AN ITERATED LOCAL SEARCH WITH ADAPTIVE MEMORY APPLIED TO THE SNAKE IN THE BOX PROBLEM

by

WILLIAM J. BROWN

(Under the Direction of Walter D. Potter)

ABSTRACT

A snake-in-the box code, first described in [Kautz, 1958], is an achordal open path in a hypercube. Finding bounds for the longest snake at each dimension has proven to be a difficult problem in mathematics and computer science. Evolutionary techniques have succeeded in tightening the bounds of longest snakes in several dimensions [Potter, 1994] [Casella, 2005]. This thesis utilizes an Iterated Local Search heuristic with adaptive memory on the snake-in-the-box problem. The results match the best published results for problem instances up to dimension 8. The lack of implicit parallelism segregates this experiment from previous heuristics applied to this problem. As a result, this thesis provides insight into those problems in which evolutionary methods dominate.

INDEX WORDS: snake-in-the-box, snake, coil, Gray code, error correcting code, circuit code, hypercube, iterated local search, ILS, adaptive memory, AMP, heuristic, metaheuristic, graph theory

AN ITERATED LOCAL SEARCH WITH ADAPTIVE MEMORY APPLIED TO THE SNAKE IN THE BOX PROBLEM

by

WILLIAM J. BROWN

B.S. Mathematics, Georgia Institute of Technology, 2001

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

2005

AN ITERATED LOCAL SEARCH WITH ADAPTIVE MEMORY APPLIED TO THE SNAKE IN THE BOX PROBLEM

by

WILLIAM J. BROWN

Major Professor: Walter D. Potter

Committee: Hamid R. Arabnia
Khaled Rasheed

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
May 2005

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENTS iv

LIST OF TABLES viii

LIST OF FIGURES ix

CHAPTER

1 INTRODUCTION 1

1.1 OVERVIEW 1

1.2 COMBINATORIAL OPTIMIZATION PROBLEMS 1

1.3 THE SNAKE-IN-THE-BOX PROBLEM 2

1.4 HEURISTICS & META-HEURISTICS 3

1.5 THESIS MOTIVATION 5

1.6 THESIS ORGANIZATION 5

2 NATURE INSPIRED HEURISTICS 6

2.1 OVERVIEW 6

2.2 APPROACHES 6

2.3 COMPARING THE METHODS 11

2.4 CONCLUSIONS 16

3 ITERATED LOCAL SEARCH 17

3.1 INTRODUCTION 17

3.2 BASICS OF THE ALGORITHM 17

3.3 Diversification v. Intensification 19
3.4 ILS Components 20
3.5 Adaptive Memory Programming 23
4 The Snake-In-The-Box Problem 25
4.1 Definitions and Notation 25
4.2 Snakes 28
4.3 Applications of Snake-In-The-Box Codes 32
5 An Outline of the Process 34
5.1 Introduction 34
5.2 Snake Construction Heuristic 34
5.3 Solution Representation 35
5.4 Induced Subgraph 36
5.5 Iterated Local Search 39
5.6 Principle of Proximate Optimality 40
5.7 Parameter Tuning 42
6 Results 44
6.1 Iterated Local Search 44
6.2 Iterated Local Search with Adaptive Memory 45
7 Conclusions and Future Direction 53
7.1 Conclusions 53
7.2 Future Direction 54
REFERENCES 55

APPENDICES 59

A Java Code 59

LIST OF TABLES

Page

Table 4.1: Upper and lower bounds for *S(d)* in dimensions 7 to 11 31

Table 6.1: *S(d) r*esults for ILS with no memory 44

Table 6.2: ILS with AMP results, varying solution list size, dimension 7 45

Table 6.3: ILS with AMP results, varying solution list size, dimension 8 46

Table 6.4: ILS with AMP results, varying solution list size, dimension 9 47

Table 6.5: ILS with AMP results, varying ILS stopping criteria, dimension 7 48

Table 6.6: ILS with AMP results, varying ILS stopping criteria, dimension 8 48

Table 6.7: ILS with AMP results, varying ILS stopping criteria, dimension 9 49

Table 6.8: ILS with AMP results, stopping criteria & solution pool size, dimension 8 50

Table 6.9: ILS with AMP results, stopping criteria & solution pool size, dimension 9 51

Table 6.10: Dimension 8, ILS iterations = 50, Solution Pool size = 50 51

Table 6.11: Dimension 9, ILS iterations = 5, Solution Pool size = 100 52

LIST OF FIGURES

Page

Figure 4.1: A three dimensional hypercube.....26

Figure 4.2: A four dimensional hypercube 26

Figure 4.3: A standard cube with binary encoding of vertices 27

Figure 4.4: A four dimensional hypercube depicted as the joining of two three dimensional 28

Figure 4.5: A snake of length 4 on the standard cube.....28

Figure 4.6: A coil of length 6 on the standard cube.....29

Figure 4.7: A comparison of *S(d)* bounds derived computationally and non-computationally.....32

CHAPTER ONE INTRODUCTION

1.1 OVERVIEW

This chapter introduces the combinatorial optimization problems and the notion of an intractable problem. Heuristics and meta-heuristics are then discussed with a focus on metaheuristics utilizing adaptive memory. A problem called the *snake-in-the-box* problem is then briefly introduced.

1.2 COMBINATORIAL OPTIMIZATION PROBLEMS

Many problems exist that cannot be solved by traditional methods. These problems often suffer from *combinatorial explosion*, meaning that the solution space (or number of states) of the problem grows as an exponential function of the problem size. In the literature, when a problem suffers from combinatorial explosion and the instance of the problem is large enough it is commonly referred to as an *intractable* problem. An intractable problem is one for which no known polynomial-time algorithm exists and whose solution cannot be computed in a straightforward way in a reasonable amount of time.

Given a set of feasible solutions, S, a *discrete optimization problem* is one for which a real-valued objective function, f, consisting of one or more discrete values, is defined on S. The set S is commonly referred to as the *state-space* and the function f is referred to as the *fitness function* (within the context of Genetic Algorithms), or the *cost function*. The element(s) of S

with a minimal (or possibly maximal) f value is known as the optimum of the problem. If S is a finite set, the problem then becomes a *combinatorial optimization problem* [Lee, 2004]. Combinatorial optimization involves determining an optimal value from the elements of S. Since S is finite, an enumeration of all possible solutions will provide the optimum value in any combinatorial optimization problem. Often, however, enumeration is not a desirable choice and in many cases an intractable one. The notion of a combinatorial optimization problem can be formalized as follows. *Given a set, S, of feasible solutions and an objective function f, such that f maps elements of S to* ***R,*** *find the element s* of S, such that f(s*) <= f(s) for all s in S.*

1.3 THE SNAKE-IN-THE-BOX PROBLEM

The *snake-in-the-box problem* (or the problem of finding the longest snakes in hyper-cubes) is an example of a combinatorial optimization problem. The problem entails finding the longest achordal path through an n-dimensional hypercube. Interest in snake-in-the-box codes began with [Kautz, 1958] in the context of unit distance, error checking codes. Similar to Gray codes, snakes are commonly referred to as *circuit codes (*or *circuit codes of spread 2*). By using the visited nodes of the hypercube as code words, SIB codes found application in analog to digital conversion [Klee, 1970]. If a codeword has one bit corrupted in the communication process, it either corrupts to a neighbor on the path (negligible errors) or a non-codeword (error detected). Similarly, if a subset of a SIB code is chosen in such a way that code words differ by at least two bits, the result is an error-correcting code (for single-bit errors). SIB codes have found application in other areas as well. They have been used in constructing electronic locking

schemes in [Kim, 2000] and [Dunin-Barkowski, 1999] uses a SIB code in a cerebellar learning model. A formal introduction to the Snake-in-the-Box problem is presented in chapter 4.

1.4 HEURISTICS & META-HEURISTICS

In searching for an optimal solution in S, a *heuristic* is a "rule of thumb" that is believed to encourage the search into areas with optimal or near-optimal solutions. Heuristic approaches differ from exact methods in that they do not guarantee that a globally optimal value will be reached. A popular heuristic search algorithm is the classical descent method. In this method a random element of S is initially chosen as the starting point. Iteratively, the starting point is replaced with an element adjacent to it with the smallest f value. This continues until a local minimum is found (not necessarily a global minimum). Note that this heuristic may only be applied to problems for which S is a *metric space* or for S where the notion of "adjacent" elements is meaningful.

The term *meta-heuristic* describes a more sophisticated search where a heuristic is used to guide other, simpler heuristics. Some examples of the more popular meta-heuristics are given below:

- The *genetic algorithm* uses a population of solutions and "evolves" better population members (with respect to the fitness function) by repeatedly applying mating and mutation operators [Holland 1975][Goldberg 1989].
- The *simulated annealing* algorithm simulates the cooling process of a metal [Kirkpatrick 1983].

- *Ant colony optimization* emulates a colony of ants by simulating "pheromone trails" the ants would leave as they search the solution space [Colorni 1992].
- *Tabu Search* attempts to emulate human memory by keeping track of previously visited solutions in a "tabu list". This is hoped to drive the search out of local minima [Glover 1986].

These and other meta-heuristic strategies have gained popularity with many combinatorial optimization problems. Good meta-heuristics can dramatically reduce the time required to solve a problem by eliminating the need to consider unlikely possibilities or states. Meta-heuristic methods however, do not guarantee that an optimal solution will be found in a tractable way. If the optimal solution to problems is not always found, then why has so much attention been diverted to these methods in the last two decades? The answer is that modern meta-heuristic approaches provide "good" quality (optimal, or near optimal) solutions in a short amount of time. Many practical optimization problems to which meta-heuristics are applied are created by roughly simulating a real problem in such a way that to find a global optimum is not necessarily more beneficial than finding a "near" optimum.

