem have theoretical as well practical import. Like the TSP and the HCP, the Snake Problem belongs to a general class of problems known to be non-deterministically polynomial (NP). Problems in the class NP cannot typically be solved using a deterministic algorithm in polynomial time, however candidate solutions can be checked quickly in polynomial time. It is often possible to develop heuristics for such problems that find good solutions faster than a brute force search. Theoretical value is gained in studying any one of these problems because heuristic methods developed for one kind of problem may be applied to other problems in the same class. The

length of the longest snake, e.g., corresponds to the worst case number of iterations in local search algorithms [Tovey81].

Good solutions to the Snake Problem have practical use in electronic combination locking schemes [Black64, Chien64, Jablonskii74, Paterson98], error detection in analog-to-digital conversion, and disjunctive normal form simplification [Klee70]. The details of the applications of Snakes to these practical problems is not within the scope of this paper, but suffice it to say that in all cases the longer the snake the better.

With respect to the Snake Problem the focus of this project was four-fold. First, an evolutionary programming technique known as a Genetic Algorithm (GA) was used with several modifications to search for snakes. Second, all coding was in the Prolog programming language (using LPA Version 4.x for Windows), for which there were significant benefits as well as drawbacks. Third, a more traditional heuristic search is proposed for use alone or in conjunction with the GA. Using this heuristic in combination with the technique of using maximal snakes from lower dimensions [Rajan99] beat the current records found in the literature [Abbott91b, Paterson98]. This project establishes new lower bounds for snakes and coils in dimension nine (coils will be explained in the next section). Lastly, this project contributes to the literature some information and conjecture about Snakes. The use of Prolog and a heuristic as an add-on to the GA in a search for long Snakes expands on other methods tried to date.

1.2 Coils and Snakes

A snake is a simple path in a hypercube with no cycles and without chords, i.e., the snake is not allowed to "touch itself." A *simple path* is a path with no repeat nodes. A *chordless path* is a simple path in a hypercube, where given two nodes N_a and N_b, they are never adjacent when their codewords differ by more than 1. In other words, a snake is *chordless* when all consecutive nodes that make up the snake are also adjacent to each other in the hypercube (their codewords differ by only one).

A coil is a snake that "eats its tail," i.e., a coil is a simple path, which cycles exactly once without chords. Coils are a similar problem and are often considered in conjunction with snakes [Abbott88, Harary88]. See Figure 1.1 for a diagram of a optimal coil of length eight in dimension four.

An edge is the basic component of a snake or coil. An edge is formed when two adjacent nodes within the hypercube are active (form part of a path). Nodes are adjacent within a hypercube when consecutive binary codewords that make up the path differ by one bit (one coordinate). The length of the snake or coil is defined by the number of edges existing in the path of active nodes.

Coils and snakes are called different things in the literature. A coil is sometimes referred to as a *closed snake*. What is referred to in this paper as a snake is sometimes referred to as an *open snake* [Paterson98]. This paper considers both *closed* and *open snakes* in hypercubes, and will refer to *closed snakes* as coils and *open snakes* as snakes.

The search for snakes or coils of optimal length becomes exponentially explosive as the number of dimensions in the hypercube grows. For every node in a cube of d dimensions there are d connecting nodes. Consequently, the maximum branching factor is d.

The following functional notation describes the length of a given snake or coil respectively: $s(d) = SnakeLength$ and $c(d) = CoilLength$, where $SnakeLength$ will give the length of the snake found in dimension d and $CoilLength$ will give the length of the coil found in dimension d. For example, saying an optimal snake in dimension four is of length seven can be succinctly expressed by "$s(4) = 7$ is optimal."

Using a GA to find long snakes or coils has only been formally tried by Potter et. al. [Potter94] and by his students. There is little to go on for literature on programming a GA in Prolog, however it has been done and written about [Olsson96, Lamma00, Salah02]. Potter et al. found several longest or optimal snakes for s(7) using a GA programmed in C, which was verified by depth first search using parallel computing on multiple machines. An exhaustive search was also performed on snakes in dimension seven on a dedicated supercomputer by

Shende [Scheidt01]. This search took about a month, which indicates the complexity of finding optimal snakes in hypercubes.

There are numerous articles published on the Snake Problem. Many of these problems are strictly mathematical studies, but many of the more recent papers (from the past decade or so) involve computational approaches. These papers propose new lower or upper bounds for Snakes in varying dimensions and/or study the problem in general [Kautz58, Davies65, Singleton66, Danzer67, Klee67, Douglas69, Klee70, Adelson73, Deimer85, Abbott88, Abbott91a, Abbott91b, Snevily94, Potter94, Kochut94, Kochut96, Paterson98, Rajan99, Hiltgen01, Prasanna01, Kiranmayee01]. Barring mathematical discovery or insight into the Snake Problem, computational approaches will be a necessary to find new long snakes.

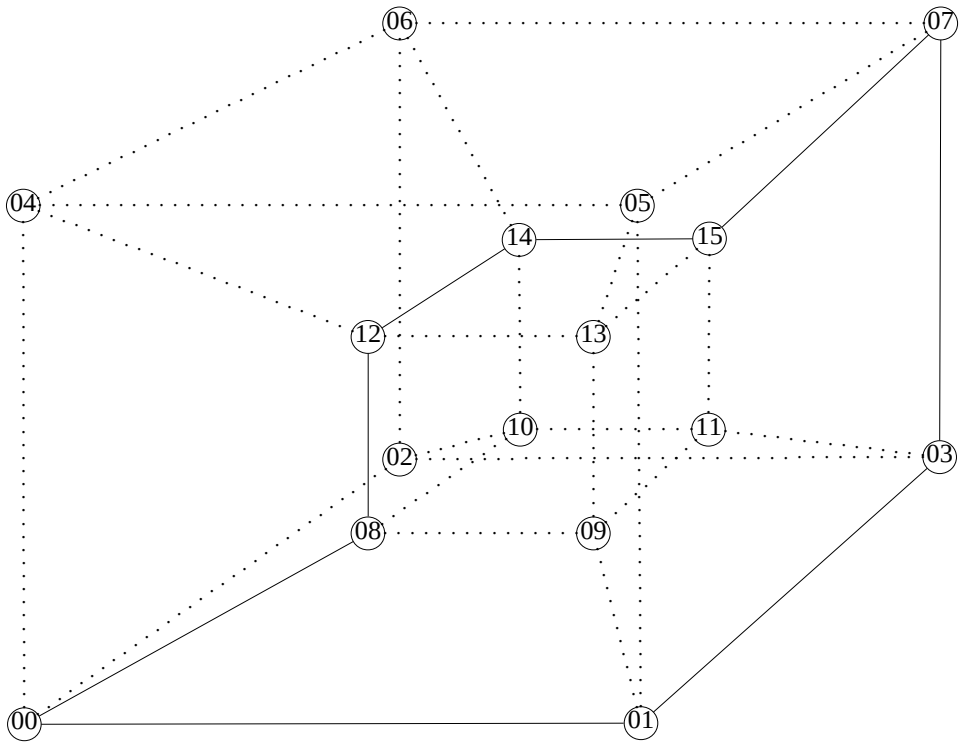

Figure 1.1: Coil of Length 8

1.2.1 Representing Coils and Snakes

There are three ways to represent a coil or snake: as binary codewords, a node sequence, or as a transition sequence. Transition and node sequence representations are based on binary codewords. This project implicitly examines snakes as binary codewords, but the two representations used explicitly throughout are node and transition sequence representations. Like [Potter94], transition and node sequence representations are used for the GA. Only node representation is used for traditional searches.

Coils and snakes are a set of binary codewords known as Gray codes. A Gray code is a set of binary codewords with the property that only one bit changes between any two codewords (each consecutive codeword has a Hamming distance of one). A snake in dimension three of length four represents five binary codewords:

```
[0,0,0]
[0,0,1]
[0,1,1]
[1,1,1]
[1,1,0]
```

The set of bits in each codeword (each representing a vertex on the cube) only differ by one bit between codewords. In a d-dimensional cube this allows the snake a maximum of d possible moves. A Snake is an *acyclic* Gray code because the last codeword does not have a Hamming distance of one with respect to the first codeword. If it did, then the Snake could be a Coil and would be a *cyclic* Gray code [Etzion96], i.e., the path would form exactly one cycle.

Node representation is achieved by converting each of the binary codewords that make up a snake into integers. The snake shown above would translate like so: [0,1,3,7,6].

Snakes can also be represented in terms of bit transition. An integer will represent which bit was flipped between codewords in order of sequence. Assuming a starting codeword of zero [0,0,0], the snake example (as binary codewords) above would be represented as [1,2,3,1], which can be read that between codewords the first, second, third, and then first bits were

flipped in sequence. Such a representation insures that the set of nodes that make up a possible snake always have a Hamming distance of one. However, this representation does not insure that the set of nodes represent a snake. For example, The following transition sequence representation [1,2,3,2] shows a Hamming distance of one between codewords, but is not a snake. The last move is invalid because it touches a previously visited node and is therefore not a simple path.

Chapter 2

Genetic Algorithms

2.1 Overview

A Genetic Algorithm was first proposed in 1973 by John Holland and his students [Holland75]. A GA consists of a population of individuals and a set of operators performed on these individuals that simulate Darwinian evolution and natural selection. A GA is not guaranteed to find the optimal solution to a problem. However, it has proved useful in finding optimal or near optimal solutions in certain problem sets, which require inordinate computational time with traditional or deterministic search techniques.

The traditional GA of the sort proposed by Holland will be referred to as a Simple Genetic Algorithm (SGA). An SGA encodes a set of candidate solutions to a problem using the binary alphabet of ones and zeroes. Each one or zero is a gene. A specific set of genes is analogous to a biological chromosome. Each chromosome is considered an individual in a GA population. A GA population is nothing more than a set of candidate solutions to a problem.

An SGA is made up of three operators executed on the population: selection (also called reproduction), crossover (also known as recombination), mutation. An SGA generation (also called a cycle or epoch) is defined as the execution of all three operators (along with a fitness evaluation of each individual) on the population. These operators will be explored in detail later.

The SGA has expanded in the years since Holland's initial proposal to allow for gene encoding of various types. In fact there is no theoretical limit as to how complex the encoding can get as long as it's finite. Integers, characters, and real numbers could all be used as

genes. The basic operations of the GA can remain the same [Mitchell98]. Integers are the most common and likely extension of the binary encoding, where some range can be used as possible genes. For example, the GA was expanded to encode and search for solutions to the Traveling Salesperson Problem, where each city in the search is assigned an integer between one and n, where n cities are examined. The individual is a chromosome of n genes having a length of n [Homaifar93, Tamaki94].

Like the project of [Potter94], this project extends the SGA to encode a snake in one of two ways: transition sequence or node sequence. So called "messy GAs" have been proposed and studied [Goldberg90], where the length of the individuals is variable. This project used, in all cases, a fixed length chromosome. It is known from mathematical proof that the upper-bound U (the longest a snake can possibly be) for $d \geq 7$ satisfies the following expression: $U \leq 2^{d-1} - (2^{d-1}/20d - 41)$ [Snevily94]. This equation is used to establish the lengths of individuals for the GA in searching for snakes. For s(8) this equation calculates 126.9. Thus, all individuals are set to a fixed length in dimension 8 of 127 (the upper-bound of 126 + 1).

A core tenet of GAs is the development and evolution of good low ordered schemas. Schemas are subsets of genes (from the entire individual/chromosome) called "building blocks." It is through these building blocks (smaller chunks of genetic material) that good individuals overall are formed. This logic describes the schema theorem of GAs [Holland75, Goldberg89]. A schema is a gene pattern from an individual that represents aspects of the total chromosome.

Maximizing good results and thus the power of GAs is a delicate balancing act. On the one hand, there is the risk of too much genetic diversity. This can be caused by a myriad of factors including too high a rate of mutation or crossover, as well as a faulty fitness function (that may bias results to sub-optimal paths). On the other hand, there is the problem of not enough genetic diversity caused by too low a mutation or crossover rate. Too little diversity in the population can result in premature convergence to a local maxima. Premature convergence is when the average fitness of the general population hovers around

the fitness of a sub-optimal and typically undesirable individual(s). In effect the population has evolved to a relatively low fitness state with little genetic diversity.

To attempt a greater balance between too little and too much genetic diversity, three additional operators are introduced to the SGA to search for long snakes: (1) elitism — to preserve the best individual between generations, (2) catastrophe — to combat premature convergence, and (3) seeding — to encourage better individuals overall. The following sections explain how SGA operators and these three additional ones were implemented. In addition the GA was combined with traditional search techniques and other minor enhancements, which will be discussed in coming chapters.

2.2 Initial Population Creation

With the Snake Problem, it is hypothesized that it is undesirable to create the initial GA population completely at random. By completely at random, it is meant to build individuals with genes picked at random from the total set of possible genes (respective to representation).

To combat the problem of poor initial individual creation, a Guided Random Initial Population (GRIP) operator is introduced. The GRIP operator creates an individual using transition sequence representation by removing the current and previous gene (if they exist) from the list of possible genes (one to d), and picking from the remaining list at random. Assume a search for s(4) with an existing individual of [2,1]. The list of all possible genes is [1,2,3,4]. Removing the current gene [2] and the previous gene [1] leaves [3,4], thus the next gene in the individual must be 3 or 4. All coding was done in Prolog and all individuals were represented with the Prolog *list* data structure. The simplest and most intuitive way to construct lists in Prolog is by adding members to the left of the list. Thus, the new individual will be [3,2,1] or [4,2,1]. The operator continues to build the individual in this way until the desired length is reached.