The aforementioned meta-heuristic strategies are all modeled after naturally occurring processes. A study of these techniques follows in the next chapter. By investigating these heuristics, we hope to determine which features are necessary when tailoring a heuristic to a specific problem. The result is an Iterated Local Search (ILS) algorithm that utilizes adaptive memory. Details of the algorithm are presented in Chapter 5.

1.5 THESIS MOTIVATION

The snake-in-the-box problem has been analytically studied by many researchers, including [Kautz, 1958][Wojciechowski, 1989][Abbot, 1991][Snevily, 1994]. Many of these approaches resulted in establishing new upper and or lower bounds for the SIB problem in various dimensions. A lower-bound proof by construction was first attempted heuristically by [Potter 1994] when they tailored a genetic algorithm to the dimension 8 SIB problem. An exhaustive search of the solution space in dimension 7 was carried out computationally by [Kochut 1996]. Since then, there have been a number of computational methods employed, many of these being heuristic methods. The heuristic approaches used to date have been primarily based on evolutionary methods. This thesis presents a non-evolutionary method and hopes to present a new strategy for finding maximal snakes and tighten the bounds on maximal snakes. This thesis arrives at an Iterated Local Search algorithm with adaptive memory tailored for finding snake-in-the-box codes.

1.6 THESIS ORGANIZATION

The thesis is organized as follows. Chapter 2 presents a study of some of the widely used heuristics that are founded on naturally occurring systems. Chapter 3 gives a formal presentation of the snake-in-the-box problem. Chapter 4 describes the metaheuristic known as Iterated Local Search and the more general class of metaheuristics that employ Adaptive Memory. Chapter 5 outlines the process utilized in this paper and Chapter 6 presents our results. We present our conclusions and further research goals in Chapter 7.

CHAPTER TWO NATURE INSPIRED HEURISTICS

2.1 OVERVIEW

Heuristic approaches to combinatorial optimization problems have developed dramatically in the last three decades. They have been successful in tackling many difficult optimization problems for which finding a solution in a straight-forward manner is computationally infeasible. Some of the most widely used heuristic techniques are inspired from naturally occurring systems and include: genetic algorithms, tabu seach, simulated annealing and ant colony optimization. The systems that these approaches are based on are biological evolution, intelligent problem solving, physical sciences and swarm intelligence, respectively. These heuristic approaches can be classified according to the particular characteristics of each algorithm. This classification leads to a better understanding of what strengths and shortcomings each method contains.

2.2 APPROACHES

In this section, we present an overview of some of the most popular heuristic approaches that owe part of their inspiration to a naturally occurring process.

2.2.1 GENETIC ALGORITHMS

Inspired by Darwin's theory of evolution, genetic algorithms were first developed by [Holland, 75] and formalized into their modern representation by [Goldberg, 89]. Genetic Algorithms solve problems by evolving an answer from a pool of possible solutions. The algorithm begins with a set of possible solutions (chromosomes) called a population. Each chromosome in the population has an objective function value (fitness) associated with it. This fitness is an indicator of how "good" of a solution the chromosome is. Certain (more fit) chromosomes are then chosen to help create a new population. This is motivated by the hope that the new population (next generation) will be, on average, more "fit" than the previous one. The process of choosing chromosomes is called selection. Once a set of chromosomes has been selected, they must mate to form a new population – this is called mating or recombination. This process is repeated for either a predetermined number of generations or until the average fitness stops improving for a set number of generations. This setup poses a problem. What happens if all of the initial chromosomes are too similar? Or what happens if, through generations of breeding with the same population size, the population becomes too similar? This can be problematic if the algorithm converges around a local minimum/maximum. To combat this, a mutation operator is applied after recombination. The mutation operator slightly changes chromosomes to give each generation some diversity.

2.2.2 Ant Colony Optimization

The ant colony optimization technique was developed as a result of an experiment performed with Argentine ants by [Goss, 1989]. In the experiment, a group of ants and supply of food were placed together. The food supply, however, was physically separated from the ants in such a way that the ants had to take one of two bridges to reach the food. At first, the ants chose random paths. As time progressed, the ants began to take the shorter of the two bridges. This result can be attributed to the pheromone trails that the ants leave as they travel. Since the travel time is shorter on the small bridge, more ants are able to cross it when compared to the longer bridge, thus marking the small bridge with more pheromones. The ACO algorithm follows this process by considering a population of solutions (ants) that move from neighboring solution to neighboring solution. As each solution is visited, information about the move to that solution is recorded as a pheromone trail. This information may be updated as the solution is being built (online step-by-step pheromone updating) and/or after the solution has been built (online delayed pheromone updating). These pheromone trails affect the future move selections of all of the ants of the population (ants are encouraged to move on pheromone trails). To discourage the ants from converging too quickly along a path, the pheromone trail dissipates, much in the same way that a real scent does.

In addition to the details of the aforementioned process, ant colony optimization techniques are commonly equipped with other features. One common practice is for an ant to deposit an amount of pheromone proportional to the quality of the solution it is building. Depending on the problem being tackled, the ants are sometimes given some type of memory structure. In some versions of the ant colony optimization technique, the pheromone trail is

updated by a more centralized entity (that is, an agent that incorporates information about the entire search, not just one ant). This type of update is referred to as offline pheromone updating [Dorigo, 1999].

2.2.3 TABU SEARCH

Tabu search's modern form can be accredited to [Glover, 86], based on work that he had done previously [Glover, 77]. Tabu search is now an established optimization technique that is competitive with nearly all other heuristic techniques. A heuristic designed to exploit an element of human cognition in its searches, tabu search does this by "memorizing" its states as it visits solutions in the domain. The memories that it builds are then used to influence the search. There have been a large collection of tabu search variants proposed for various combinatorial optimization problems since Glover's seminal paper. The basic structure of the tabu search method, however, is a local search algorithm equipped with a tabu list. The tabu list records information about the state of the search at each visited solution in order to discourage the search from revisiting those solutions. Just what information is recorded, as well as how that information influences the search, is a topic that has been actively researched in recent years.

While the basic structure of tabu search was outlined above, it should be noted that most tabu search implementations are highly customized to the problem at hand. While much of the literature focuses on the tabu list and how long it keeps its contents, this should not be the only focal point. The real focus of tabu search is not to just keep a list of information about the search, but to use that information in a way to guide the search to better regions in the search space. In doing this, the notions of long-term and short-term memory must be applied. Short-

term memory is typically represented by the use of the tabu list to record the characteristics of recently visited solutions (also known as recency-based memory) and thus keeps the search diversified. Long-term memory is commonly used to record information about the best solutions in order to intensify the search around those particular regions of the search space.

2.2.4 SIMULATED ANNEALING

An annealing process begins at a high temperature, which allows atoms to move freely. As the temperature is decreased, the atoms slow and eventually settle, forming a crystal. If the temperature is decreased rapidly, the resulting crystal is typically marred with defects. If the temperature is decreased slowly (annealed), then the resulting crystal typically suffers from far fewer defects. The simulated annealing technique, motivated by observations of crystallization during an annealing process, was introduced by [Kirkpatrick, 83]. The annealing process is translated into algorithmic form by incorporating a temperature parameter, T, and a cooling schedule into a local search algorithm. First, the local search begins with a solution, S. This solution is perturbed in some way and accepted by the search with a probability that is dependent on the objective function values of the solutions and on the temperature parameter, T. As T decreases, the probability of moving the search from a better state to a worse state decreases as well. Determining the optimal cooling schedule is left up to the implementer of the algorithm. Once it has been found, it can usually be applied to similar problems.

2.3 COMPARING THE METHODS

The general structures of the aforementioned heuristic approaches may, at first glance, not seem to share common characteristics. There is, however, a simple unifying agent among these algorithms. Each heuristic mimics the naturally occurring concepts of selection (optimization) and mutation (randomization) [Colorni, 1996]. Genetic algorithms follow this trend in a straightforward manner. Ant colony optimization achieves randomization through the use of random-walk agents (ants). The optimization process is garnered in the building of pheromone trails. In the simplest case, the tabu list found in tabu search implementations provides an element randomization in the search. Optimization occurs as a result of the hill-climbing strategy commonly invoked. The tabu search literature (as well as the literature of most other heuristic approaches) focuses on the balancing of intensification (thoroughly inspecting local minima) with diversification (thoroughly inspecting the solution space). This concept is comparable to the ideas of optimization and mutation, respectively. Randomization is an obvious element of simulated annealing. The optimization occurs as the temperature drops and the search converges around a local minimum.

While the optimization and randomization concepts previously outlined provide insight into the fundamentals of nature-inspired heuristics, a more intimate comparison of the methods follows.

2.3.1 MULTIPLE SEARCH AGENTS V. SINGLE SEARCH AGENTS

Perhaps the most obvious way of partitioning the above methods is to separate them into population-based and single-point searches. Genetic algorithms and ant colony optimization are both methods that are searching with a population of solutions at each iteration. Tabu search and simulated annealing, however, keep only one current solution. While at a first glance it may seem that a population-based approach is automatically superior to the single-point approach, this is not always the case.

For example, [Areibi, 2001] asserts that single-point searches are superior to population-based methods in solving the partitioning and placements problems. This is due to single-point searches being more equipped to perform finely tuned searches. Population-based approaches, especially evolutionary methods, are good at exploring the solution space since they search from a set of designs and not from a single design. Typically population-based searches, while good at covering a diverse part of the search space, are susceptible to being very "near" optimal solutions and never finding them. This behavior is a result of genetic algorithms and ant colony optimization making large jumps in the neighborhood graph (via constructing solutions with ants and applying genetic operators, respectively) [Birattari, 2001]. These methods are able to follow a discontinuous walk through the search space. The single-point approaches do not have this behavior. In a single point approach such as tabu search or simulated annealing, a continuous walk through the search space is typically adhered to. At each iteration, the solution changes to another solution in its neighborhood.

The purpose of population-based searches such as genetic algorithms and ant colony optimization is for the agents of the populations to eventually "agree" on an optimal solution. In

contrast to this idea, one of the main goals of tabu search is to prevent the search from converging, so that the search may continue investigating unexplored regions of the solution space until a threshold is satisfied. Convergence in a tabu search algorithm manifests itself as *cycling* - repeating a sequence of solution transitions indefinitely, or as pointed out by [Battiti, 1994] as restricting the search around a chaotic attractor. The list from which tabu search gets its name is responsible for keeping the search out of local minima. In simulated annealing, the element of randomization is so prevalent that the search only converges as the temperature drops.

2.3.2 Memory Utilizing V. Memoryless Methods

Of the four approaches highlighted here, three of the four utilize information about previous iterations when calculating search moves in the current iteration. The most obvious candidate for being a memory utilizing method is tabu search. Typically, tabu search uses a combination of explicit and attributive memories. Explicit memory is used when entire solutions are memorized in the search. Typically, only elite solutions are recorded in this fashion, although versions of tabu search exist that record entire solutions in the tabu list (this is known as Strict-Tabu) [Glover, 97]. Attributive memory uses information about the attributes of previously visited solutions. This memory is used to encourage moves toward particular areas of the solution space (intensification) and to discourage moves toward particular areas of the solution space (diversification).

While tabu search is based upon one observer that “remembers” aspects of solutions in order to make “smarter” decisions in the future, ant colony optimization relies on a population of “dumb” observers. The tabu list style of adaptive memory is replaced with a group memory of

pheromone trails. Ant colony optimization takes a set of random walking agents and encourages them to converge by affecting their decisions with the pheromone trails. These pheromone trails can be considered as a type of adaptive memory [Birattari, 2001], since they are constructed from visited solutions and their corresponding fitness values. These pheromone trails encourage the ants to choose paths that lead to relatively "good" regions of the solution space (intensification). An evaporation mechanism is usually employed, so that the ants do not follow along a single path of pheromones (diversification). The information contained in the pheromone trails may also be utilized and modified in a more straightforward manner, i.e. by a daemon that uses global information to affect the search.

The population of solutions acts as a form of memory in genetic algorithms. In particular, the schemata of the solution strings are "remembered" (or more appropriately, "inherited") from the previous generation. Thus, the population can be thought of as an overview of information about previous iterations. Reproduction and recombination operators eventually cause schemata of the solution strings to increase/decrease in proportion to their relative fitness in the population [Holland, 75], leading the search into better regions of the solution space.

Simulated annealing does not utilize any type of memory in its search.

The group of heuristics that utilize a form of memory have recently been grouped under the name "Adaptive Memory Programming" [Taillard, 1998]. These methods are characterized by common elements in their general structures. These commonalities are: keeping some type of memory about previously visited solutions, using this memory to generate a new starting solution and then applying a search method to this new starting solution. For a more in-depth discussion on the topic, see [Taillard, 1998].

2.3.3 NEIGHBORHOOD STRUCTURES

Solution construction in genetic algorithms is not the result of static neighborhoods. In fact, the mutation operator can arbitrarily change a solution to a different neighborhood at each iteration. This type of neighborhood, coupled with the concept of a population suggests that genetic algorithms will search more of the solution space than would a single-pointed, neighborhood-based search. [Areibi, 1995] asserts that interchange methods, such as those used in the recombination stage of genetic algorithms, are more likely to fail to converge to "optimal" solutions.

Tabu search's use of long-term and short-term memory have affects on the neighborhood structure of the search. Short-term memory typically reduces the size of the current neighborhood by excluding the moves listed as "tabu". Longer term memories are used to expand the current neighborhood to include solutions that would not be in the neighborhood otherwise [Glover, 1997]. Thus, the use of memory creates a dynamic neighborhood for the search at each iteration.

Simulated annealing utilizes what is typically referred to as a static neighborhood. That is, any solution visited twice in the search will experience the same set of neighboring solutions.

In ACO, the term neighborhood is reserved for the choices available by each agent when constructing a solution. These neighborhoods are static in nature, and choices made are affected by the pheromone trails. Neighborhoods of entire solutions are not defined, as each solution is built be a separate, semi-random process (ant). ACO algorithms that utilize the notion of a daemon may incorporate global information into the search and therefore could alter the

neighborhood structure that the ants experience. In a graph problem, for example, a daemon could rule out a section of the graph by pruning edges.

2.4 CONCLUSIONS

In investigating the heuristic approaches that are based in naturally occurring systems similarities emerge. The roles of randomization and optimization are prevalent in each of the techniques. The methods differ in their approaches and by what importance they place on the ideas of randomization and optimization. Simulated annealing and ant colony optimization are essentially randomized searches with constraints (in the form of an annealing schedule and pheromone trails) that encourage optimization. Tabu search and genetic algorithms, however, are optimizing searches with constraints enforcing randomization (in the form of a tabu list and the mutation operator).

In pointing out the differences and potential shortcomings of these heuristic techniques, it should not be surprising that, in practice, hybrid algorithms are commonly employed. In order for a metaheuristic to be successful on an optimization problem, it must balance the effort it exerts on exploring the search space with the effort it exerts in exploiting the information gathered from previously visited solutions [Birattari, 2001]. Incorporating elements from multiple heuristic approaches allows more flexibility to achieve this balance.

CHAPTER 3 ITERATED LOCAL SEARCH

3.1 INTRODUCTION

Over the last two decades, heuristic techniques have become more and more competitive. Less attention has been given to the general structures of the algorithms and more attention has been reserved for problem specific tailoring of the methods. Iterated Local Search (ILS) is a metaheuristic designed to embed another, problem-specific, local search as if it were a "black-box". This allows Iterated Local Search to keep a more general structure than other metaheuristics currently in practice.

The basic idea of ILS is described as follows. Generate a starting solution, s0, and repeat the following. Execute the black-box local search on s0 to obtain s0', a locally optimal value. Modify this value in some way as to arrive at a new solution s1. This simple type of search has be reinvented numerous times in the literature, with one of its earliest incarnations appearing in [Lin, 1973].

3.2 BASICS OF THE ALGORITHM

In order to construct an ILS, one first needs an optimization problem, with an objective function, *f*, defined over a solution space, S. Given these things, a local search procedure specific to the problem must be obtained. This local search procedure, denoted by $local_f$, should take in a starting solution $s \in S$ and return a locally optimal value. Let this locally optimal value

be denoted by s^* and let the set of all local optima be denoted by S^*, a proper subset of S. Then $local_f$ defines a surjection from S to S^*. The ultimate goal of the Iterated Local Search is to perform a search on the set S^*. It accomplishes this by repeatedly iterating the procedure $local_f$, at each iteration generating an s^* of S^*.

First, an initial solution, s_o, is constructed, either randomly or with some construction method. This solution is passed to the local search, resulting in a new locally optimal solution, s. This is the "current" solution of the algorithm and will remain so until another solution is accepted to be the current solution. The locally optimal solution is changed in some way, resulting in s'. How the locally optimal solutions are changed is an implementation detail carried out in the procedure *perturb*. Ideally, the perturb operation "transforms an excellent solution into an excellent starting point for a local search" [Lourenco, 2001]. s' is then passed to the local search, resulting in a another new locally optimal solution, s^*. This new local optimum then must pass certain acceptance criterion (this is defined in the procedure *accept*) and if it does, s^* becomes the new "current" solution. Note that the "current" solution of the ILS metaheuristic refers to the solution that will act as the starting solution for the black-box search in the next iteration. More details regarding the *perturb* and *accept* procedures are covered in the sections that follow.

The Iterated Local Search involves four main components: creating an initial solution, a black-box heuristic that acts as a local search on the set S, the operation perturb, which modifies a solution; and the procedure accept, which determines whether or not a perturbed solution will become the starting point of the next iteration. Pseudo-code for the algorithm is given below.

```
Iterated Local Search
     Let:
     f          the function to minimize
     Local_f    a local search with the objective of minimizing f
     S          the set f is defined on
     s          an element of S
     S*         the set of f(s), or local optima
     s*         an element of S*
     Perturb    Perturbation function
     Accept     Acceptance Criterion
     Then:
     Choose an initial solution s_0 in S
     s* = L_f(s_0)
     s  = s*
     Loop
            s' = Perturb( s )
            s* = Localf( s' )
            if( Accept( s* ) == true )
                 s = s'
     endLoop
```

3.3 DIVERSIFICATION V. INTENSIFICATION

When exploring the search space, it is important for the ILS procedure to adequately search local regions. It is also important for the ILS procedure to not spend too much of its computational efforts around local optima, effectively limiting the search to a few regions of the domain. The former need describes the idea of intensification (ensuring that the process thoroughly inspects each local minima) of the search, while the latter describes the idea of diversification (making sure the process is not searching a subset of the domain). The effective balancing of intensification and diversification is one of the largest hurdles encountered when tailoring any metaheuristic for a specific problem. Multiple strategies exist for accommodating both, and a few of them are discussed in the following sections.

3.4 ILS Components

This section explores the components that compose the ILS heuristic. These include determining the starting solution, a black-box search, a method of perturbing a solution and a method to determine whether a solution should be accepted for the next iteration.