The GRIP operator prevents two things from happening. It prevents a return to the current node, which occurs with identical adjacent genes, e.g., [1,1]. And it prevents small coils of length four from forming, as in the case of [1,2,1]. In node representation this is [0,1,3,2,0]. Due to their shortness, coils like this are undesirable in searching for long snakes.

A node representation individual is formed by creating an individual with the GRIP operator (in transitive representation) and then converting to node. Thus, the same GRIP logic applies for both node and transitive snake representations.

The GRIP operator creates individuals at random, yet in a guided way. The initial population will contain relatively good schemas from the start without sacrificing genetic diversity. It is hypothesized that this will aid in the GA finding longer snakes.

2.3 Selection

Selection simulates part of a reproductive cycle in biological systems. It is the operator that decides who lives or dies within a generation. Selection is also called reproduction [Goldberg89]. It is important to remove, at a reasonable rate, bad individuals from a population so that it will gradually move toward a more fit population. The selection operator selects p number of individuals out of the current population, where p is the population size. The variable p will be used to denote population size in this section. Selection compares the fitnesses of individuals from the population.

There were three types of selection used in this project: roulette wheel, rank, and t-tournament. All forms of selection run p times to select p individuals, where p is the total population size.

Since snake fitness is independent of snake representation, all three types of selection can be utilized with either node or transition representation. This project makes no changes to the ways in which selection is typically used in an SGA [Holland75, Goldberg89].

2.3.1 Roulette Wheel Selection

In roulette wheel selection the wheel is a metaphor for the sum of all fitnesses in the population and a "slice" of the wheel is given to each individual corresponding to its fitness. Put mathematically, roulette wheel selection assigns the probability of survival of each individual as equal to an individual's fitness divided by the sum of all fitnesses. In this way, individuals of higher fitness are assigned a larger "slice" of the wheel than individuals of lower fitnesses. This allows for the possibility of lower fitness individuals to survive but with chance proportional to fitness. This encourages genetic diversity in the population while giving appropriate bias to the good genetic material.

Within each iteration of roulette wheel selection (of which there are p iterations), the wheel is "spun" and lands on exactly one individual. The code does this by summing up the fitnesses of all individuals and picking a pseudo-random number between zero and this sum. Each individual is assigned a range equal to its fitness within the total range of zero to the sum of all fitnesses. If the pseudo-random number picked falls within the range of a particular individual, then this individual is chosen from the general population to go on to the next generation. This operator is a reliable form of selection, but has the downside that it requires more computation and time to run than t-tournament.

2.3.2 Rank Selection

Rank selection was first proposed by James Baker [Baker85] and is often chosen to prevent premature convergence to a local maximum. It works much in the same way as roulette wheel selection. Each individual is assigned a corresponding chance of survival based on its rank in the total population. First, the population is ordered according to fitness. Second, each individual is given a rank relevant to the ordering. Thus, the least fit individual is assigned a rank of one. The second least fit is assigned a rank of two, and so on. The most fit individual is assigned a rank of p. Third, the sum of these ranks is obtained using the following expression for summing arithmetic sequences: $p((p+1)/2)$. Selection proceeds in

the same way as it does with roulette wheel, except the sum of the ranks is used instead of the sum of the fitnesses. Each individual is given a probability of survival equal to its rank divided by the sum of all rankings. Due to having to order p individuals, rank selection is the most time intensive of the three types of selection used in this project.

2.3.3 t-Tournament Selection

Tournament selection or t-tournament selection requires the least computation and thus runs the fastest of all three types of selection. With each new generation, Rank selection requires that the entire population be ordered according to fitness. Whereas Roulette Wheel selection must traverse the entire population to compute the sum of all fitnesses before performing selection, t-tournament selection only needs to pick t members of the population at random for comparison p times.

Tournament selection [Goldberg91, Mitchell98] picks two individuals from the population at random. The individual with the better fitness is dominant. The better individual is picked over the lesser one based on a probability. This probability gives the user greater leeway to adjust selection pressure among candidate solutions. The probability can be adjusted to slow or speed up general population convergence toward better individuals.

This project modifies tournament selection as examined by [Goldberg91, Mitchell98] with t-tournament selection. t-tournament adds to tournament selection by not limiting the comparison pool to only two individuals. Instead t individuals will be picked at random from the total population. The value of t is set at run time by the user. Just like traditional tournament selection there is a percentage set for picking the dominant individual. If the dominant individual is not selected for survival, it is removed from the pool of individuals being compared. The next best individual is then established as dominant. Selection continues in this way on the reduced pool until either a dominant individual is selected or only one member in the pool remains. Experimentation was done with two and three tournament selection.

The term *tournament selection* will refer to t-tournament selection as defined above.

2.4 Crossover

Crossover simulates mating in biological systems. For GAs it is the process of combining the genetic material of two "parent" individuals into one or two offspring. For the Snake Problem this will be done by exchanging integers (genes) between the two parents in some way.

There were two types of crossover used in this project: n-point, and Enhanced Edge Recombination (EER). EER was used only if node representation was used. n-point crossover was used solely with the transition representations. The general crossover mechanism picked two individuals from the population at random and mated them.

2.4.1 n-point Crossover

n-point crossover is easy to implement and useful for representations that don't require strict adjacency between genes or schemata. n-point crossover picks two individuals from the population at random. Since all individuals for s(d) are of the same length L, then there are $L-1$ possible crossover points in all cases. Any two genes have one crossover point between them. For n-point crossover, n random integers between one and $L-1$ are picked for crossover. For variation this project experimented with one, two, and three point crossover. Consider the following two transition sequences for a individual represented as a potential snake in a cube of four dimensions:

```
[3,2,1,4,1,2,3]
[1,2,3,4,3,2,1]
```

If the GA is set to execute one-point crossover on these two "parents", then a random number between one and six is generated. Suppose this random number is three. The two parents would crossover after the third gene like so:

```
[3,2,1|4,1,2,3]
[1,2,3|4,3,2,1]
```

The genetic material from one parent before the crossover point would be traded with the other parent resulting in two children. Each of the children contains transition sequence information from each of the parents resulting in:

```
[1,2,3,4,1,2,3]
[3,2,1,4,3,2,1]
```

One-point crossover is limited to swapping the front-end (or the back-end depending on your point of view) set of genes between parents. Two-point crossover is a bit more complicated, with two random numbers picked within a range of 1 to $L - 1$. Consider the following two parents:

```
[2,3,2,1,4,3,4]
[4,3,4,1,2,3,2]
```

Assume that the two crossover points picked are two and five:

```
[2,3|2,1,4|3,4]
[4,3|4,1,2|3,2]
```

Unlike one-point crossover the genetic material between the two crossover points is exchanged resulting in the following two children:

```
[2,3,4,1,2,3,4]
[4,3,2,1,4,3,2]
```

It is in this way that n-point crossover works for this project in traditional ways [Goldberg89, Mitchell98]. For every two parents crossed over, two children are produced, thus there are $PopulationSize/2$ iterations of n-point crossover per generation.

2.4.2 Enhanced Edge Recombination Crossover

n-point crossover cannot be used for node representation because gene/schemata adjacency is crucial. Longer snakes would be disrupted too often and in undesirable ways. Enhanced Edge Recombination (EER) crossover was implemented to preserve basic adjacency. It is a useful crossover where the adjacency of the genes factors into the value of the solution [Starkweather91]. This technique is often used in the TSP problem where the particular adjacency of the cities (represented by integers) is crucial [Whitley89, Whitley90]. EER is also used to prevent the repetition of redundant genes. In the case of the TSP problem as well as Snakes, the repetition of gene "2", for example, is redundant. For the TSP problem, the same city is not to be visited twice. For the Snake Problem, it's illegal to visit the same node twice.

This project implemented EER as proposed by Whitley, Starkweather, et. al. [Whitley89, Whitley90, Starkweather91]. Potter et al. also used EER as their crossover technique for the Snake Problem [Potter94].

Like n-point crossover, EER picks two individuals at random from the population, but unlike n-point, these two "parents" will only produce one "child." The reasons for this will become apparent in the explanation below. For EER an "edge table" is constructed, which is also an adjacency table with respect to the individual. This table is then queried and modified to preserve adjacency.

Consider the following two snakes of length eight in dimension four:

```
[0,1,3,7,6,14,12,13]
[0,8,12,13,15,4,3,2]
```

An adjacency table is constructed for both parents (see Table 2.1).

This table reflects which genes are adjacent to which. Notice nodes 12 and 13 are flagged. Nodes are flagged in the table to indicate that they are part of a common adjacency subsequence within the individuals. Nodes 12 and 13 are connected to each other in both parents.

Gene	Adjacencies
0	1,8,13,2
1	0,3
2	0,3
3	1,7,4,2
4	15,3
6	7,14
7	3,6
8	0,12
12	14,-13,8
13	0,-12,15
14	6,12
15	13,4

Table 2.1: Enhanced Edge Recombination Adjacency

To further preserve adjacency and good schemata, such common subsequences are given the highest priority in the creation of offspring.

Given these adjacency tables offspring are created as follows: (1) Select a starting gene at random. Initialize this gene as the *CurrentGene*. (2) Find all genes that are connected to *CurrentGene*. Choose the connected gene that has negated (flagged) nodes first. If there are no genes that have flagged connections, then choose the connected gene that has the fewest number of links remaining in its edge table entry. In the case of a tie, pick a gene at random from the candidates. (3) Update the edge table by removing from it all instances of *CurrentGene*. (4) Reset *CurrentGene* variable to newly picked gene. (5) Repeat steps two through four until the "child" offspring is complete. The child will be complete once the number of genes is equal to the set length of the individual.

Using the example from above, assume that gene [1] was picked as the staring gene. Gene [1] is connected to [0,3]. [0] is connected to [1,8], but [3] is connected to [1,7,4,2]. Since [0] has fewer remaining links than [3], then [0] is chosen as the next gene in the sequence. Once a node is chosen, i.e., once a node becomes the *CurrentGene*, then that node is removed from the table.

There are three drawbacks to using EER: (1) there are significantly more steps involved in constructing and modifying tables that wouldn't be a concern with n-point crossover. (2) Each EER operation only produces one child. These are unfortunate, but unavoidable side effects of EER. Either it or a similar crossover mechanism must be used to preserve adjacency for snakes represented as node sequence. (3) EER as proposed by Whitney et al. works well with the TSP problem because the set of all nodes in both parents are identical. With the individuals as constructed in this project, this is not the case. The Snake Problem as represented in this project allows for the possibility of repeat nodes within an individual, e.g., [0,8,12,13,15,8,0,2] where [0] and [8] appear twice. Since the search for snakes (unlike coils) doesn't allow cycles, the first and last nodes in an individual are not connected. The length an individual is defined by the upper-bound established for s(d). This means for example that s(8) is of length 127. Even assuming there are no repeat nodes in each parent, it would be possible for each parent to have no common nodes at all between them. This disrupts the functioning of EER and lead to very poor performance.

EER crossover needs significant modification to work well with node representation and the Snake Problem. Notwithstanding its poor performance, it remains included in the paper as a warning to those who would use it. EER may not be without redemption. One could, for example, make the individual length equal to the total number of possible nodes 2^d and not allow repeat nodes. This would mimic the TSP in that all parents would contain the same set of nodes. Since transition sequence obtains good results, this project focuses on it and enhancing other GA functions to search for long snakes.

2.5 Mutation

When an organism in nature reproduces there are often random fluctuations in gene structure called mutations. In an SGA, where individuals' genes consist only of ones and zeros, a gene is picked at random for mutation and the bit is simply flipped. If the gene is the bit '1', it is made '0' and vice versa. Since this project used integer representation, mutation consisted

of picking a gene (a positive integer) at random and replacing it with another valid positive integer — also picked at random. Genes are changed in this manner with a probability set by the user at run time. Mutation increases genetic diversity in a population and helps to mimic natural evolution.

While the set of possible genes into which a gene could mutate was different for node and transition sequence representations, the basic concept behind the mutation was the same. In the most general case, node representation could mutate to any integer ranging from zero to $2^d - 1$ and transition sequence representation to any integer ranging from 1 to d.

In order to save time, the number of possible genes for mutation was computed at the start of a GA run. This number M_p is equal to the number of individuals in the population p times the number of genes in each individual. The number of genes in an individual is also the length of the individual l. This method yields the following equation: $M_p = p * l$

The number of genes to be mutated per generation was computed by picking a random real number between zero and one M_p times. If this number fell below the set mutation rate, then a counter was incremented. Once the number of genes to be mutated G_m was calculated, then the mutation procedure was called exactly G_m times per generation. This reduced total mutation operation time significantly. Instead of running the mutation operator (where the probability of mutation is checked) for every gene in an individual, it only runs G_m times. Computing the total number of genes to mutate before running the GA diverges from the standard SGA practice [Goldberg89], but achieves the same result while saving time.

After the number of genes to be mutated is calculated, mutation proceeds according to the following steps: (1) Pick an individual at random. (2) Pick a gene at random from the individual. (3) Replace the picked gene with another gene picked at random from a set of valid genes. The set of valid genes will depend on the type of mutation and representation used. (4) Decrement the counter of genes to be mutated and repeat all steps until the counter is 0.

There were two types of mutation used in this project: random, and guided mutation.