3.4.1 Starting Solution

The initial solution used in the ILS is typically found one of two ways: a random solution is generated or a greedy construction heuristic is applied. In applying a greedy heuristic, better solutions can be found in a shorter amount of time. It has been shown, however, that this is true only in the short-term. Longer running algorithms see no significant difference in solution quality based on the initial solution [Stutzle, 1998].

3.4.2 Black-Box Local Search

Ideally, the local search that provides the backbone of the ILS method should always return a local optimum and it should find that local optimum as efficiently as possible. Since this step is usually the most time consuming and since it occurs at each iteration of the metaheuristic, a slow local search can lead to poor performance of the overall method.

3.4.3 PERTURBATION SCHEME

The perturbation scheme takes a locally optimal solution, s* and produces another solution to start the local search from in the next iteration. In the best case, the perturb procedure will result in a solution outside of the visited basins of attraction. That is, it will be "near" a previously unvisited local optimum. Choosing the correct perturbation scheme is important, because it has a great influence on the intensification/diversification characteristics of the overall algorithm. Perturbation schemes are commonly referred to as "strong" and "weak", depending on how much they affect the solution that they change. A perturbation scheme that is too strong has too much diversity and will reduce the ILS to an iterated random restart heuristic. A perturbation scheme that is too weak has too little diversity and will result in the ILS not searching enough of the search space. The perturbation scheme should be chosen in such a way that it is as weak as possible while still maintaining the following condition: the likelihood of revisiting the perturbed solution on the next execution of $local_f$ should be low [Lourenco, 2001]. The strength should remain as low as possible to speed up execution time. The desired perturbation scheme will return a solution near a locally optimal value. If this is the case, the local search algorithm should take less time to reach the next locally optimal value.

Components from other metaheuristics have been incorporated into the perturbation phase. [Battiti, 1997] use memory structures similar to tabu search to control the perturbation. In doing so, one can force intensification when globally "good" values are reached and force diversification when the search stagnates in an area of the search space. Borrowing from simulated annealing, temperature controlled techniques have been used to force the perturbation

to change in a deterministic manner. Basic variable neighborhood search employs a deterministic perturbation scheme.

3.4.4 ACCEPTANCE CRITERIA

When the current solution, *s*, is perturbed, the result is the new solution *s′*. *s′* is then passed to the black-box search heuristic. The resulting local optimum must pass acceptance criterion for *s′* to be designated as the new "current solution". Just as perturbation can range from too much intensification (no perturbations) to too much diversification (perturb all elements of the solution), acceptance criterion choices affect the search in a similar way. The most dramatic acceptance criterion on the side of diversification is to accept all perturbed solutions. This type of practice can undermine the foundations of ILS, since it encourages a random-walk type search. Contrasting with this is to accept only solutions that are improvements to the globally optimal value. Many implementations of ILS employ this type of acceptance strategy [Rossi-Doria, 2002]. This type of criterion, especially with a weak perturbation scheme, can restrict the search from escaping the current basin of attraction. Moreover, with this type of scheme the probability of reaching the same locally optimal value increases – a trait that reduces the algorithm's overall effectiveness. Large perturbations are only useful if they can be accepted. This only occurs if the acceptance criterion is not too biased towards better solutions [Lourenco, 2001]. The tabu search relies on occasionally moving the search into areas with worse objective functions in order to better search the solution space. [Stutzle, 1998] shows that acceptance criteria that accept *some* worse solutions outperform their best-only counterparts.

3.4.5 Stopping Criteria

Generally, the algorithm executes until one of the following conditions is met:

- A predetermined number of cycles have occurred
- The best solution has not changed for a predetermined number of cycles
- A solution has been found that is beyond some predetermined threshold.

Notice that the 3 methods constitute 3 different approaches: The first method executes independently of the performance of the process (time). The second method stops executing when the performance of the method stops improving (performance). The third method stops executing when a solution is found that is "good enough" (utility).

3.5 Adaptive Memory Programming

The term Adaptive Memory Programming (AMP) was first used in [Glover, 1997]. Here it refers to long term memory strategies that can be applied to tabu search. These memories, Glover proposes, can be used to intensify and diversify the search in a more effective way than is possible with short-term memory alone. In [Taillard, 1998] an investigation into heuristics that utilize a form of memory is undertaken. These heuristics included genetic algorithms, scatter search, tabu search and ant colony optimization. By their investigation into these memory utilizing methods, a new, more general understanding of the term adaptive memory programming surfaces. The characteristics common to heuristics that employ memory structures are the following:

1) a set of solutions/solution attributes or an aggregation of the solutions and their attributes is memorized
2) a provisory solution is constructed using this information
3) the provisory solution is improved upon, typically with some well-known heuristic
4) the memory is updated with information from the solution

AMP, when defined this way, is found in many of the most successful metaheuristic strategies [Taillard, 1998]. Typically, solutions in step 2 are constructed by taking elements of different solutions from step one and combining them (usually with some linear operator). The data structures holding the solutions or the solutions themselves are then modified in step 4 after a local search has been applied.

[T. Stutzle 1998] asserts that "incorporating memory into ILS improves performance". Our investigation into nature-inspired heuristics shows us that each relies on an element of randomization and an element of memory. Since ILS is generally implemented as a stochastic search, it provides the element of randomization needed. Many metaheuristics, including those inspired by nature, can be considered as adaptive memory programs. Incorporating a form of adaptive memory into the ILS will allow us to create an algorithm that incorporates these ideas.

CHAPTER 4 THE SNAKE-IN-THE-BOX PROBLEM

This chapter presents an overview of the Snake-in-the-box (SIB) problem.

4.1 DEFINITIONS AND NOTATION

Let *G* represent a finite, non-directed graph. Then *V(G)* and *E(G)* are the vertex and edge sets of *G*, respectively. A path *P* in *G* is defined as a subgraph of *G*, where $V(P) = \{x_0, x_1, x_2, ...x_k\}$ and $E(P) = \{x_0x_1, x_1x_2, x_2x_3, ...x_{k-1}x_k\}$. When $x_0 = x_k$, the path is said to be a *circuit.* A *chord* defined on the path *P* in the graph *G* is an edge $e \in E(G)$ such that *e* is not an element of *E(P)* and its defining vertices are elements of *V(P).* Thus a chord is and edge not in the path, but whose vertices are in the path.

Hypercubes

Q is a standard cube if $|V(Q)| = 8$, $|E(Q)| = 12$ and each vertex, $v \in V(Q)$ has degree 3. Figure 4.1 depicts a cube graph.

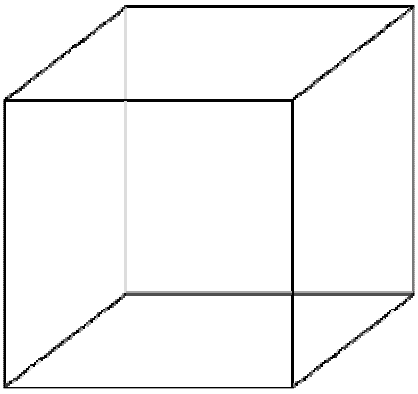

Figure 4. 1 – A three dimensional hypercube

Similar to the graph of a cube, a *hypercube* is an extension of this idea into dimensions 4 and greater. A hypercube of dimension d is an undirected graph on 2^d vertices and $2^{d-1} \cdot d$ edges, where each vertex is connected to exactly d other vertices. While the definition of a hypercube holds for any positive d, the word hypercube is typically reserved for graphs with $d > 3$. Hypercubes of dimension d will be represented by Q^d. Figure 2 depicts a hypercube of dimension 4 (Q^4). For a more detailed description, we direct the reader to [Harary, 1988].

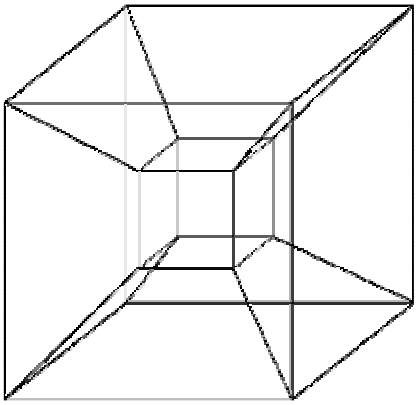

Figure 4. 2 – A four dimensional hypercube

A useful property of hypercubes is that the nodes can be mapped to binary sequences of length *d*. In this mapping, two vertices in a hypercube are adjacent if and only if their binary sequences differ by exactly one bit. Figure 4.3 depicts the standard cube when this mapping is applied. From this point on, we will not distinguish between a vertex on the hypercube and its binary sequence.

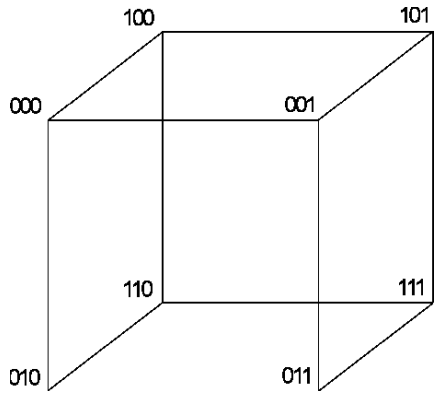

Figure 4. 3 – A standard cube with binary encoding of vertices

It is interesting to note that a hypercube of dimension *d* can be constructed from two hypercubes of dimension *d-1*. This construction is performed by connecting similar vertices. For example, consider the following figure, where a hypercube of dimension 4 is constructed from two hypercubes of dimension 3. Then the bit representation of each node can be updated by adding a '1' to each node originally in the left dimension 3 hypercube and by adding a '0' to each node originally in the right dimension 3 hypercube.

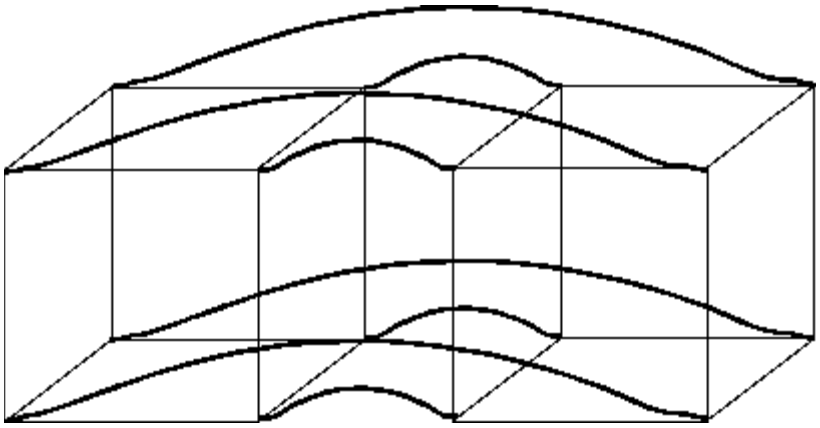

Figure 4. 4 – A four dimensional hypercube depicted as the joining of two three dimensional hypercubes

4.2 SNAKES

A *d-dimensional snake* is defined as a path S on Q^d such that S is achordal. The following figure depicts a snake in dimension 3.

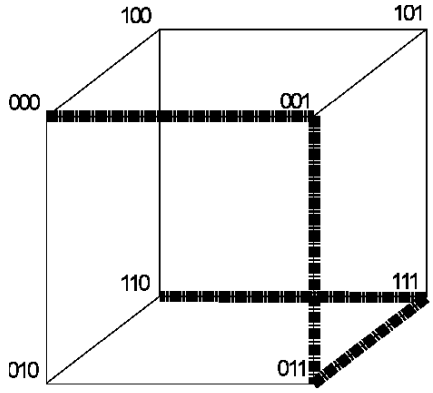

Figure 4. 5 – A snake of length 4 on the standard cube

Similarly, a *d-dimensional coil* is a achordal circuit through Q^d. Figure 4.6 depicts a coil in dimension 3.

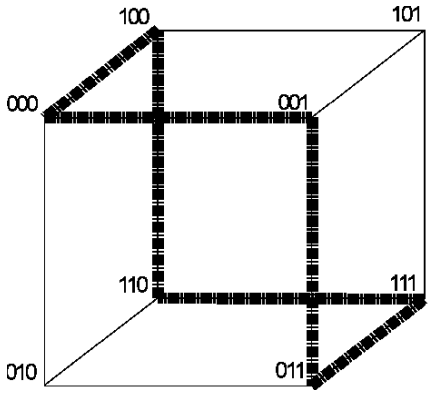

Figure 4. 6 – A coil of length 6 on the standard cube

Throughout this paper, a *d*-dimensional snake will be referred to as a *d-snake* and similarly, a *d*-dimensional coil will be referred to as a *d-coil*. A *maximal snake* is a snake that violates the definition of a snake when it is extended by adding any vertex to the front or back of its path sequence. While coils and snakes are closely related in their definitions and applications, this thesis does not address issues pertaining directly to coils.

A *longest snake* in Q^d is the snake with the largest possible number of vertices. *S(d)* will denote the longest snake in dimension *d*. Similarly, a longest *d*-coil will be denoted by *C(d)*. Longest snakes have been determined for hypercubes up to dimension 7. Longest snakes for dimensions > 7 remain an open problem.

Much attention has been given to determining theoretical bounds for *C(d)* (a bound for *S(d)* can easily be derived from this by subtracting two). Kautz showed a lower bound in [Kautz, 1958], namely that

$$C(d+2) \geq 2C(d)$$

from which it follows that

$C(d) > \lambda 2^{d/2}$, where λ is a positive constant

Various researchers improved on this bound with [Abbot, 1991] constructing the current best lower bound, expressed as

$$\frac{77}{256}2^d \le C(d)$$

The current tightest upper bound for dimensions 7 <= *d* <= 11 was given in [Snevily, 1994] and is expressed as

$$C(d) \le 2^{d-1}\left(1-\frac{1}{(d^2-d+2)}\right)$$

As pointed out by [Rajan, 1999], these bounds leave a large margin of error: $\Theta(2^d / d^2)$. This analysis of *C(d)*, while interesting, does not directly apply to *S(d)* [Potter, 1994]. The only bound that can be directly derived is the one for which a *d*-snake is constructed directly from *C(d)*, giving

$$S(d) \ge C(d) - 2$$

[Abbott, 1991] increased the lower bounds for coils in dimension 8-20 using proof by construction techniques. [Paterson, 1998] used a technique based on arranging equivalence classes of the snakes to improve many of these bounds. The current lower bounds found non-computationally for *S(d)* are shown in table 4.1.

Table 4. 1 – Upper and lower bounds for *S(d)* in dimensions 7 to 11

D	Calculated Lower Bound	Computed Lower Bound	Calculated Upper Bound
7	46 [Kautz, 1958]	50 [Kochut, 1996]	60 [Snevily, 1994]
8	96 [Paterson, 1998]	97 [Rajan, 1997]	123 [Snevily, 1994]
9	168 [Abbott, 1991]	186 [Casella, 2005]	250 [Snevily, 1994]
10	338 [Paterson, 1998]	358 [Casella, 2005]	504 [Snevily, 1994]
11	618 [Paterson, 1998]	680 [Casella, 2005]	1012 [Snevily, 1994]

The value for *S(d)* in dimension 7 is exact – it was determined computationally through exhaustive search.

The current best value of *S(8)* of 97 was determined computationally by [Rajan, 1999] using a technique that incorporated maximal snakes from smaller dimensions. Beginning with [Potter, 1994], heuristic approaches have been applied to the problem, aiming to improve lower bounds of SIB codes by constructing examples. Recently, [Casella, 2005] reported using a stochastic hill-climber to find values of 186, 358 and 680 for dimensions 9, 10 and 11, respectively. Figure 4.7 depicts the effect that computation has had on tightening the bounds on *S(d)* for dimensions 5 to 11.

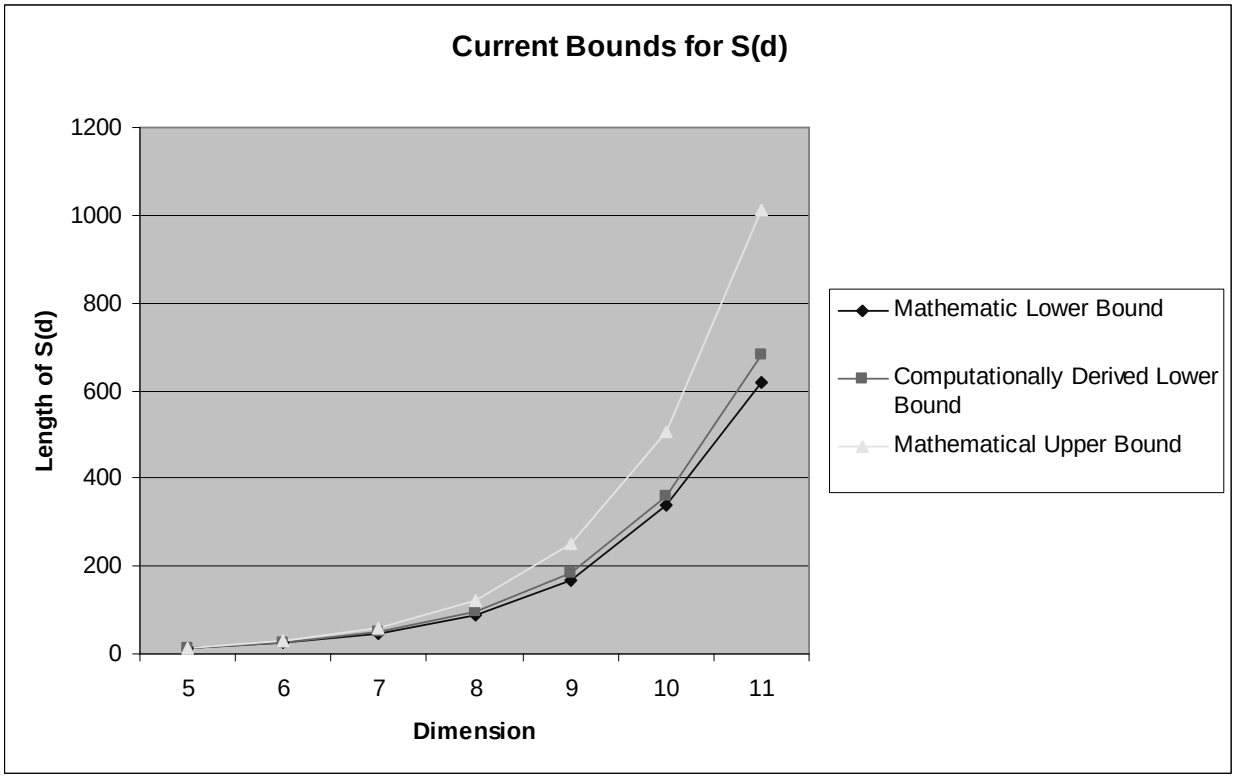

Figure 4. 7 – A comparison of *S(d)* bounds obtained computationally & non-computationally

4.3 Applications of Snake-In-The-Box Codes

Snakes and their closed counterparts, coils, are often referred to as *circuit codes of spread 2*. Interest in snake-in-the-box codes began with [Kautz, 1958] in the context of unit distance, error checking codes. By using the visited nodes of the hypercube as code words, SIB codes found application in analog to digital conversion [Klee, 1970]. If a codeword has one bit corrupted in the communication process, it either corrupts to a neighbor on the path (negligible errors) or a non-codeword (error detected). Similarly, if a subset of a SIB code is chosen in such a way that code words differ by at least one bit, the result is an error-correcting code (for single-bit errors). SIB codes have found application in other areas as well. They have been used in

constructing electronic locking schemes in [Kim, 2000] and [Dunin-Barkowski, 1999] uses a SIB code in a cerebellar learning model.