2.5.1 Random Mutation

The random mutation operator uses as its set of valid genes any gene from the range of all possible genes for a snake in a hypercube of d dimensions respective to representation. The advantage to random mutation is that it is runs quickly. It does not need to adjust the list of valid genes into which to mutate. However, its drawbacks are many. It is entirely possible that the node could mutate to itself. In an eight dimensional hypercube using node representation, node 233 has just as much chance as mutating back to 233 as it does any other single node. The much greater danger lies in mutation overly disrupting good schemas.

Since any node from the set of all possible nodes can be picked, it is just as likely as not that repeat nodes will occur for transition sequence representation. Assume that the following schema's second node is to be mutated for node representation in a cube of three dimensions: [0,1,3,7]. In this case, 1 could just as easily mutate to 0, 3, or 7 as it could any other possible node. Even more likely is the mutation to an illegal gene (repetitious or not) that breaks a long snake into two short ones hopelessly unable to connect.

Schema disruption may not be so bad in some cases, and may actually lead to a good solution upon crossover or future mutation. Schema disruption is more of a problem when the adjacency of genes is important. Gene adjacency is crucial with both node and transition sequence representations. A guided version of mutation was developed to combat these problems.

2.5.2 Guided Mutation for Node Representation

Guided mutation operates in one of two ways depending on the snake representation. In node representation the following steps take place: (1) Take the nodes preceding and succeeding the node to be mutated and retrieve the lists of the nodes connected to these from the connectivity table. Append these lists together (removing any repeat nodes) to create a new list of valid mutation points. Note that if mutating the first or last nodes of the individual there will be only one node either succeeding or preceding it, respectively. (2) If the node to

be mutated is in the list of possible move points, then remove it. Else, leave the list as is. (3) Pick a node at random from the new list and use this node as the new mutated gene.

Assume the third gene in the following individual has been picked for mutation in a search for s(4):

```
[0,1,9,7,15,14,12,8]
```

The gene to be mutated is [9]. The preceding gene is [1] and the succeeding gene is [7]. The preceding and succeeding nodes serve as guiding points. The nodes connected to [1] are retrieved from a Prolog table. They are [0,5,3,9]. The nodes connected to [7] are [6,5,3,15]. Appending the two lists (removing any repeats) results in [0,5,3,9,15]. Since the gene to be mutated [9] is a member of the list, then it is removed. This leaves [0,5,3,15] as the set of possible nodes into which to mutate. A node from this set is picked at random and replaces the gene to be mutated.

From the example above, nodes [5,3] are better picks than [0,15]. Nodes [0,15] already exist in the current individual whereas nodes [5,3] do not. Further modification and improvement of the guided mutation operator for node representation might remove the two preceding and two succeeding nodes from the list of valid nodes. This would likely narrow the list of mutation points to a set of more favorable nodes for snakes. Since the focus of this project was on transition sequence, this modification was not made.

2.5.3 Guided Mutation for Transition Sequence Representation

Guided Mutation for transition sequence representation follows a similar logic, but the list of possible mutation points is different. For transition sequence the list of possible mutation points will always be a list of positive integers from 1 to d in a cube of d dimensions. Guided mutation for transition sequence proceeds by these steps: (1) Make a list of guiding genes consisting of the gene to be mutated, the previous gene and the succeeding gene. If the first gene of the individual is to mutate there is no previous node and the list of guiding genes will

be the gene to mutate and the succeeding gene. Likewise, if the last gene in an individual is to mutate, then the list of guiding genes will consist of the gene to mutate and the previous gene. When joining these lists any repeats are removed. (2) Get the list of possible mutation points. (3) Remove all members of the list of guiding points from the list of possible mutation points. (4) From this remaining list pick a member at random as the mutation point.

As an example, take the following transition sequence for a snake in five dimensions. Assume the third gene is picked for mutation:

```
[1,2,1,4,1,5,2,3,4,5,3,2,1]
```

The list of guiding points is the node to mutate [1], the previous node [2], and the succeeding node [4]. Thus, the list of guiding points is [1,2,4]. The list of all possible mutation points is [1,2,3,4,5] (1 to d). Removing the guiding points from this list leaves [3,5]. One of these genes is picked at random as the gene in which to mutate. Guided mutation can only be used for transition representation in a search for s(d) where $d > 3$. For an s(d) search where $d < 4$, it would be possible to remove all nodes from the set of possible mutation points, leaving no nodes into which to mutate.

Since there are significantly more steps in guided mutation compared to random mutation, the downside is greater computational time. The upside is that greater node adjacency is preserved in the case of node representation, and longer snakes are more likely to be preserved or created in both node and transition sequence representations.

2.6 Catastrophe

A catastrophe operator was implemented to combat GA convergence and stagnation in the average fitness of a population. A similar operator with the sexier name "Total Comet Strike" (TCS), has shown to yield significant improvements in "(1) the number of runs that will converge within a desired time and (2) the average time for convergence for the set of

runs" [Dickens98a, 7]. The TCS destroys all but the best individuals in a population after a set number of generations.

Another similar operator exists in a GA product called GeneHunter. GeneHunter "destroys most of the population (all but the elite individuals) and refreshes it with a large set of new individuals... causing extinction when so much inbreeding takes place that no offspring are fitter than their parents for many generations" [Lewinson95, 26].

The operator used in this project, simply called *catastrophe*, can be set to destroy n individuals or $p1$ percent of the population, but only after the average fitness of the population has not changed by more than $p2$ percent for the past g generations.

During a typical testing run using catastrophe, 90% of the population was set to be destroyed and replaced with a new initial population (using the set initial population creation mechanism). This segment of the population was destroyed after 10 generations where the average fitness did not deviate by more than two percent. The remaining 10% of the population consisted of randomly chosen individuals. If elitism was active, the set of elite individuals was preserved.

As results will show, catastrophe did not improve GA performance.

2.7 Elitism

Elitism is a widely used operator with GAs where the best n individuals or best p percentage of a population are selected to carry over between generations. It was first proposed by De Jong [DeJong75]. The set of individuals selected as elite are exempted from other GA operations. Elitism preserves high fitness at the cost of genetic diversity, but can prevent the best individuals from being lost or disrupted by crossover, selection, mutation, or a catastrophe. Many researchers have found that elitism can (in many cases) significantly improve a GA's performance [DeJong75, Goldberg89, Mitchell98].

In all cases for this project, the elitism operator used did not diverge from that first proposed by De Jong. That is, n individuals were selected as elite and preserved between

generations for all generations. In all cases n was set to one. Only the single best individual was considered elite. This insured that the best individual was never lost, yet genetic diversity was maximized.

2.8 Seeding

Seeding is inserting a biased set of individuals (usually better than the average) into the population. The intended effect is to sway the average fitness of a population toward the better seeded individual(s). In this way, the population is infused with good genetic material at some point. In theory this will give the GA a boost in performance. Ultimately, it is the hope that this good genetic material will lead to better solutions.

Like elitism, seeding is a commonly used operator for enhancing GA performance. It was used by Potter et al [Potter94] in their GA to attack the Snake Problem. They seeded GA populations with the best snakes found from previous GA runs. Unlike Potter et al. the seeded individuals in this project come from a Random Snake Generator (RSG) function that builds snakes by picking nodes at random using a Depth Limited Search (DLS). Seeding took place only when the initial population function was called. This means that seeding could also occur when the catastrophe operator was invoked.

In all cases some percentage of the initial population (set by the user) was seeded instead of being created by the initial population creation function. The RSG generates all snakes in node representation. If transition representation is being used, then the resulting snake is translated from node to transition representation. There are two settings the user must provide at run time for the seeding RSG mechanism to work: (1) the percentage of the population to seed and (2) the number of states s for the RSG to search before halting and returning a snake.

RSG is a kind of traditional search. By "traditional search" it is meant that states in a search tree are examined explicitly (as opposed to GAs, which examine states implicitly). All explicit searches involve a start and goal state and a queue of states left to examine

before the search returns success or failure. Traditional searches are differentiated only in how newly expanded states (the child states of the current state) are ordered relative to the queue. Depth First Search simply places these child states at the front of the queue for consideration without ordering. Like DFS, RSG places the newly expanded nodes at the front of the queue, but these states are put in random order first. It is only this that differentiates RSG from DFS.

RSG generates a snake by randomly searching through a tree of possible solutions examining s states before halting and returning the best snake found so far. These snakes are then used to seed the population when the initial population operation is called.

2.9 Fitness Functions

Of all the operators that make up the GA, the fitness function is perhaps the most crucial. Without it selection would not be possible and there would be no way for the GA to converge on good or optimal solutions. A fitness function is what guides the evolution of a population.

For all GAs, the fitness function evaluates every individual's genetic material assigning a value to each at the beginning of a GA generation. There were three fitness functions used: (1) The fitness function proposed by Potter et al. [Potter94], which is that of the length of the longest snake in an individual cubed. (2) The snake cubed plus the sum of the squares of all lesser snakes within an individual. (3) A fitness evaluation based on a snake's narrowness with respect to the search tree.

2.9.1 Longest Snake Cubed Fitness

The fitness function developed by Potter et al. works well in two ways. First, by cubing the length of the longest snake in an individual, fitnesses are weighted to greatly encourage individuals containing long snakes. This is a very intuitive approach. Second, when using roulette wheel selection a fitness function of this type is crucial. Individuals with long snakes need to be given sufficient bias on the wheel to increase chance of selection.

2.9.2 Longest Snake Cubed with Lesser Snakes Fitness

A fitness function that considers lesser snakes within an individual adds to that of [Potter94]. Individuals with long snakes might also contain other long snakes that are shorter than the longest. This is good genetic material, which might result in a good long snake later in the evolutionary search. With the [Potter94] fitness function, these lesser snakes are ignored. This project proposes taking the lengths of all lesser snakes, squaring each of their lengths, and adding them to the length of the longest snake cubed. This gives the lesser snakes some value proportional to their length, but still allows the best snake to be dominant.

2.9.3 Narrowest Path Fitness

Results from the Narrowest Path Heuristic (which will be explained in greater detail in the "Traditional Search Techniques" chapter) show that snakes with more possible legal nodes in which to move are more likely to lead to longer snakes. Snakes that follow a narrower path through the search space tend to maximize the number of legal nodes left in which to move (generally speaking). This project proposes evaluating the longest snake found in each individual in terms of the narrowness of its path through the hypercube search space. Narrowness is determined by evaluating the actual next node N_b with respect to the other possible next nodes, i.e., the other possible legal moves from the current node N_a.

The Narrowest Path Fitness followed these steps in calculating fitness:

1. Pull the longest snake from the individual. Convert the snake to node representation, if not already in this form.

2. Assign the variable $CurrentNode$ to the starting node of the snake. Remove the starting node from the total list of legal nodes, which will always initially contain the range of integers from zero to $2^d - 1$. Assign the remaining list of legal nodes to the variable $Legals$.

3. Retrieve all connections to the $CurrentNode$ from the lookup table. Remove any members from the list of connections that are not in the $Legals$ list. Assign the remaining list of connections to the variable $ValidConns$.

4. Evaluate each of the members of the $ValidConns$ list in terms of narrowness. First, remove all members in the list $ValidConns$ from the $Legals$ list to obtain an updated $Legals$ list. Second, query the look up table for the list of nodes connected to each member of $ValidConns$. Call this list of the list of connections to $ValidConns$ $SubConns$. Third, count the number of legal moves that can be made from each member of $ValidConns$. The number of legal moves that can be made from each of the $ValidConns$ is found by counting the number of each of the $SubConns$ that are also members of $Legals$ list.

5. Order these nodes into sets based on those nodes that have only one legal move, then two, then three, and so on. The paths that have the fewest number of legal nodes are the narrowest.

6. Take the *actual* next node and get its rank from the sets of rankings. Calculate fitness by dividing the actual node's rank into one. This means that if the actual node was one of the narrowest nodes (a rank of one), then its value would be one. If the actual node was one of the second narrowest nodes (a rank of two), then its value would be 0.5 and so on. Assign the *actual* next node to the variable $CurrentNode$ and repeat steps (3) through (6) until the end of the snake is reached. The values for each node are summed and the summed value is cubed to obtain the individual's fitness. A perfectly narrow snake would always move to the narrowest path and would have a Narrowest Path Fitness equal to the length of the snake cubed.

As an example take the following individual in a search for s(4) in node representation: [0,1,3,7,15,14,12,4]. The longest snake in this individual is [0,1,3,7,15,14,12]. Assume the fitness has been evaluated for the snake until $CurrentNode$=7. A detailed example of a telling step in the fitness function using the narrowest path fitness evaluation is below. Relevant variables are in italics with explanation.

$CurrentNode = 7$.

$NextActualNode = 15$.

$CurFit = 3$.

$Legals$ = [6,10,12,13,14,15] — all possible nodes in a four cube minus the starting node of [0] and all the connections to all nodes that make up the previously evaluated snake (currently [0,1,3]) excepting the $CurrentNode$.

$ConnsToCurrentNode$ = [3,6,15,5].

$NewLegals$ = [10,12,13,14] — $Legals$ less $ConnsToCurrentNode$.

$ValidConns$ = [6,15] — $ConnsToCurrentNode$ minus those connections not in the $Legals$ list.

$SubConns$ = [[7,4,2,14],[14,13,11,7]] — a list of the list of connections to nodes [6] and [15] respectively.