CHAPTER 5 AN OUTLINE OF THE PROCESS

5.1 INTRODUCTION

The task of finding SIB codes is equivalent to finding circuit codes of spread 2. Highest-quality solutions to this problem are those snakes with the longest path lengths. The longer the path, the more useful the code is in applications [Klee, 1970]. Aside from the lengths of the snakes, the execution time is also an important characteristic to consider when evaluating the usefulness of a particular method. This chapter begins by describing a simple snake construction heuristic. We then show how this heuristic can be used within the Iterated Local Search metaheuristic. We then present an adaptive memory based heuristic that utilizes the ILS procedure developed.

5.2 SNAKE CONSTRUCTION HEURISTIC

In order to implement an Iterated Local Search, one needs some type of local search heuristic to iterate over. That local search heuristic should be computationally efficient, since it will be executed repeatedly throughout the ILS. Previous heuristic approaches to finding maximal snakes rely on populations of feasible paths in the hypercubes. These paths must then be inspected to see if they lie in the domain of feasible snakes. This check for feasibility is typically carried out in an objective function evaluation for the snake at each iteration of the algorithm. Objective function values are typically dependant upon the maximal length snake

contained in the sequence. This is usually realized as a function that, beginning with the first node of the sequence, checks each subsequent node for feasibility until a non-feasible node is found. The number of feasible nodes found before this point constitutes the length of the snake. Given a node sequence of length *n*, the average objective function evaluation is on the order of $O(n^2)$. Another commonly employed method for determining the objective function value of a node sequence is to treat the sequence as a circular list, and then finding the longest snake within that list. Computing the objective function this way means that the previously detailed objective function must be executed *n* times, resulting in an operation on the order of $O(n^3)$.

Here, we take a different approach. Instead of constructing a feasible *path* in the hypercube and then determining its fitness, we construct a feasible *snake*, updating its fitness as each node is added to its end. The result is a local search heuristic that restricts its search to those valid paths through the hypercube that constitute snake-in-the-box codes. In order to accomplish this task, information about the state of the hypercube that the current solution lies in must be recorded. More particularly, an induced subgraph is constructed alongside the solution. The induced subgraph is a subgraph of the hypercube, consisting of all nodes and edges contained in the current solution, as well as all possible chordal nodes and edges to the current path of the solution.

5.3 Solution Representation

Node Sequence

An array of integers, in the range $[0,2^d)$, represents the path as a sequence of nodes of the hypercube. The length of this sequence corresponds to the length of the snake. One such possible node sequence of length 6 in dimension 4 is

{0, 1, 3, 7, 15, 6}

Transition Sequence

In [Potter, 1994], a GA is employed to find high-quality solutions to the SIB problem. In doing so, the traditional method of representing a path as a sequence of nodes on the graph is replaced by a representation of the path as a sequence of *transitions*. The transition sequence, originally introduced by [Abbott, 1991] can be constructed from the node sequence as follows:

```
for( int i = 1; i < nodeSequence.length; i++ ){
 transitionSequence[i]=log2(nodeSequence[i-1]^nodeSequence[i]);
}
```

Similarly, the node sequence can be reconstructed from the transition sequence as follows:

```
for( int i = 0; i < transitionSequence.length; i++ ){
 nodeSequence[i+1]=(nodeSequence[i]^(1<<transitionSequence[i]));
}
```

The transition sequence takes advantage of the symmetry of the hypercube since it is possible for a single transition sequence to describe multiple node sequence paths by beginning each sequence at a different node of the hypercube (of course, this can be achieved with a node sequence by fixing the first element in the sequence). More importantly, any modification to the transition sequence will result in a valid path. That is not to say, however, that any modification to the transition sequence will result in a valid snake. The aforementioned snake represented as a transition sequence would appear as

{0, 1, 2, 3, 0}

5.4 INDUCED SUBGRAPH

As mentioned earlier, each solution keeps an induced subgraph constructed from the current path. This graph contains all of the nodes of the path, as well as any nodes directly

adjacent to the graph. Thus, when a node is to be added to the transition sequence, each of the *d* possible nodes to append can be checked for viability via the induced graph.

```
boolean procedure tryAddEnd( transitionNode end ){
    // the Transition Node to add, converted to a Node Sequence
    // Node
    NodeSequenceNode endNode = convertToNodeSequenceNode( end );
    // for each dimension
    for (i = 0 to dimension)
      // check to see if the node to add will form a chord
      // omit checking if i == end, b/c that refers to the
      // previous node in the path
      if( i != end && inducedGraph.canFormChord( endNode ) )
          return false;
    return true;
}
```

As previously mentioned, evaluating the length of the snake beginning with the first node is an operation on the order of $O(n^2)$, where n is the length of the sequence. Each node must be compared with all other nodes in the sequence to determine feasibility. In our construction heuristic, once a maximal snake has been found (no nodes exist that can be added while maintaining solution feasibility), the effective runtime for constructing the solution is on the order of $O(dn)$, where n is the length of the snake and *d* is the dimension.

Another reason for choosing the construction technique stems from the uncertainty involved when assigning a SIB code an objective function value. The most obvious choice is the length of the snake. This is problematic because the cardinality of the set of maximal snakes of length n in a given dimension can be unknown. This evaluation does not provide a practical way to distinguish solutions in a way suitable for intelligent search through the solution space. That is, the range of the objective function is too small to make intelligent choices in a straightforward manner. [Casella, 2005] proposed a modified objective function where “tightly-coiled” (that is,

they left the most nodes of the graph free for future extension of the snake) snakes were given a higher fitness value. While this helps to granulate the range of objective function values, it does not do so by much and could possibly discourage optimal solutions from being found. Since a straightforward objective function evaluation does not seem to add additional insight to our search, we use the construction technique.

In constructing a SIB code, feasible nodes are appended to the solution randomly. We decided to make this process random primarily to keep the process as computationally efficient as possible. A randomized search, while not "intelligent", does benefit from low computing overhead and avoids some of the pitfalls of a deterministic search (cycling, for example). These things, coupled with the fact that so little is known about the structures of optimal SIB codes influenced this decision.

As covered in the Chapter 3**,** the ILS algorithm is composed of four basic elements: a function to generate an initial solution, a function to perturb the current solution, a black-box search procedure and a set of acceptance criteria. The SIB construction heuristic outlined above will act as the black-box local search procedure in our ILS algorithm.

```
Iterated Local Search
Let:
f    the function to minimize
Local_f    a local search with the objective of minimizing f
S    the set f is defined on
s    an element of S
S*   the set of f(s), or local optima
s*   an element of S*
Perturb   Perturbation function
Accept    Acceptance Criterion function
Then:
Choose an initial solution s0 in S
s* = L_f(s0)
s=s*
Loop
     s' = Perturb( s )
     s* = Localf( s' )
```

```
        if( Accept( s* ) == true )
             s = s*
    end Loop
```

5.5 ITERATED LOCAL SEARCH

Section 5.4 discusses how each of the elements of the iterated local search were implemented.

5.5.1 INITIAL SOLUTION

The starting solution is the transition sequence {0,1,2}. This is chosen as the seed since all SIB codes are isomorphic to a SIB code that originates from {0,1,2} [Kochut, 1996].

5.5.2 PERTURBATION SCHEME

Once a solution reaches a maximal state (no nodes can be added to the end while maintaining its feasibility status), nodes are removed from the end. This constitutes the perturbation scheme. How many nodes to remove in a single perturbation of a solution is a function of the length of the current solution. The number of possible nodes to remove ranges between 1 and n, where removing one node returns the solution to its state on the previous iteration and removing n nodes returns the solution to its state on the first iteration. A linear probability distribution dictates the likelihood of any particular perturbation being chosen. Therefore, the average perturbation will cut each of the solutions in half. Thus, an implicit

intensification occurs around better fit solutions. This type of scheme encourages exploration around each solution, while still maintaining the possibility of large perturbations.

5.5.3 Acceptance Criteria

The most dramatic acceptance criterion on the side of diversification is to accept all perturbed solutions. This undermines the foundations of ILS, since it reduces the search to a random-walk. Similarly, the most dramatic acceptance criteria on the side of intensification is to accept only those solutions that are better than the best value found. This type of criterion, especially with a weak perturbation scheme, can restrict the search from escaping the current basin of attraction. [Stutzle, 1998] shows that acceptance criteria that allow *some* worse solutions outperform their best-only counterparts.

If the solution resulting from a local search on the perturbed solution is at least as good as the best solution, the perturbed solution is accepted for the next iteration. Thus, worse moves are allowed, since multiple snakes with the same lengths may have different potential for aiding the search.

While the process outlined above proves an efficient method to construct snakes, the lack of intelligent decision making makes it little more than a random search. The section that follows outlines a strategy for incorporating memory into the search in order to intensify the search in an intelligent way.

5.6 PRINCIPLE OF PROXIMATE OPTIMALITY

As illustrated by [Rajan, 1999], snakes of dimension *d* are often composed of maximal snakes of dimension *d*-1. This insight allows the exploitation of a technique employed by certain tabu search algorithms – the principle of proximate optimality. In [Glover, 1997], the proximate optimality principle is outlined. It states that "good solutions found at one level are likely to be found close to good solutions at an adjacent level". In tabu search, this principle is realized by searching within one level for a set number of iterations and then restoring the best solution found before proceeding to the next level.

This type of strategy is realized here by searching first for suitable maximal snakes in dimension *d* and using these solutions as starting solutions for the search in dimension d+1. Given the symmetry of the hypercube and, particularly the ability to form a hypercube of dimension *d* from two hypercubes of dimension *d*-1, this is an intuitive strategy.

As noted by [Taillard], the characteristics common to heuristics that employ memory structures are the following:

1. a set of solutions/solution attributes or an aggregation of the solutions and their attributes is memorized
2. a provisory solution is constructed using this information
3. the provisory solution is improved upon, typically with some well-known heuristic
4. the memory is updated with information from the solution

As an example, assume that we are interested in finding s(7). Then we can utilize the ILS described above, where the initial solution is {0,1,2} to do so. A more powerful alternative, however, is to seed our ILS in dimension 7 with a maximal snake from dimension 6. This still leaves the question of *which* dimension 6 snake to use. Our approach builds a population of the

longest snakes at each dimension and repeatedly uses them as seeds in an ILS. Each of the snakes is reduced to its isomorphic equivalent up to reversal as in [Rajan, 1999]. This reduces the search space by ensuring that any two snakes that we search are not permutations of one another. The likelihood that a snake will be chosen as the initial solution of a search in the next dimension is dependent on its own length and how well it has performed in the past as an initial solution. That is, given two dimension 6 snakes, s_1 and s_2, of the same length, if s_1 has resulted in finding better dimension 7 snakes it will be chosen with a higher probability and vice versa. This is realized by keeping the solutions sorted, first by length and then by their performance. A linear probability distribution based on this ordering is used to determine the likelihood that a solution will be chosen as the next seed.

```
s    = current solution
ILS  = Iterated Local Search
SPi  = pool of dimension i solutions

Repeat:
     for ( int d = lowestDimension to highestDimension )
          if( d == lowestDimension )
               // perform ILS with {0,1,2}
               s = ILS( {0,1,2} )
          else
               // perform ILS with a seed from the previous
               // dimension
               s = ILS( SP(d-1).getSolution() );
               // update status of s in the d-1 dimension
               // solution pool
               SP(d-1).updateSolution(s);
          // add s to the pool of dimension d solutions
          SP(d-1).add(s);
```

It should be noted that the lowest dimension searched in this scheme does not benefit from having a population of solutions to begin its search with. Instead, the solution {0,1,2} is used. Therefore, consideration must be taken to ensure that the ILS can properly explore this

dimension without the aide of a population of solutions. In our experiment, we determined that the ILS could easily explore dimension 6 (this is discussed in the following chapter), so we utilized it as the starting point of our searches.

5.7 Parameter Tuning

In the process outlined above, consideration must be given to two things: how long to execute each ILS and how large to make each solution pool. Making each ILS run for too many iterations will result in more intensification around solutions, but take more computation time that could be spent exploring other areas. Similarly, a solution pool that contains too few solutions will restrict the search to certain areas, while a solution pool with too many solutions may cause the search to avoid necessary intensification around potential optimum values. These decisions are made empirically and will be discussed in the results chapter. Stopping criteria for the procedure outlined above consists of executing either for a predetermined number of iterations or until a suitable solution has been found.

CHAPTER 6 RESULTS

6.1 ITERATED LOCAL SEARCH

Prior to incorporating the ILS into the AMP algorithm, it was necessary to inspect its performance alone. The results below were obtained by executing the ILS alone with initial solution {0,1,2}. The following table outlines our results.

Table 6.1 – *S(d) r*esults for ILS with no memory

Dimension	***Average Best***	***Overall Best***	***Current Lower Bound, S(d)***
6	26	26	26
7	*47.1*	*48*	*50*
8	79.7	84	>=97
9	*131.6*	*132*	*>=186*

As can be seen from the results, the ILS performs well in dimension 6 (consistently converging to *S(d)* in a few seconds), but fails to sufficiently explore problem instances of greater dimension. For this reason, dimension 6 is chosen as the vantage point for the AMP algorithm.

6.2 ITERATED LOCAL SEARCH WITH ADAPTIVE MEMORY

When incorporating adaptive memory into the ILS as described in chapter 5, two main parameters must be tuned: the solution pool size and the stopping critera for the ILS at each iteration of the AMP algorithm. Increasing the solution pool size diversifies the search by allowing more seeds to be explored. As a consequence, the average length of snakes in this list grows at a rate slower than that of a solution pool of a smaller size. This leads to more time being spent searching from seeds with shorter lengths. The length of the seed, however, is not the only criteria that determines whether it is a worthy starting point for a search.

The program was executed with solution list sizes of 5, 10, 25, 50 and 100. For each of these instances the program was executed with varying stopping criteria in the embedded ILS. The ILS stopping criteria were to stop after 5, 10, 25, 50 and 100 iterations. Each of the 25 instances described here were executed 10 times for approximately 30 minutes each.

Table 6.2 outlines the consequences of varying the solution list size in the algorithm. Each value in column 2 represents the average found over all instances with varying stopping criteria.

Table 6. 2 – ILS with AMP results, varying solution list size, dimension 7

Solution List Size	Average Best Snake Found	Worst Snake Found Upon Termination	Best Snake Found Upon Termination
5	48.75	48	50
10	49.05	48	50
25	48.85	48	50
50	49.3	49	50
100	50	50	50

As one can see, increasing the solution list size allows for the type of diversification necessary to achieve optimal values. There is an almost steady progression of solution quality with the increase. Table 6.3 outlines the same situation in dimension 8.

Table 6. 3 - ILS with AMP results, varying solution list size, dimension 8

Solution List Size	Average Best Snake Found	Worst Snake Found Upon Termination	Best Snake Found Upon Termination
5	94.15	91	97
10	94.15	92	97
25	94.3	94	97
50	94.8	93	97
100	95.25	94	97

The results of increasing the solution list size in dimension 8 mirror those of dimension 7. There seems to be a direct correlation between the solution size and solution quality. As the two previous tables show, finding the best snake does not seem to be effected by the list size at all. Figure 6.3 outlines the same situation in dimension 9.

Table 6. 4 - ILS with AMP results, varying solution list size, dimension 9

Solution List Size	Average Best Snake Found	Worst Snake Found Upon Termination	Best Snake Found Upon Termination
5	166.9	164	174
10	165.9	163	171
25	167.2	164	172
50	168.4	165	174
100	168.2	164	170

In dimension 9, the situation has changed somewhat. While there does seem to be a slight correlation with the solution list size and solution quality, the relation is not straightforward. The success of the instances with solution list = 5 can be attributed to the need for more intensification around solutions in dimension 9. While some dimension 8 snakes may provide seeds that extend to dimension 9 snakes in the 170's easily (that is, the number of possible paths allowed from the snake is small relative to the number of possible paths allowed from other snakes of the same length), other dimension 8 snakes may require ample exploration to achieve such results. Therefore, in keeping a small solution list size, these dimension 8 snakes are extended enough times to achieve this.

Decreasing the amount of time each ILS executes diversifies the search by allowing more time to be spent on a variety of seeds. The following table outlines the effects in dimension 7 of varying the stopping criteria of the ILS within the algorithm. The ILS executed at each iteration of the AMP algorithm for 5, 10, 25, 50 or 100 iterations.

Table 6. 5 - ILS with AMP results, varying ILS stopping criteria, dimension 7

ILS Iterations	Average Best Snake Found	Worst Snake Found Upon Termination	Best Snake Found Upon Termination
5	49.15	48	50
10	49.12	48	50
25	49.05	48	50
50	49.19	48	50
100	49.0	48	50

In dimension 7, the stopping criteria of the ILS appears to have a minimal effect on the performance of the algorithm.

Table 6. 6 - ILS with AMP results, varying ILS stopping criteria, dimension 8

ILS Iterations	Average Best Snake Found	Worst Snake Found Upon Termination	Best Snake Found Upon Termination
5	93.7	92	96
10	94	92	96
25	94.75	94	96
50	94.7	94	97
100	95.2	94	97

The results in dimension 8 are more forthcoming with establishing a relationship between the stopping criteria of the ILS and the solution quality found. More intensification improves the average performance of the algorithm and allows for the best solutions to be found.

Table 6. 7 - ILS with AMP results, varying ILS stopping criteria, dimension 9

ILS Iterations	Average Best Snake Found	Worst Snake Found Upon Termination	Best Snake Found Upon Termination
5	166.0	164	173
10	166.2	164	171
25	167.8	166	171
50	169	166	171
100	170.3	166	174

The effects of increasing the number of ILS iterations at each step of our AMP algorithm are more obvious in dimension 9. There is an established correlation between solution quality and the number of iterations performed. All of the best solutions are found when the ILS performs more consecutive iterations.

So we see that increasing the solution pool size and increasing the number of ILS iterations has a direct affect on the quality of solutions obtained. It is a mistake, however, to assume that both parameters should be tuned according to these findings only. Since each of these parameters effects the overall intensification and diversification of the search, we felt that they were closely related. Thus, we found it necessary to tune them simultaneously.

Since solutions of high quality seem to be easily obtained for dimension 7, the following section focuses on optimizing the parameters for snake hunting in dimensions 8 and 9. The following table shows the average best snake found from 10 executions of our algorithm with varying ILS stopping criteria and solution pool size.

Table 6. 8 - ILS with AMP results, stopping criteria & solution pool size, dimension 8

		ILS Iterations				
		5	10	25	50	100
Solution Pool Size	5	93	93	94.75	95.25	95.25
	10	93.25	93.25	95.75	95	95
	25	94.25	94.75	94	94.5	96
	50	95.25	95	95	***96.5***	95.75
	100	95.25	95	95	96.25	95.25

As can be seen from the table, the best overall values are obtained when a balance is reached between intensifying the search through more ILS executions and diversifying the search through larger solution pool sizes.