If a move was made to node [6] there would be only one valid node in which to move: [14]. The other nodes connected to [6] are not in the current $Legals$ list. Since this is the narrowest path possible (one legal move), then assign this move a rank of one. If a move to node [15] was made, there would be two legal nodes in which to move: [13,14]. Assign node 15 a rank of two. Since the actual node is [15], the fitness value for moving to this node is $1/ActualNodeRank$. In this case the $NewFit = CurFit + 1/2 = 3.5$. Once the entire snake is examined in terms of narrowness, the narrowness value is cubed to obtain the final fitness of the individual.

Evaluating individuals in terms of narrowness takes additional computational time, which is a drawback. Though it was hoped that this evaluation would aid in finding better snakes, the results were not promising. Because of this the obvious extension of this fitness function to evaluate lesser snakes (as was done with the Longest Snake Cubed with Lesser Snakes Fitness) was not implemented.

The narrowest path fitness evaluation evaluates snakes in terms of how narrow their path is through the hypercube search space. This fitness function uses the logic of the NPH, which is explained in the next chapter.

Chapter 3

Traditional Search Techniques

3.1 Overview and Depth Limited Search

In addition to using a GA to search for snakes, this project uses Depth Limited Search (DLS) and a modified form of DLS called the Narrowest Path Heuristic (NPH). DLS is nothing more than Depth First Search (DFS), bounded by a specific depth. DLS and DFS are both uninformed. DFS and DLS are uninformed because they do not evaluate newly expanded nodes (the child nodes) and order them in any way before placing them into the search queue. The nodes are left in the order they are found (from the look up table) and placed at the front of the queue for expansion. Once the desired depth is reached or if the search space is exhausted then the search halts returning the best solution found. In the case of the Snake Problem, the specified depth is also the length of the snake. Using a DLS is intuitively obvious for the Snake Problem which is, by definition, a successful constrained search within a hypercube to a certain depth.

Both DLS and NPH keep a list of legal nodes in memory to which expansion can occur. This list of legal nodes must be updated at each step in the search. At each step in the search the list of legal nodes is the list of all possible nodes in the hypercube minus the first node of the snake and minus any nodes connected to those nodes that make up the current snake excepting the head. Thus, the list of valid child nodes to which expansion can occur will always be the set of nodes connected to the current head of the snake that are also contained in the set of legal nodes. Consider a DLS for a s(3)=4. Assume a starting node of zero.

```
Depth=0; Snake=[0]; Legals=[7,6,5,4,3,2,1] -- set of all possible
```

```
nodes minus the first node; ConnectionsToHead=[1,2,4];
ValidConnectionsToHead=[1,2,4].
```

Since this is a DLS, the first member of the set of valid connections is picked for expansion. In this case that node is [1]. This leaves nodes [2] and [4] as backtrack points, i.e., the search could come back and move to [2] and [4], if it hit a dead end.

```
Depth=1; Snake=[1,0]; Legals=[7,6,5,3] -- previous legals minus
Connections to 0; ConnectionsToHead=[0,3,5];
ValidConnectionsToHead=[3,5].

Depth=2; Snake=[3,1,0]; Legals=[7,6] -- previous legals minus
connections to 1; ConnectionsToHead=[2,1,7];
ValidConnectionsToHead=[7].

Depth=3; Snake=[7,3,1,0]; Legals=[6] -- previous legals minus
connections to 3; ConnectionsToHead=[6,5,3];
ValidConnectionsToHead=[6].

Depth=4; Goal Depth Reached -- End Search; Return
Snake=[6,7,3,1,0]; Legals=[]. ValidConnectionsToHead=[].
```

After performing run time comparisons, a list of legal nodes was kept instead of illegal nodes because it was faster. From the simple example above, it is seen that as the search progresses the set of legal nodes has progressively fewer members. At every move into a deeper level of the search, the list of legal nodes shrinks. The reverse is true for a list of illegal nodes. In order to avoid a move to an invalid node, one of these two lists must be searched and kept in memory. Since most of the search (especially in higher dimensions) occurs deep with the search tree, keeping a list of legal nodes is computationally cheaper.

Optimality is a property of searches whereby the search always terminates in the optimal path, if there is one [Russell95]. Note that this property is called admissibility by some authors [Luger98]. Given current knowledge of the Snake Problem, the best solution will always be the solution found given the depth in which to search. In the case of both DLS and NPH search, the longest snake is the snake found at the depth specified. And since an optimal depth can be specified and since all searches terminate, DLS and NPH are both optimal.

Due to the possibility of an infinite search space or an infinite loop within the space, DFS and DLS can be, for many problems, incomplete. As they are constructed in this project both DLS and the NPH are complete, i.e., they are both guaranteed to find a solution when there is one, and they halt searching when there is not. Both searches are finite. And because both searches keep a list legal nodes in memory, i.e., remove previously traversed nodes and their connectors, there is not the possibility of an infinite loop.

The results of both kinds of search, how they perform alone and in conjunction with the GA will be explored in greater detail in the "Results" section.

3.2 The Narrowest Path Heuristic

Heuristic search is the ordering of newly expanded nodes according to some guiding principle that is typically problem specific. The difference between DFS and the NPH is only in how the valid expanded nodes are ordered. DFS or DLS can be thought of as a blind or unintelligent search. The newly expanded valid nodes are not ordered and merely placed in the front of the queue. For the NPH search, the newly expanded child nodes are ordered by gathering information about the search space. This makes the NPH an intelligent search or a heuristic.

NPH looks ahead n number of moves in the search tree in an attempt to pick a more fruitful path. The following steps were used:

1. Construct a list of legal nodes from the current snake. The list of legal nodes is always the set of the total number of possible nodes (ranging from zero to $2^d - 1$) minus two sets: (A) the first node of the snake and (B) all of the connecting nodes to each of the nodes that makes up the current snake except for the head.

2. Retrieve a list of the nodes connected to the snake's head (the current node) from the look-up table.

3. Remove any nodes connected to the head, which are not in the current list of legal nodes. If the remaining list of valid nodes to the head is empty, then backtrack to the last tried

alternative. If the list of valid nodes to the head is non-empty (i.e., there are nodes in which to move), then proceed to the next step. Obtain a list of new legal nodes by removing any of the nodes connected to the head that exist in the list of legal nodes.

4. For each of the valid nodes connected to the snake's head sum up the number of valid moves that can be made from each node.

5. Order the nodes in the queue according to the sum, with the smallest non-zero sum to the greatest. If a dead end is ever reached (a path with zero possible move-points) in the search, then exclude this path from further search.

The path with the fewest number (greater than zero) of total possible move points (fewest legal nodes) is always picked as the place to move next. In this way the narrowest valid path is followed through the search space. As a working example, assume the following snake in dimension four in a search to Depth=7 that looks ahead one move in the search space:

```
Depth=3; Snake=[7,3,1,0]; LegalNodes=[15,14,13,12,10,6];
```

The look up table returns ConnectionsToHead=[15,6,3,5]. Removing those nodes not in the list of LegalNodes we are left with ValidConnections=[15,6]. DLS would pick the first node in this list to expand: 15. However, given the current snake of [7,3,1,0] this node leads to a path that will not terminate in a snake of length 7. Backtracking would need to occur to find the optimal snake.

The NPH heuristic will look ahead one move and order the nodes accordingly. Looking ahead one move from [15] we see there are two possible move points: [14,13]. This will assign the value of 2 to node [15]. Looking ahead one move for node [6] only allows for one move point: [14], thus node [6] will be assigned a value of one. Ordering the nodes from the lowest to the greatest value will place node [6] at the front of the queue and node [15] next. In this way, the node that has the least positive number of possible moves ahead of it will be expanded first. This causes the snake to compress within the hypercube. Generally speaking

with NPH a greater number of nodes remain in the list of legal nodes allowing the snake more room to grow.

The NPH performed as well as or better than DLS in all cases (of which there were hundreds tried) except for two. These anomalous cases will be shown in the "Results" chapter. Excepting these two cases, the NPH finds a snake of the desired length in the same number of steps or fewer than DLS. It takes slightly longer to run because it must perform additional computation to look ahead in the search space, but the computational time used is well worth the reduction in the number of states that have to be considered to find the snake of a desired length.

It was the NPH in conjunction with using maximal snakes that successfully beat the current records reported for s(9) and c(9).

3.3 Narrowest Path Heuristic and Maximal Snakes

Searching for snakes by building on maximal snakes was a method proposed relatively recently by Rajan and Shende [Rajan99] and explored further by [Scheidt01] and others [Kiranmayee01, Prasanna01]. A maximal snake is formally defined in the following way: a snake $S_d = (s_0, s_1, ..., s_k)$ is maximal in H^d if and only if, by definition, there is no vertex v in H^d such that $(v, s_0, ..., s_k)$ or $(s_0, ..., s_k, v)$ is a snake in H^d. Put informally, a snake is maximal, if and only if it can't be extended in either direction. Crucially, a maximal snake is not necessarily the longest snake. Rajan and Shende conjectured that for all d, there is a longest d snake that has a maximal (d-1)-snake as its initial segment. They later proved this conjecture wrong, but still obtained promising results. They currently hold the record for the longest snake in dimension eight at length 97, which they obtained via search from a maximal (and longest) snake in dimension-7 of length 46 [Rajan99].

It is through using their technique of "building on the shoulders of giants" by using maximal snakes in conjunction with DLS and the NPH that new lower bounds are discovered for c(9), and s(9).

Chapter 4

Combining the GA with NPH and DLS - Super-Individuals

4.1 Overview

Due to good results using the NPH and DLS, it is hypothesized that combining their strengths with the GA will produce better snakes than any of the three techniques alone. Growing longer snakes (and thus better individuals) is a technique that is a kind of seeding during run-time. This technique is called breeding super-individuals. This section explains how the GA was combined with DLS and the NPH to create super-individuals in evolving snakes.

The primary motivation with introducing super-individuals into the population was to improve GA performance. Using the same logic as is used with seeding, especially good individuals (or genetic material) is introduced into the population in hopes that it will contribute to a better overall population. As with seeding, the danger with this approach is that it will hamper genetic diversity and become a detriment.

There were two ways in which super-individuals were created and introduced into the population: (1) By extending *all* individuals to contain a maximal snake and (2) by running a DLS or the NPH on snakes in the population.

4.1.1 Maximal Snake Super-Individuals

Maximal Snake Super-Individuals (MSSIs) were created by taking the best snake in an individual and extending the best snake until it was maximal. Since the longest snake in an individual was already being extracted by the fitness function, the added computation to introduce MSSIs was minimal. The search for maximal snakes is very quick because the

search proceeds until a dead end and stops. There is no backtracking to untried paths in the search space. It is only desirable to search until the snake can be extended no more. This means that only s States need to be examined in the search where s is equal to the maximum snake length minus the best original snake length.

Because the overhead for creating maximum snakes was small, this function extended *all* snakes to their maximum when fitness was calculated. This meant that the entire population was made up of MSSIs at all times when the MSSI operator was used.

4.1.2 Searched Super-Individuals

Searched Super-Individuals (SSIs) differ from MSSIs in that a search is performed (either DLS or the NPH) on the best snake in an individual with backtracking. Once a maximal snake is found the search for SSIs continues to examine the search space for s states, returning the best snake found. This best snake is reintroduced into the population.

There were three settings relevant to combining the GA with DLS and the NPH: (1) the number of individuals to make into SSIs (either a set number or a percentage of the population), (2) the number of states to examine in the search space before stopping, and (3) how often in the GA run to attempt the SSI operator (every g Generations).

For one experiment the extension algorithm ran with both DLS and the NPH on 1% of the population searching 25,000 states every 10 generations. This means that every 10 generations 1% of the population was selected at random. The longest snake from each of the individuals was extracted from the genetic material. This longest snake was then run with either DLS or the NPH to examine 25,000 states in an attempt to grow the snake longer. If a longer snake was found, it was reintroduced into the population.

Chapter 5

Using Prolog

5.1 Overview

Prolog, which stands for "programming in logic," was created by Alain Colmerauer and his colleagues in 1972. It now ranks with LISP as a mainstay in Artificial Intelligence programming languages. Prolog is a high level language typically used for expert systems, natural language processing, general search, and for any problem domain that is easily modeled with logic [Covington94, Covington97, Sterling94]. Thus, Prolog is used in some atypical ways. Using the Prolog programming language (in LPA Prolog 4.x) to attack the Snake Problem with a GA and traditional search techniques was incredibly useful, but not without some drawbacks.

Generally speaking Prolog has some distinct advantages over other programming languages. Prolog has been around for a relatively long time and is highly stable with a well established syntax. Prolog is declarative — not procedural. This allows the programmer to concentrate more on the logic of program and the problem domain itself and spend less time worrying about implementation. Take for example a simple traditional genetic algorithm programmed in Prolog. A simple GA executes the selection, crossover, and mutation operators on a population for g generations. This can be expressed at a high level with the following recursive Prolog program, which assumes that the population and GA settings are stored elsewhere in memory:

```
ga(0):-!.

ga(Generation):-
```

```
selection,
crossover,
mutation,
NextGeneration is Generation - 1,
ga(NextGeneration).
```

In this simple case and at the highest level, a Prolog implementation of a GA is logical and succinct.

Prolog makes use of pattern matching and list processing efficiently and can handle complex data structures elegantly. A GA population can be represented as a list of individuals. Where:

```
Population = [Individual1,Individual2, ... , IndividualN]
```

An individual is a list with two members: the fitness and a list of genes that make up the individual.

```
Individual = [Fitness, [Gene1, Gene2, ... , GeneN]]
```

Such a generalized schema for GA populations is adaptable, robust, and simple.

Because things like variable declaration and memory management are handled on-the-fly by the Prolog engine, code development is faster and easier than it would be for a language where variable types needed to be declared. Avoiding variable declaration is a double edged sword. Since Prolog must interpret the variable types during run time, there can be significant computational time lost. This leads to slower running programs. And with both GAs and traditional searches efficiency is crucial. This is a downside to using Prolog.

On the upside, the Prolog inference engine uses depth first search by default. Notwithstanding some time lost due to run-time variable interpretation, Prolog's DFS is fast and efficient. And since this is its default search technique, it allowed for very fast modification of DFS, DLS, and the NPH.

Chapter 6

Results: Genetic Algorithm, Narrowest Path Heuristic, and Depth Limited Search

6.1 Overview

The results for the NPH and the GA were mixed. The GA by itself performed poorly overall and the NPH performed much better than expected. Maximum lengths (maximum lower bounds or conclusive upper bounds) have been established by exhaustive search for s(d) and c(d), where $d < 8$ (see table 6.1).

Dimension d	Max s(d)	Max c(d)
2	2	4
3	4	6
4	7	8
5	13	14
6	26	26
7	50	48

Table 6.1: Optimal Snakes and Coils Previously established by exhaustive search

Dimensions greater than seven must be explored to establish new lower bounds. This project explored dimensions 8–10 using the GA, DLS, and the NPH. New lower bounds for c(9), and s(9) were found using maximal snakes and both DLS and the NPH. They are presented with analysis below.

6.2 GA Results

Before running tests that combine the GA with traditional search good GA settings needed to be established. This section reports on findings that pertain to GA runs only.

The permutations of all possible GA parameters were quite large, so it was necessary to apply restrictions by isolating certain independent variables. Unless otherwise noted, results are presented for GA searches for s(8) with populations of 500 ran for 500 generations. And because the EER crossover performed so poorly with node representation, this project focuses on data from transition representation. For s(8) the following sets of GA operators are compared: (1) Guided Random Initial Population (GRIP). (2) Selection: roulette wheel, tournament and rank. (3) Crossover: one, two, and three point, as well as enhanced edge recombination. (4) Mutation: blind and guided. (5) Catastrophe. (6) Elitism. In all cases, at least 10 separate runs were performed to test each independent variable.

In all cases and given a static set of GA parameters, there are three results reported in the subsequent tables: (1) The best snake from each GA run was reported and the average of these best snakes results in the value reported in the emphAve Best Snake Found column. (2) The best snake found among *all* runs is reported in the emphBest Snake Found column. (3) The standard deviation of the best snake found from all runs is given in the *StdDev* column. This standard deviation is the deviation from the average reported in the emphAve Best Snake Found.

6.2.1 Initial Population Creation Results

This section shows the effects of the Guided Random Initial Population (GRIP) operator on GA performance by comparing GRIP results to a population created completely at random.

All GA settings were kept static for both runs. The only thing changed was the way in which the initial population was created. The averages and data in the table below are taken from 30 runs for s(8) with a population of 500 (see table 6.2).

Creation Operator Used	Ave Best Snake Found	Best Snake Found	StdDev
Random	56.10	61	3.76
GRIP	57.70	67	4.27

Table 6.2: Initial Population Operator Results

These results show that using the GRIP operator provided marginal improvement (of 1.6) over random initial population creation in the average overall best snakes found. However, the best snake found among all runs using GRIP was significantly better than the best snake found using the completely random initial population operator. This indicates that the GRIP operator generally helps in the search for snakes using a GA. The standard deviation shows that the outcome is not conclusive. Regardless, the GRIP did improve performance in these tests and was used for all other GA test runs.

6.2.2 Selection Results

Three types of selection were tested: tournament, roulette wheel, and rank. Note that tournament selection was tested by comparing two and three individuals picked at random. The dominant individual was given a 90% chance of being picked. With the exception of changing the selection method, 40 runs were made for each with identical settings (see table 6.3).

Selection	Ave Best Snake	Best Snake Found	StdDev
2-Tourney	58.9	63	2.56
3-Tourney	57.6	61	3.10
Roulette	58.1	66	3.44
Rank	58.4	66	3.88

Table 6.3: Selection Method Results

While 2-tournament selection performed the best in terms of producing better snakes on average, the standard deviation shows that all of the averages are close. Both Roulette Wheel and Rank selection produced a better snake overall than tournament selection. These results show that there is no clear winner with respect to comparing these three selection methods, but Roulette Wheel and Rank may produce better snakes.

6.2.3 Crossover Results

Two kinds of crossover were used in this project: n-Point (for which 1, 2, and 3 point crossover was tested) and Enhanced Edge Recombination (EER). As was mentioned before, due to

the setup of EER as proposed by [Starkweather91], it performed very badly. Thus, with the exception of this test, this project used transition sequence representation and n-point crossover in all other trials.

Thirty runs were made for one, two, and three point crossover, and EER with varying crossover rates of 30, 40, and 80% set between runs (see table 6.4).

Xover	Ave Best Snake	Best Snake Found	StdDev
EER	46.3	53	3.54
1-point	58.6	69	3.96
2-point	58.5	65	4.38
3-point	57.4	66	3.91

Table 6.4: Crossover Method Results

EER took the longest to run (primarily due to table modification) and clearly performed the worst. Conclusively showing that n-point crossover is the superior method of crossover. The poor performance of EER was anticipated given the implementation problems stemming from how individuals are constructed in this project. One-point crossover performed the best on all counts producing the best snakes on average as well as a snake of length 69.

6.2.4 Mutation

Random mutation and guided mutation are compared. Running 70 trials with populations of 500 for 500 generations showed that guided mutation performed marginally better than just random mutation by producing slightly better snakes on average. However, Random mutation found a snake of length 66, which beat the best snake found by guided mutation (see table 6.5).

Op	Ave Best Snake Found	Best Snake Found	StdDev
Random Mut	53.3	66	3.42
Guided Mut	54.1	63	3.80

Table 6.5: Random vs. Guided Mutation

By only improving on the average by 0.8, smart mutation is not shown to be any better or worse than a completely random mutation especially given the standard deviations. Smart

mutation may lead to slightly better snakes overall, but random mutation may allow for a degree of diversity in genetic material that will lead to occasional best snakes superior to what guided mutation could produce.

The data clearly show that mutation of some sort leads to better snakes. Consider table 6.6, which used smart mutation and ran 10 times for each mutation percentage.

Mut Rate	Ave Best Snake	Best Snake Found	StdDev
0%	46.0	50	3.30
0.1%	52.2	57	3.91
0.2%	54.8	62	3.68
0.5%	57.3	62	3.33
0.8%	59.0	63	4.35
1%	59.4	63	1.84
2%	49.0	56	3.56
4%	46.9	50	2.64

Table 6.6: Guided Mutation Rate Results

Using no mutation leads to poor overall snakes. This is no surprise given the history of GAs and the traditional use of mutation. But the question remained as to which mutation rate worked best with the Snake Problem. These data show that a mutation rate of around 1% proved best. Thus, a mutation rate of around 1% was used for future GA runs.

6.2.5 Catastrophe

Thirty identical runs were made with and without the catastrophe operator active. A catastrophe was set to destroy 90% of the population after 12 generations of a less than 2% variance in the average fitness (see table 6.7).

Op	Ave Best Snake Found	Best Snake Found	StdDev
No-Catastrophe	59.07	63	2.19
Catastrophe	57.70	62	3.21

Table 6.7: Effects of Catastrophe

Using the catastrophe operator clearly did not improve GA performance, and looks to quite possibly be hurting performance, but the results are still too close to call. It should

also be noted that in monitoring GA performance there was rarely an improvement in the best snake after catastrophe. Thus, catastrophe seems to not aid in GA performance with the Snake Problem.

6.2.6 Fitness Functions

There were three fitness functions used in this project: (Fit-1) the cube of the longest snake borrowed from [Potter94], (Fit-2) the cube of the longest snake plus the sum of the squared shorter snakes, and (Fit-3) an evaluation of snakes based on the narrowness of their path through the hypercube. All settings were left the same for all runs excepting the fitness function. Only rank and roulette wheel selection were used because they better account for gradation and give each individual a proportional chance of survival based on fitness. Forty runs were made with each fitness function to obtain a good overall sense of performance (see table 6.8).

Op	Ave Best Snake Found	Best Snake Found	StdDev
Fit-1	58.0	66	3.42
Fit-2	57.9	66	3.83
Fit-3	57.9	62	3.06

Table 6.8: Comparison of Fitness Functions

There is no noticeable benefit to using one fitness function over another. The results are remarkably close for the average best snake found. Fit-3 might hurt the possibility of outstanding individuals from being formed by weighting the individuals inappropriately. Fit-3 maybe undervaluing especially good individuals. It does generally give individuals a lower fitness than the other two fitness functions.

6.2.7 Seeding

The results from seeding were very promising, and the hypothesis that the more of the population that is seeded the better the snakes found proved to be true. As is seen so far the GA alone with no special help returns snakes of rather short length. As will be seen in the

next section, the DLS or NPH consistently outperform the GA alone. The GA needs some help. Seeding is one way to help.

Table 6.9 shows the results from seeding some percentage of the population during the initial population creation. In all cases a random search was performed using the Random Snake Generator for 2,000 states to create a seeded population. This search organized the newly expanded nodes randomly before placing them in the queue, and then picked the first node from the queue.

% Pop Seeded	Ave Best Snake Found	Best Snake Found	StdDev
1.25%	70.6	75	2.50
2.5%	71.9	77	2.18
5%	72.1	78	1.71
10%	72.6	75	2.60
20%	74.4	78	1.50
40%	74.8	77	1.23
80%	75.6	78	1.50
100%	75.5	76	0.71

Table 6.9: Seeding Results

As was hypothesized, the average best snake found steadily increased as more of the population was seeded with good individuals. One exception is the jump from seeding 80% of the population to 100%. Increasing seeding at this level resulted in a slight drop in GA performance. The average best snake found dropped by 0.1, and the best snake found overall fell by a length of two. It might be the case that the population benefits from a degree of genetic diversity lost by seeding *every* individual.

Seeding significantly improved GA performance with the Snake Problem. Also, of note is a lower standard deviation in the best snakes found (as compared to previous trials). This indicates a greater consistency in results.

6.3 Depth Limited Search and Narrowest Path Heuristic

The results for DLS and NPH alone were much more promising than those for the GA alone. Below are various results for searches using DLS and the NPH. The first column in the

following tables 6.10 through 6.16 gives the set depth (the length of the snake) at which search was attempted from a starting node of zero. The second and third columns give the number of states examined before a solution was found for DLS and NPH respectively. There are two and only two known values in the NPH column (highlighted by double asterisks) that performed worse than DLS.

Many of the searches were stopped due to time constraints, i.e., it was taking too long to find a solution at the desired depth. When search was terminated before a snake of the desired length was found, the number of states examined are reported (indicated by a plus sign). If the column is blank, the number of states to find a snake at the length specified is unknown and search was not attempted.

Snake Length	DLS States	NPH States
6	6	6
7 max	7	7

Table 6.10: DLS vs NPH — Results for s(4)

Snake Length	DLS	NPH
11	11	11
12	12	12
13 max	228	204

Table 6.11: DLS vs NPH — Results for s(5)

Snake Length	DLS	NPH
21	21	21
22	41	22
23	42	23
24	65	24
25	4,091	67, 203**
26 max	1,257,290	749,336

Table 6.12: DLS vs NPH — Results for s(6)

Snake Length	DLS	NPH
39	40	39
40	49	40
41	264	41
42	441	42
43	960	43
44	53,699	51
45	152,850	52
46	4,636,365	465
47	9,885,097	502
48	132,434,982	698, 461, 143+**
49		

Table 6.13: DLS vs NPH — Results for s(7)

Snake Length	DLS	NPH
70	662	70
71	972	71
72	1,042	72
73	2,147	73
74	2,164	74
75	2,165	75
76	17,176	76
77	17,177	79
78	132,911	203
79	132,912	4,324
80	1,234,673	9,426
81	4,619,851	9,515
82	6,249,356	449,239
83	41,751,926	449,240
84	41,884,523	5,957,827
85	$850,000,000+$	5,957,828
86		12,784,930
87		29,711,206
88		843,571,363
89		

Table 6.14: DLS vs NPH — Results for s(8)

Snake Length	DLS	NPH
132	138	137
133	147	140
134	6,093	141
135	6,124	142
136	6,125	143
137	497,437	144
138	1,552,955	145
139	1,552,986	146
140	1,552,987	147
141	2,057,146	148
142	2,308,698	149
143	2,812,857	231
144	16,375,897	232
145	16,375,906	233
146	$104,620,516+$	43,231
147		43,232
148		146,330
149		480,044
150		561,217
151		575,249
152		1,340,445
153		1,340,446
154		3,009,101
155		3,009,102
156		$632,464,332+$
157		

Table 6.15: DLS vs NPH — Results for s(9)

Using the NPH does require more computational time to look ahead in the search space (to determine the narrowest path). The number of states each method can examine per second varies on the dimension of the hypercube and the depth at which the search is taking place (there is more backtracking done when deeper in the search tree). Given the same amount of time to run both DLS and NPH to the same depth and in the same hypercube, the NPH is able to examine about 80% of the states DLS can. But as is plainly seen in the tables above, the tradeoff is, more often than not, well worth it.

This many data are reported for three reasons. First, to show that the NPH is not perfect. It does perform as good or better than DLS in all known cases except for two. These two anomalous cases (the snakes found) have been examined, but a determination as to why DLS performs better than the NPH is unknown. Second, the NPH is good and seems to perform better as the dimensions increase. Finally, all of these data show and reflect (albeit in a rudimentary way) the wily nature of the search space. There seem to be paths in the hypercube that are good for a while and then dead-end having to backtrack a large number of times to find a longer snake. Compare searching s(10) in table 6.16 using NPH to a depth of one to 275 from a starting node of zero. Solutions are found examining less than 14,000 states, but to find a snake of length 276 — 8,321,179 need to be examined.

6.4 Using Maximal Snakes and Narrowest Path Heuristic

To further test the efficacy of the NPH, it was tried with good maximal snakes. It was conjectured that using a maximal snake from s(8) of length 97 as a base (this base was found by [Rajan99, Rickabaugh99] using a maximal snake of length 46 from s(7)), the heuristic would find a good snake in s(9). This conjecture turned out to be correct. This method resulted in snakes and coils, which beat the current records reported in the literature of s(9)=168 and c(9)=170 [Abbott91, Paterson98]. DLS beat the record of s(9)=168, but the NPH heuristic performed significantly better than DLS. The data in table 6.17 further adds to the evidence that the NPH performs well by comparing runs using the same maximal

snake 97=s(8). The first column shows the desired length of the snake. The second and third columns report the number of states examined before a snake of the desired length was found for DLS and the NPH respectively.

Of particular note in table 6.17 is searching for a snake of length 173. The DLS was unable to find a snake of this length after searching more than 3.6 billion states, whereas the NPH found one in only 85,630 states and in less than five seconds. The NPH went on to significantly improve on the current lower bound of 170. A new lower bound of 174=c(9) was obtained using NPH and the same maximal snake of 97.

6.5 GA Combined with DLS and NPH

Compared with DLS and the NPH the GA performed poorly when evolving solutions from scratch. The GRIP operator along with tweaking various GA settings gave a small boost to overall fitness and found better snakes, but not enough to make the GA (by itself) a good way to search for snakes. To obtain good snakes with a GA, seeding needed to be used or super-individuals needed to be introduced into the population via maximal snake super individuals (MSSIs) or by search for super-individuals (SSIs) using DLS or NPH. Using either a DLS or a NPH alone quickly found snakes better than those found by the GA alone. This section examines combining the GA, maximal snakes, DLS, and NPH to search for snakes.

The following tables in this section all reflect data from GA runs with populations of 300 for 300 generations, unless otherwise noted. These are shorter runs than were used for general GA testing, but they yielded much better results. This should show the power of combining a GA with a traditional search. One-point crossover performed significantly better than two point when combining the GA with traditional search methods.

6.5.1 Searched Super-Individuals (SSIs)

Below are the results for introducing SSIs into the population. In all cases some percentage of the population was picked at random every 10 generations. The best snakes were extracted

from the genetic material of each of these snakes and either DLS or the NPH ran in an attempt to extend the snake by examining 25,000 states (see table 6.18).

The efficacy of the NPH and combining a GA with the NPH is shown once again. In *all* comparable cases the average best snake found using NPH to extend snakes is better than the comparable average using DLS. In *all* cases the best snake found overall by the NPH was as good as or better than those found by DLS with comparable settings. A snake of length 91 was found using NPH and the GA. s(8)=91 beats the best snake found by [Potter94] of s(8)=89 using a GA. Also of note is that these data come from runs with much smaller populations (300) and significantly fewer generations (300) than those used by [Potter94]. Potter et al. used population sizes ranging from 10,000 to 25,000 and ran them for around 50,000 generations. Furthermore their project seeded the population with the best individuals found from previous GA runs. Using super-individuals can be thought of as a form of seeding, but it is done during run time so results are obtained without any outside or predefined information. In this sense results are evolved from scratch.

It is also important to note that combining the GA with traditional search beat the best results found by searching for snakes from scratch using either DLS or NPH alone. It took the NPH about a half-day on a computer with a 1.4 GHz AMD Athon processor considering over 843,571,363 states to find s(8)=88. The GA-NPH combination found s(8)=91 in less than 30 minutes.

6.5.2 Maximal Snake Super-Individuals (MSSIs)

The best results found so far by combining a GA with traditional search methods were only rivaled by using the MSSI technique. During fitness evaluation (which must extract the longest snake in an individual to determine fitness) the MSSI technique extended the longest snake until it became maximal using both DLS and NPH. The results are presented below in table 6.19:

This technique of using MSSIs produced the highest best snake average among all previous trials at 88.8 using the NPH. And once again the NPH proved to be superior to DLS even when combined with a GA. With its especially low standard deviation among the best snakes found, this method proves to be efficient and reliable for evolving good snakes.

And finally, table 6.20 reports the results from combining a Genetic Algorithm, Searched Super Individuals, Maximal Snake Super Individuals, and the Narrowest Path Heuristic. These results report on 10 runs with 300 individuals for 300 generations, where every snake was made maximal upon fitness checking and 4% of the population was extended for 25,000 states every 10 generations.

Using all of the techniques together produced the highest average best snake found as well as the best snake found of length 92 overall among all previously tried methods. The best snake *average* is higher than the best snake found by [Potter94], and is the best snake found overall using a genetic algorithm among all testing done in this project.

Snake Length	DLS	NPH
235	398	237
236	405	238
237	406	241
238	407	242
239	1,711	242
240	1,712	243
241	18,334	245
242	18,335	246
243	274,285	247
244	274,286	248
245	11,802,537	249
246	11,802,538	250
247		251
248		254
249		255
250		256
251		257
252		258
253		259
254		260
255		261
256		262
257		263
258		264
259		265
260		266
261		267
262		268
263		278
264		279
265		282
266		283
267		284
268		291
269		292
270		293
271		310
272		311
273		3,632
274		5,845
275		13,153
276		8,321,179
277		

Table 6.16: DLS vs NPH — Results for s(10)

Snake Length	DLS	NPH
155	80	65
156	81	66
157	534	67
158	538	68
159	905	69
160	909	72
161	1,508	73
162	2,428	74
163	2,429	75
164	20,072	76
165	24,149	77
166	26,060	78
167	26,061	79
168	297,442	82
169	352,367	83
170	6,945,534	1,277
171	6,961,687	85,472
172	6,962,518	85,473
173	$3,600,000,000+$	85,630
174		85,631
175		845,484
176		1,215,394
177		1,396,464
178		4,800,254
179		1,396,490
180		4,800,254
181		8,799,607
182		4,233,104,576
183		

Table 6.17: DLS and NPH Results Searching s(9) Using a Maximal Snake from s(8)

Search	% Pop To Extend	Ave Best Snake	Best Snake	StdDev
DLS	1%	80.2	83	2.04
DLS	2%	81.6	86	2.22
DLS	4%	81.7	85	1.77
DLS	8%	85.7	88	1.89
NPH	1%	82.1	85	1.20
NPH	2%	83.4	86	1.71
NPH	4%	85.6	91	2.27
NPH	8%	86.4	88	1.17

Table 6.18: GA with DLS and NPH — SSIs — Every 10 Gens for 25,000 States

MaxSnake Generator	Ave Best Snake	Best Snake	StdDev
DLS	85.5	90	2.50
NPH	88.8	90	0.63

Table 6.19: Extending All Snakes to be Maximal — DLS and NPH

Ave Best Snake	Best Snake	StdDev
89.6	92	0.51

Table 6.20: MSSIs, SSIs, and the NPH with a GA

Chapter 7

Conclusion and Future Direction

7.1 General Attacks on the Snake Problem

The Snake Problem is a wily one. Its search space is unpredictable and combinatorially explosive. The Narrowest Path Heuristic generally and significantly improves upon Depth Limited Search, which was previously the only known way to search for snakes using traditional methods. Barring a mathematical discovery with the Snake Problem and given current computer processing speed, it is unlikely that the longest snake for $d > 7$ will be verified any time soon. As the search space grows at higher hypercube dimensions, it is increasingly unlikely that traditional search methods will produce increasingly good snakes.

The results from this project show promise in hybrid techniques for attacking the Snake Problem. Non-traditional search techniques like a Genetic Algorithm do not promise optimal results. A GA is neither optimal nor complete. Traditional search methods can be both optimal and complete. Combining the exploratory power of GAs with more steadfast traditional search has resulted in better snakes when searching from scratch.

The use of maximal snakes proved to also be quite useful. While this project did not engage in a special mathematical study of snakes, it used the maximal snake conjecture of [Rajan99] to beat the current lower bounds for s(9) and c(9). This indicates that further study of the Snake Problem itself and the nature of snakes as mathematical entities in graph theory will be required for future advances.

The NPH might be improved greatly by a more detailed study of the search space. Some form of iterative deepening where the search space is mined for information might be helpful as well. Examining known long or maximal snakes in lower dimensions should shed light on

what properties good snakes exhibit. Discovering what properties are good can greatly aid in heuristic development.

The great thing about combining a GA with traditional methods is that *any* traditional method can be plugged into the GA to create super-individuals during run time.

7.2 GA Enhancements

Due to the large number of permutations in setting GA parameters, there are many more variables with the GA settings that went untested. Since no selection method proved to be better than any other, it might help to experiment with alternating between methods during a GA run. Various selection methods might be used with some probability when the selection function is called at each generation. It might be the case that one selection method works better than another at different stages in the GA search.

This project did not experiment with seeding the population (either from the get-go or during run-time) with really good previously established snakes. The evidence from seeding with super-individuals suggests that an outright seeding with known good snakes would benefit the search space greatly.

One might also use a maximal snake as an unchanging base from which to evolve add-on material. That is, only evolve parts of snakes to add on to previously established good segments.

Prolog is probably not the best language to use for coding a GA. A procedural language that can be better and more easily optimized (like C) would almost certainly result in a faster running GA. Also, some of the techniques used in this paper maybe used with a commercial GA product to reduce configuration/coding time.

Enhanced Edge Recombination performed terribly with the Snake Problem and node representation. With some modification, it can likely be made to work properly. This might improve GA results.

In future research the attempt should be made to strike the best balance possible between exploration and exploitation. A two-fold approach where the Snake Problem itself is studied while experiments on methods (like GAs) to attack the problem are carried out might be the best way to go.

Appendix A

General Code

A.1 Prolog Comments and Snake Connectivity Tables

This appendix will list much of the Prolog code used in this project. The notation and commenting style as proposed by Covington et al. [Covington97:114]. The Prolog language consists of facts and rules which take a set of arguments like so:

```
predicate(Arg1,Arg2,...,ArgN)
```

Arguments meant to be instantiated (supplied with an input value when executed) will be denoted with a plus sign: '+.' Uninstantiated arguments (variable arguments without values) meant to be supplied with a value by execution will be denoted by a minus sign: '−.' Such arguments can be considered output values. Arguments that are meant to be instantiated or uninstantiated (may or may not have a value coming in or going out) are denoted by the question mark: '?.' Ambiguous arguments, such as these, may be inconsequential to program execution depending on the particular function of predicate at a particular time.

Databases can be created in Prolog as Prolog facts. Facts generally contain atomic or axiomatic values with respect to the system or program. To define node adjacency such a Prolog database was created for each hypercube. An example of such a database is shown below for a cube of dimension three:

```
% c(+Node,-Connections)
% 3-Cube Connectivity Table in Prolog
c(0,[1,2,4]).
c(1,[0,3,5]).
```

```
c(2,[3,0,6]).
c(3,[2,1,7]).
c(4,[5,6,0]).
c(5,[4,7,1]).
c(6,[7,4,2]).
c(7,[6,5,3]).
```

The first argument represents all possible vertices for an d Cube. The second argument is a list of the nodes connected to the vertex defined in the first argument. Such connectivity tables were used for all search techniques used in this project. Because LPA Prolog has built-in first argument indexing and because each vertex is unique with respect to cubes, this hypercube representation was fast (for lookups) and easy to program. A separate program was written to automate the creation of these tables.

As might be imagined, these tables grow quite large as the dimension of the hypercube increases. Only the table for dimension three is supplied for brevity, as the logical extension of this table to higher dimensions should be obvious.

Appendix B

GA Specific Code

B.1 General GA Code

The amount of Prolog code associated with the GA in this project is quite large. As a result only the most general GA predicates are listed.

```
/*********************************************************************
run_snake_ga(+Generations,+Population)
   Executes the GA for the Snake Problem. All relevant files for
   the GA settings are stored in memory elsewhere and should be loaded.
   Population = [[Fitness1,Ind1], ... ,[FitnessN,IndN]]
   Individual = [Gene1, ... ,GeneN]
*********************************************************************/
run_snake_ga(0,Pop):-
    !,
    info(0,Pop).            % Prints out data on whole GA run.

run_snake_ga(G,Pop):-

    fit(Pop,FitPop),        % Establishes fitness for current Pop.

    info(G,FitPop),         % Reports on GA performance.

    elite(FitPop),          % Saves best N individuals.

    catast(FitPop,CPop,G),  % Catastrophe operator called.

    sel(CPop,SelPop),       % Takes CPop and selects SelPop.

    xover(SelPop,XoverPop), % Crosses over SelPop to XoverPop.

    mut(XoverPop,MutPop),   % Mutates XoverPop into MutPop.

    spec(MutPop,SpecPop,G), % Performs any special operations, e.g.,
                            % creating super-individuals.
```

```
    !,
    NewG is G-1,
    run_snake_ga(NewG,SpecPop).

run_snake_ga(_,_):-
    write('GA Failed!'), nl.
```

Appendix C

Traditional Search Code

C.1 DLS and NPH

The code for the Depth Limited as well as the Narrowest Path Heuristic searches is below. Some of the predicates used in this algorithm are not defined below as their function should be obvious from the predicate names.

```
/*********************************************************************
snakeSearch(+DCube,+DesiredSnakeLength,+Heuristic,+InitialSnake,-NewSnake)
    Fails if InitialSnake is not a valid snake or if there is no
    snake that exists at the DesiredSnakeLength.
    If Heuristic = 0, then a DLS is performed.
    If Heuristic = 1, then NPH is used.
*********************************************************************/
snakeSearch(DCube,DesiredLen,Heur,IniSnake,NewSnake):-
    load_lookup_table(DCube),
    getLegals(IniSnake,CurLen,LegalNodes),
    NewDesiredLen is DesiredLen - CurLen,
    snakeSearch_1(NewDesiredLen,LegalNodes,Heur,IniSnake,NewSnake).

snakeSearch_1(0,_,_,NewSnake,NewSnake):-!.
snakeSearch_1(LenLeft,Legals,Heur,[SnakeHead|T],NewSnake):-
    c(SnakeHead,ConnsToHead),
    membersInCommon(ConnsToHead,Legals,ValidConns),
    removeAll(ValidConns,Legals,NewLegals),

    orderMoves(Heur,NewLegals,ValidConns,ValidConnsOrdered),
    nextMove(ValidConnsOrdered,NextMove),

    NewLenLeft is LenLeft - 1,
    snakeSearch_1(NewLenLeft,NewLegals,Heur,[NextMove,SnakeHead|T],NewSnake).


/*********************************************************************
orderMoves(+Heuristic,+LegalNodes,+ValidConnections,-Ordered)
```

```
    Orders moves if the Heuristic is set to 1.
**********************************************************************/
orderMoves(0,_,OrderedMoves,OrderedMoves).
orderMoves(1,Legals,ValidConns,OrderedMoves):-
    evalMoves(ValidConns,Legals,EvaledConns),
    orderEvaledConns(EvaledConns,OrderedMoves).


/**********************************************************************
evalMoves(+ValidConns,+LegalsNodes,-EvaledConns)
    Evaluates each connection to the current snake head.
    Returns a list of lists. Each sublist is a connection-value
    pair: EvaledConns = [[Conn1,Val1],...,[ConnN,ValN]].
**********************************************************************/
evalMoves([],_,[]).
evalMoves([H|T],LegalNodes,[[H,Narrowness]|TEvaledConns]):-
    c(H,Conns),
    cntMembersInCommon(Conns,LegalNodes,MemsInCommon),
    MemsInCommon \= 0, % This removes deadends.
    !,
    Narrowness = MemsInCommon,
    evalMoves(T,LegalNodes,TEvaledConns).
evalMoves([_|T],LegalNodes,EvaledConns):-
    evalMoves(T,LegalNodes,EvaledConns).


/**********************************************************************
orderEvaledConns(+EvaledConns,-OrderedMoves)
    Orders the evaluated conns in terms of narrowness. This means
    that the best node is the one with the least number of nodes
    to move to (greater than 0). Throw out zero because this is an
    obvious dead end.
**********************************************************************/
orderEvaledConns([],[]).
orderEvaledConns(EvaledConns,[Best|OrderedMoves]):-
    [CurBest|TEvaledConns] = EvaledConns,
    getBestConn(TEvaledConns,RestEvaledConns,CurBest,Best),
    orderEvaledConns(RestEvaledConns,OrderedMoves).


/**********************************************************************
getBestConn(+EvaledConns,-OrderedMoves)
    Gets the best connection.
**********************************************************************/
getBestConn([],[],[Best,_Val],Best).
getBestConn([NewBest|T],[CurBest|RestEvaledConns],CurBest,Best):-
    [_,CurBestVal] = CurBest,
```

```
    [_,NewBestVal] = NewBest,
    NewBestVal < CurBestVal,
    !,
    getBestConn(T,RestEvaledConns,NewBest,Best).
getBestConn([NotBetter|T],[NotBetter|RestEvaledConns],CurBest,Best):-
    getBestConn(T,RestEvaledConns,CurBest,Best).


/**********************************************************************
nextMove(+ListOfValidConns,-NextMove)
    This is where the backtracking to other possible nodes occurs.
**********************************************************************/
nextMove([NextMove|_],NextMove).

nextMove([_|T],NextMove):-
    nextMove(T,NextMove).
```

Appendix D

Snakes and Coils Found

D.1 New Lower Bounds

The following new lower bounds beat the best snakes and coils found in the literature for s(9)=168 and c(9)=170 [Abbott91]. They are listed below as transition sequence as this is the more general form of snakes. Note that both the snake and coil have the exact same sequence as their first 97 edges. The first 97 edges also constitute a long and maximal s(8) and its terminus is denoted by a double asterisk.

New Lower Bound of s(9)=182:

```
[1,2,3,4,5,6,4,3,2,1,4,3,6,4,2,3,4,5,6,3,4,2,1,4,3,7,1,6,5,4,1,2,
4,6,1,3,4,2,1,4,6,5,4,1,2,3,6,1,3,4,8,7,4,3,1,2,4,6,5,4,1,2,4,6,1,
3,4,2,1,4,6,5,4,1,2,4,7,3,5,4,6,1,2,4,6,5,4,2,3,5,2,1,4,6,5,4,1**,
9,6,1,2,3,6,1,4,2,1,6,5,4,1,2,4,3,5,7,4,5,2,1,4,2,3,4,6,2,4,1,6,5,
4,6,8,2,1,6,3,2,1,4,5,6,4,1,2,4,3,1,6,4,2,1,4,5,6,3,8,2,7,8,4,2,3,
6,1,2,4,1,5,4,3,2,4,6,3,1,4,2,3,8,1,2]
```

New Lower Bound of c(9)=174:

```
[1,2,3,4,5,6,4,3,2,1,4,3,6,4,2,3,4,5,6,3,4,2,1,4,3,7,1,6,5,4,1,2,
4,6,1,3,4,2,1,4,6,5,4,1,2,3,6,1,3,4,8,7,4,3,1,2,4,6,5,4,1,2,4,6,1,
3,4,2,1,4,6,5,4,1,2,4,7,3,5,4,6,1,2,4,6,5,4,2,3,5,2,1,4,6,5,4,1**,
9,6,1,2,3,6,1,4,2,1,6,5,4,1,2,4,3,5,7,4,3,2,4,5,6,1,4,2,1,3,7,8,3,
1,6,3,2,4,6,3,1,6,5,2,1,6,3,4,2,3,1,7,2,4,5,1,3,4,2,1,6,8,1,2,4,3,
5,6,3,2,6,8,2,4,3,2]
```

D.2 More Long Snakes from s(8)

The current longest snake known in s(8) is of length 97 [Rajan99]. This project applied the same logic used in finding new long snakes in s(9) and c(9) to s(8) and c(8). A search was

performed using the NPH on an initial maximal snake of length 45 from s(7). While this method did not beat the record, it did find 17 new snakes not reported in [Rajan99], where s(8)=97. The first snake listed below is that found by [Rajan99].

```
[1,2,3,4,5,6,4,3,2,1,4,3,6,4,2,3,4,5,6,3,4,2,1,4,3,7,1,6,5,4,1,2,
4,6,1,3,4,2,1,4,6,5,4,1,2**,3,6,1,3,4,8,7,4,3,1,2,4,6,5,4,1,2,4,6,
1,3,4,2,1,4,6,5,4,1,2,4,7,3,5,4,6,1,2,4,6,5,4,2,3,5,2,1,4,6,5,4,2]

[1,2,3,4,5,6,4,3,2,1,4,3,6,4,2,3,4,5,6,3,4,2,1,4,3,7,1,6,5,4,1,2,
4,6,1,3,4,2,1,4,6,5,4,1,2**,3,6,1,3,4,8,3,1,6,3,5,2,4,1,2,6,3,1,2,
4,1,6,5,4,6,3,1,4,7,3,6,5,4,1,2,4,6,1,3,4,2,1,4,6,5,4,1,2,3,6,1,3]

[1,2,3,4,5,6,4,3,2,1,4,3,6,4,2,3,4,5,6,3,4,2,1,4,3,7,1,6,5,4,1,2,
4,6,1,3,4,2,1,4,6,5,4,1,2**,3,6,1,3,4,8,3,1,6,3,5,2,4,1,2,6,3,1,2,
4,1,6,5,4,6,7,2,1,6,3,2,1,4,5,6,4,1,2,4,3,1,6,4,2,1,4,5,6,3,7,4,2]

[1,2,3,4,5,6,4,3,2,1,4,3,6,4,2,3,4,5,6,3,4,2,1,4,3,7,1,6,5,4,1,2,
4,6,1,3,4,2,1,4,6,5,4,1,2**,3,6,1,3,4,8,3,1,6,5,3,2,4,1,2,6,3,1,2,
4,1,6,5,4,6,3,1,4,7,3,6,5,4,1,2,4,6,1,3,4,2,1,4,6,5,4,1,2,3,6,1,3]

[1,2,3,4,5,6,4,3,2,1,4,3,6,4,2,3,4,5,6,3,4,2,1,4,3,7,1,6,5,4,1,2,
4,6,1,3,4,2,1,4,6,5,4,1,2**,3,6,1,3,4,8,3,1,6,5,3,2,4,1,2,6,3,1,2,
4,1,6,5,4,6,7,2,1,6,3,2,1,4,5,6,4,1,2,4,3,1,6,4,2,1,4,5,6,3,7,4,2]

[1,2,3,4,5,6,4,3,2,1,4,3,6,4,2,3,4,5,6,3,4,2,1,4,3,7,1,6,5,4,1,2,
4,6,1,3,4,2,1,4,6,5,4,1,2**,3,6,1,3,4,8,7,4,3,1,2,4,6,5,4,1,2,4,6,
1,3,4,2,1,4,6,5,4,1,2,4,7,3,5,4,6,1,2,4,6,5,4,2,3,5,2,1,4,6,5,4,1]

[1,2,3,4,5,6,4,3,2,1,4,3,6,4,2,3,4,5,6,3,4,2,1,4,3,7,1,6,5,4,1,2,
4,6,1,3,4,2,1,4,6,5,4,1,2**,3,6,1,3,4,8,7,4,3,1,2,4,6,5,4,1,2,4,6,
1,3,4,2,1,4,6,5,4,1,2,4,7,3,5,4,6,1,3,6,2,1,4,6,3,1,6,5,4,6,3,1,2]

[1,2,3,4,5,6,4,3,2,1,4,3,6,4,2,3,4,5,6,3,4,2,1,4,3,7,1,6,5,4,1,2,
4,6,1,3,4,2,1,4,6,5,4,1,2**,3,6,1,3,4,8,7,4,3,1,2,4,6,5,4,1,2,4,6,
1,3,4,2,1,4,6,5,4,1,2,4,7,3,5,4,6,2,1,4,6,5,4,2,3,5,2,1,4,6,5,4,1]

[1,2,3,4,5,6,4,3,2,1,4,3,6,4,2,3,4,5,6,3,4,2,1,4,3,7,1,6,5,4,1,2,
4,6,1,3,4,2,1,4,6,5,4,1,2**,3,6,1,3,4,8,7,4,3,1,2,4,6,5,4,1,2,4,6,
1,3,4,2,1,4,6,5,4,1,2,4,7,3,5,4,6,2,1,4,6,5,4,2,3,5,2,1,4,6,5,4,2]

[1,2,3,4,5,6,4,3,2,1,4,3,6,4,2,3,4,5,6,3,4,2,1,4,3,7,1,6,5,4,1,2,
4,6,1,3,4,2,1,4,6,5,4,1,2**,3,6,1,3,4,8,7,4,3,1,6,3,2,1,4,5,6,4,1,
2,4,3,1,6,4,2,1,4,5,6,3,7,4,1,3,6,2,4,1,3,5,2,4,1,2,6,3,1,2,4,1,6]

[1,2,3,4,5,6,4,3,2,1,4,3,6,4,2,3,4,5,6,3,4,2,1,4,3,7,1,6,5,4,1,2,
4,6,1,3,4,2,1,4,6,5,4,1,2**,3,6,1,3,4,8,7,4,3,1,6,3,2,1,4,5,6,4,1,
```

2,4,3,1,6,4,2,1,4,5,6,3,7,4,1,3,6,2,4,1,5,3,2,4,1,2,6,3,1,2,4,1,6]

[1,2,3,4,5,6,4,3,2,1,4,3,6,4,2,3,4,5,6,3,4,2,1,4,3,7,1,6,5,4,1,2,
4,6,1,3,4,2,1,4,6,5,4,1,2**,3,6,1,3,4,8,7,4,3,1,6,3,2,1,4,5,6,4,1,
2,4,3,1,6,4,2,1,4,5,6,3,7,4,1,3,6,4,5,6,1,4,2,1,3,6,2,1,4,2,3,5,6]

[1,2,3,4,5,6,4,3,2,1,4,3,6,4,2,3,4,5,6,3,4,2,1,4,3,7,1,6,5,4,1,2,
4,6,1,3,4,2,1,4,6,5,4,1,2**,3,6,1,3,4,8,7,4,3,1,6,3,2,1,4,5,6,4,1,
2,4,3,1,6,4,2,1,4,5,6,3,7,4,1,3,6,4,5,6,1,4,2,1,3,6,2,1,4,2,5,3,6]

[1,2,3,4,5,6,4,3,2,1,4,3,6,4,2,3,4,5,6,3,4,2,1,4,3,7,1,6,5,4,1,2,
4,6,1,3,4,2,1,4,6,5,4,1,2**,3,6,1,3,4,8,7,4,3,2,1,4,6,5,4,1,2,4,6,
1,3,4,2,1,4,6,5,4,1,2,4,7,3,5,4,6,1,2,4,6,5,4,2,3,5,2,1,4,6,5,4,1]

[1,2,3,4,5,6,4,3,2,1,4,3,6,4,2,3,4,5,6,3,4,2,1,4,3,7,1,6,5,4,1,2,
4,6,1,3,4,2,1,4,6,5,4,1,2**,3,6,1,3,4,8,7,4,3,2,1,4,6,5,4,1,2,4,6,
1,3,4,2,1,4,6,5,4,1,2,4,7,3,5,4,6,1,2,4,6,5,4,2,3,5,2,1,4,6,5,4,2]

[1,2,3,4,5,6,4,3,2,1,4,3,6,4,2,3,4,5,6,3,4,2,1,4,3,7,1,6,5,4,1,2,
4,6,1,3,4,2,1,4,6,5,4,1,2**,3,6,1,3,4,8,7,4,3,2,1,4,6,5,4,1,2,4,6,
1,3,4,2,1,4,6,5,4,1,2,4,7,3,5,4,6,2,1,4,6,5,4,2,3,5,2,1,4,6,5,4,1]

[1,2,3,4,5,6,4,3,2,1,4,3,6,4,2,3,4,5,6,3,4,2,1,4,3,7,1,6,5,4,1,2,
4,6,1,3,4,2,1,4,6,5,4,1,2**,3,6,1,3,4,8,7,4,3,2,1,4,6,5,4,1,2,4,6,
1,3,4,2,1,4,6,5,4,1,2,4,7,3,5,4,6,2,1,4,6,5,4,2,3,5,2,1,4,6,5,4,2]

[1,2,3,4,5,6,4,3,2,1,4,3,6,4,2,3,4,5,6,3,4,2,1,4,3,7,1,6,5,4,1,2,
4,6,1,3,4,2,1,4,6,5,4,1,2**,4,8,5,2,4,1,2,6,3,1,2,4,1,6,5,2,4,1,7,
3,2,1,4,5,6,4,1,2,4,3,1,6,4,2,1,4,5,6,3,7,4,1,3,7,2,3,4,7,8,4,3,1]

Bibliography

[Abbott88] Abbott, H. L. and M. Katchalski. 1988. "On the Snake-In-The-Box Problem." *Journal of Combinatorial Theory*. 45:13-24.

[Abbott91a] Abbott, H. L. and M. Katchalski. 1991. "Further Results on Snakes in Powers of Complete Graphs." *Discrete Mathematics*. 91:111-120.

[Abbott91b] Abbott, H. L. and M. Katchalski. 1991. "On the Construction of Snake in the Box Codes." *Utilitas Mathematica*. 40:97-116.

[Adelson73] Adelson, L. E., R. Alter, and T. B. Curtz. 1973. "Long Snakes and a Characterization of Maximal Snakes on the d-Cube." *Proceedings, 4th Southeastern Conference on Combinatorics, Graph Theory, and Computing*. 8:111-124.

[Baker85] Baker, J. E. 1985. "Adaptive Selection Methods for Genetic Algorithms." *Proceedings of the First International Conference on Genetic Algorithms and their Applications*. 101-111.

[Black64] Black, W. L. 1964. "Electronic Combination Locks." *Technical Report 73*. Research Laboratory of Electronics, Massachusetts Institute of Technology.

[Chien64] Chien, R.T., C. V. Freiman, and D. T. Tang. 1964. "Error Correction and Circuits on the n-Cube." *Proceedings of the 2nd Annual Allerton Conference on Circuit and System Theory*. 899-912.

[Covington94] Covington, M.A. 1994. *Natural Language Processing for Prolog Programmers*. Englewood Cliffs, New Jersey: Prentice Hall.

[Covington97] Covington, M.A., D. Nute, A. Vellino. 1997. *Prolog Programming in Depth*. Upper Saddle River, New Jersey: Prentice Hall.

[Danzer67] Danzer, L. and V. Klee. 1967. "Lengths of Snakes in Boxes." *Journal Combinatorical Theory*. 2:258-265.

[Davies65] Davies, D. W. 1965. "Longest 'Separated' Paths and Loops in an N Cube." *IEEE Transactions on Electronic Computers*. 14:261.

[Deimer84] Deimer, K. 1984. "Some New Bounds for the Maximum Length of Circuit Codes." *IEEE Transactions on Information Theory*. 30:754-756.

[Deimer85] Deimer, K. 1985. "A new upper bound on the length of snakes." *Combinatorica*. 5:109-120.

[DeJong75] De Jong, K.A. 1975. "An Analysis of the Behavior of a Class of Genetic Adaptive Systems." *Dissertation Abstracts International*. 36:10, 5140B.

[Dickens98a] Dickens, T.P. and C.L. Karr. 1998. "Method to achieve better performance in genetic algorithms applied to time-constrained, multi-solution problems." *IEEE International Joint Symposia on Intelligence and Systems*. 2-9.

[Dickens98b] ed. Karr, C.L. and L.M. Freeman. 1998. *Industrial Applications of Genetic Algorithms*.

[Douglas69] Douglas, R. J. 1969. "Upper Bounds on the Lengths of Circuits of Even Spread in the d-Cube." *Journal Combinatoric Theory*. 7:206-214.

[Goldberg85] Goldberg D. and R. Lingle. 1985. "Alleles, loci, and the Traveling Salesman Problem." *Proceedings of the International Conference on Genetic Algorithms and their Applications*.

[Goldberg89] Goldberg, D. E. and K. Deb. 1991. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley Publishing Company, Inc.

[Goldberg90] Goldberg, D. E., K. Deb and B. Korb. 1990. "Messy genetic algorithms revisited: Studies in mixed size and scale." *Complex Systems*. 4:415-444.

[Goldberg91] Goldberg, D. E. 1989. "A comparative analysis of selection schemes used in genetic algorithms." In G. Rawlins, ed., *Foundations of Genetic Algorithms*. Morgan Kaufmann.

[Harary88] Harary, F., J. P. Hayes, and H. J. Wu. 1988. "A Survey of the Theory of Hypercube Graphs." *Computer Mathematics Applications*. 15:277-289.

[Hiltgen01] Hiltgen, A. P. and K. G. Paterson. 2001. "Single Track Circuit Codes." *IEEE Transactions on Information Theory*. 47:2587-2595.

[Homaifar93] Hamaifar, A., S. Guan, and G.E. Liepins. 1994. "A New Approach on the Traveling Salesman Problem by Genetic Algorithms." *Proceedings of the 5th International Conference on Genetic Algorithms*. 460-466.

[Holland75] Holland, J.H. 1975. *Adaptation in Natural and Artificial Systems*. Ann Arbor: The University of Michigan Press.

[Jablonskii74] Jablonskii, S. V. 1974. "Discrete Mathematics and Mathematical Problems of Cybernetics." *Nauka*. Moscow, Russia. 112-116.

[Kautz58] Kautz, W.H. 1958. "Unit-distance error-checking codes." *IRE Transaction of Electronic Computing*. EC-7:179-180.

[Kiranmayee01] Kiranmayee, J. L. 2001. "Existence of Maximal Snakes of Length $2(d+2)$." *Presented at Techfiesta in Coimbatore, India in December of 2001.* http://cs.roanoke.edu/∼shende/Papers/Snakes/Snake2d4.ps (accessed Oct. 10, 2004).

[Klee67] Klee, V. 1967. "A Method for Constructing circuit codes." *Journal of the Association for Computing Machinery*. 14:520-538.

[Klee70] Klee, V. 1970. "What is the Maximum Length of a d-Dimensional Snake?" *American Mathematics Monthly*. 77:63-65.

[Kochut94] Kochut, K.J., J.A. Miller, W.D. Potter, R.W. Robinson. 1994. "Hunting For Snake-In-The-Box Codes." submitted to Annals of Mathematics and Artificial Intelligence, K.J. Kochut,

[Kochut96] Kochut, K.J. 1996. "Snake-In-The-Box Codes for Dimension 7." *Journal of Combinatorial Mathematics and Combinatorial Computing*. 20:175-185.

[Lamma00] Lamma, E., L. M. Pereira, and F. Riguzzi. 2000. "Logic Aided Lamarckian Evolution." *Proceedings of the Fifth International Workshop on Multistrategy Learning (MSL2000)*. 20:175-185.

[Lewinson95] Lewinson, L. 1995. "Genehunter: GA Software From Ward." *PC AI*. 9(2):26.

[Luger98] Luger, G.F. and W.A. Stubblefield. 1998. *Artificial Intelligence - Structures and Strategies for Complex Problem Solving*. 3rd Edition. Reading, MA: Addison Wesley Longman, Inc.

[Mitchell98] Mitchell, M. 1998. *An Introduction to Genetic Algorithms*. Cambridge, MA: The MIT Press.

[Olsson96] Olsson, B. 1996. "PROGA - PROlog implementation of a Simple Genetic Algorithm." http://www.ida.his.se/∼bjorne/GA/Simuleringar/ProGA/ (accessed Sept. 13, 2004).

[Oliver87] Oliver, I., D. Smith, and J. Holland. 1987. "A Study of Permutation Crossover Operators on the Traveling Salesman Problem." *Proceedings of the Second International Conference on Genetic Algorithms and their Applications*.

[Paterson98] Paterson, K.G. and J. Tuliani. 1998. "Some New Circuit Codes." *IEEE Transactions on Information Theory*. 44(3): 1305-1309.

[Potter94] Potter, W.D., R.W. Robinson, J.A. Miller, K.J. Kochut, D.Z. Redys. 1975. "Using The Genetic Algorithm to Find Snake-In-The-Box-Codes." *Industrial and Engineering Applications of Artificial Intelligence and Expert Systems - Proceedings of the Seventh International Conference*. Austin, Texas. 421-426.

[Prasanna01] Prasanna, M.S.L. and C. Swapna. 2001. "Families of Maximal Snakes of Length 2(d+1)." *Presented at Techfiesta in Coimbatore, India in December of 2001.* http://cs.roanoke.edu/∼shende/Papers/Snakes/Families.ps (accessed Oct. 10, 2004).

[Rajan99] Rajan, D.S., A.M. Shende. 1999. "Maximal and reversible snakes in hypercubes." *24th Annual Austalasian Conference on Combinatorial Mathematics and Combinatorial Computing.* Northern Territory University, Darwin, Australia.

[Rickabaugh99] Rickabaugh, B. P. and A.M. Shende. 1999. "Using PVM to Hunt for Maximal Snakes in Hypercubes." *Journal of Computing in Small Colleges.* 14(2): 76-84.

[Russell95] Russell, S.J. and P. Norvig. 1995. *Artificial Intelligence - A Modern Approach.* Upper Saddle River, NJ: Prentice Hall.

[Salah02] Salah, A. A. 2002. "Latrunculli with Prolog." http://yunus.cmpe.boun.edu.tr/∼salah/540/index.html (accessed Oct. 10, 2004).

[Scheidt01] Scheidt, K. G. 2001. "Searching for Patterns of Snakes in Hypercubes." *Journal of Computing Sciences in Colleges.* 16(2):168-176.

[Singleton66] Singleton, R. C. 1966. "Generalized Snake-in-the-Box Codes." *IEEE Transactions on Electronic Computers.* 15:596-602.

[Snevily94] Snevily, H. S. 1994. "The Snake-in-the-Box Problem: A New Upper Bound." *Discrete Mathematics.* 133(1-3):307-314.

[Sterling94] Sterling, L., E. Shapiro. 1994. *The Art of Prolog.* 2nd Edition. Cambridge, Massachusetts: The MIT Press.

[Starkweather91] Starkweather, T., S. McDaniel, K. Mathias, D. Whitley, and C. Whitley. 1991. "A Comparison of Genetic Sequencing Operators." *Proceedings of the Fourth International Conference on Genetic Algorithms.*

[Tamaki94] Tamaki, H., H. Kita, N. Shimizu, K. Maekawa, and Y. Nishikawa. 1994. "A Comparison Study of Genetic Codings for the Traveling Salesman Problem." *Proceedings of the First IEEE Conference on Evolutionary Computation.* 1:1-7.

[Tovey81] Tovey, C. A. 1981. "Polynomial Local Improvement Algorithms in Combinatorial Optimization." Ph.D. Dissertation. Stanford University, Palo Alto, California.

[Whitley89] Whitley, D., T. Starkweather, and D. Fuquay. 1989. "Scheduling Problems and Traveling Salesman: The Genetic Edge Recombination Operator." *Proceedings of the Third International Conference on Genetic Algorithms and their Applications.* George Mason University, Fairfax, Virginia. J. D. Shaeffer, ed. Morgan Kaufmann.

[Whitley90] Whitley, D., T. Starkweather, and D. Shaner. 1990. "Traveling Salesman and Sequence Scheduling: Quality Solutions Using Genetic Edge Recombination." *In Handbook of Genetic Algorithms.* L. Davis, ed. Van Nostrand Reinhold, New York.