The following table outlines the same scenario in dimension 9. Here, the best results are obtained when the solution pool is small and the ILS iterations are large. This would imply that more intensification is needed around solutions in this dimension. Comparable solutions are obtained with larger solution pool sizes. Shorter ILS durations, however, result in degradation of the solution quality.

Table 6. 9 - ILS with AMP results, stopping criteria & solution pool size, dimension 9

		ILS Iterations				
		5	10	25	50	100
Solution Pool Size	5	165	165.2	167.1	168.2	172.6
	10	165.6	168	168.3	169.1	168.1
	25	165.1	165.6	171.2	169.7	171.7
	50	169.4	169	170	168.9	171.2
	100	165.4	167	169.8	170.2	170

Our results with the best parameters for dimensions 8 and 9 are presented in the following tables.

Table 6. 10 - Dimension 8, ILS iterations = 50, Solution Pool size = 50

Snake Length	Iterations of AMP	Iterations of ILS	Time(s)
77	1	50	0
78	2	100	0
81	3	150	0
82	5	250	0
84	25	1250	1
85	63	3150	2
86	94	4700	3
90	201	10050	5
91	744	37200	20
92	1146	57300	30
93	1690	84500	44
95	4632	231600	120
97	28528	1426400	877

Table 6. 11 - Dimension 9, ILS iterations = 5, Solution Pool size = 100

Snake Length	Iterations of AMP	Iterations of ILS	Time(s)
143	1	100	0
149	7	700	1
151	1	1000	1
152	19	1900	2
155	53	5300	6
156	58	5800	7
157	288	28800	33
158	526	52600	60
159	767	76700	88
164	1171	117100	135
165	1645	164500	190
166	2193	219300	254
167	2427	242700	2819
168	3657	265700	4275
175	9889	988900	11646

CHAPTER 7 CONCLUSIONS AND FUTURE DIRECTION

7.1 CONCLUSIONS

Given the symmetry of the hypercube in any dimension, the snake-in-the-box problem would seem to have an intuitively symmetric solution. Unfortunately, this is not the case. [Aichholzer, 1997] notes that conjectures proved about hypercubes in certain dimensions are rarely generalized for hypercubes of all dimensions. The snake-in-the-box problem, because of its ties to the hypercube, suffers the same condition. It is because so little can be determined about snakes mathematically, heuristic approaches to the problem have been successful in tightening the bounds of maximal snakes in several dimensions [Potter, 1994][Casella, 2005].

The performance of this approach is encouraging since all of the previous heuristic approaches to finding maximal snake-in-the-box codes rely on population-based searches. Population-based heuristic approaches are good at exploring the solution space since they search from a set of designs and not from a single design. Single-point searches, however, are more equipped to perform finely tuned searches. Iterated Local Search is a single-point search that incorporates elements of a population-based search in that it is able to make large jumps in the solution space. The method employed here relies on the incorporation of maximal snakes found in previous dimensions into the search.

7.2 FUTURE DIRECTION

More sophisticated strategies could be incorporated into the adaptive memory used in this implementation. For example, the size of the solution pool used could be adaptive, increasing when diversification is needed and decreasing when intensification is needed. The size of the solution pool could also differ across the varying instances of the search. That is, the size of the dimension 6 solution pool could differ from the size of the dimension 7 solution pool and so on. Additionally, elements of the ILS could be altered to improve performance. The perturb scheme could be changed to an adaptive one, so that snakes found during the ILS would be perturbed even more around poor solutions (diversifying) and even less around good solutions (intensifying).

Additionally, the ILS method outlined here could be incorporated into other, existing snake-finding strategies. Since the ILS only explores feasible snakes that can be extended from a base snake, it can be a powerful (and computationally fast) technique for performing a local search around an optimal value.

This experiment is well suited to be parallelized. Each dimension could be searched via a separate thread of execution, or multiple systems could be utilized on the same dimension, all sharing in and updating the same pool of solutions.

REFERENCES

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

H. L. Abbot and M. Katchalski, *On the construction of snake in the box codes.* In Utilitas Mathematica, 40:97-116, 1991.

O. Aichholzer, *Combinatorial & Computational Properties of the Hypercube - New Results on Covering, Slicing, Clustering and Searching on the Hypercube*. PhD thesis, IGI-TU Graz, Austria, 1997.

S. Areibi, *Towards Optimal Circuit Layout Using Advanced Search Techniques.* PhD Thesis, University of Waterloo, 1995.

S. Areibi, M. Moussa, H. Abdullah, "A Comparison of Genetic/Memetic Algorithms and Heuristic Searching", *Submitted to the 2001 International Conference on Artificial Intelligence IC-AI 2001*, Las Vegas, NV, June 25 2001.

R Battiti and G. Tecchiolli, "The reactive tabu search", *ORSA Journal on Computing 6,* 126-140, 1994.

R. Battiti and M. Protasi, "Reactive search, a history-based heuristic for the MAX-SAT." *ACM Journal of Experimental Algorithmics*, 2, 1997.

M. L. den Besten, T. Stützle, and M. Dorigo. "Design of iterated local search algorithms: An example application to the single machine total weighted tardiness problem." In E. J. W. Boers, J. Gottlieb, P. L. Lanzi, R. E. Smith, S. Cagnoni, E. Hart, G. R. Raidl, and H. Tijink, editors, *Applications of Evolutionary Computing: Proceedings of EvoWorkshops 2001*, volume 2037 of *Lecture Notes in Computer Science*, 441-452. Springer Verlag, Berlin, Germany, 2001.

M. Birattari, L. Paquete, T. Stützle, and K. Varrentrapp, "Classification of Metaheuristics and Design of Experiments for the Analysis of Components", *Technical Report AIDA-2001-05,* Intellektik, Technische Universität Darmstadt, Darmstadt, Germany, 2001.

D. Casella and W. D. Potter, "New Lower Bounds for the Snake-In-The-Box Problem: Using Evolutionary Techniques to Hunt for Snakes", to appear in *The 18th International FLAIRS Conference*, Clearwater Beach, Florida, May, 2005.

M. Chiarandini and T. Stützle. “An application of Iterated Local Search to Graph Coloring.” In D.S. Johnson, A. Mehrotra, and M. Trick, editors, Proceedings of CP 2002 - Eigth International Conference on Principles and Practice of Constraints Programming, 112-125, 2002.

A. Colorni, M. Dorigo, V. Maniezzo. “Distributed Optimization by Ant Colonies”, *Proceedings of the First European Conference on Artificial Life*, Paris, France, F. Varela and P. Bourgine (Eds), Elsevier Publishing, 134-142, 1992.

A Colorni, M. Dorigo, F. Maffioli, V. Maniezzo, G. Righini, and M. Trubian. “Heuristics from nature for hard combinatorial optimization problems”, *International Transactions in Operational Research,* 1-21, March 1996.

M. Dorigo and G. DiCaro. “The ant colony optimization metaheuristic”, In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, 11-32, 1999.

W. L. Dunin-Barkowski, D. C. Wunsch, “Cerebellar Learning: A Possible Phase Switch in Evolution” Proceedings of IEEE/INNS International Joint Conference on Neural Networks '99, Vol. 1, Washington, DC, 21-26, 1999.

F. Glover, “Heuristics for integer programming using surrogate constraints”, *Decision Sciences*, 8:156-166, 1977.

F. Glover, “Future paths for integer programming and links to artificial intelligence”, *Computers & Operations Research,* 13:533-549, 1986.

F. Glover and M. Laguna, *Tabu Search*, Kluwer: Boston, MA, 1997.

D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, 1989.

S. Goss, S. Aron, J. L. Deneubourg, and J. M. Pasteels, “Self-Organized shortcuts in the Argentine ant”, *Naturwissenschaften*, 76:579-581, 1989.

F. Harary, J. P. Hayes, and H. J. Wu, “A survey of the theory of hypercube graphs”, *Comput. Math Applic.* 15 (1988) 277-289.

J. Holland, *Adaptaion In Natural and Artificial Systems*. University of Michigan Press, 1975.

M. Kadluczka, T. M. Tirpak, P. C. Nelson, *Adaptive Memory Programming framework: parameterized meta-search platform*, in preparation, 2005.
http://www.cs.uic.edu/~mkadlucz/AMPv11.pdf (Accessed Jan. 20, 2005).

W. H. Kautz, “Unit-Distance Error-Checking Codes.” *IRE Trans. Electronic Computers* 7:179-180, 1958

S. Kim, D. L. Neuhoff, "Snake-in-the-Box Codes as Robust Quantizer Index Assignments." *International Symposium on Information Theory 2000*, Sorrento, Italy, June 25-30, 2000.

J. L. Kiranmayee, "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/Snakde2d4.ps (Accessed Oct. 23, 2004).

S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi, "Optimization by Simulated Annealing", *Science*, 220:671-680, 1983.

V. Klee, "What is the maximum length of a d-dimensional snake?" *American Mathematics Monthly*, 77:63-65, 1970.

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

J. Lee, *A First Course in Combinatorial Optimization.* Cambridge University Press, 2004.

S. Lin and B. W. Kernighan. "An effective heuristic algorithm for the traveling salesman problem." *Operations Research*, 21:498-516, 1973.

H. R. Lourenco, O. Marting and T. Stutzl, "A beginner's introduction to Iterated Local Search." In Proceedings of MIC'2001-Meta-heuristics International Conference. Vol. 1. Porto-Portugal, 1-6, 2001.

H. R. Lourenco, O. Martin, and T. Stutzle, "Iterated Local Search," in *Handbook of Metaheuristics*, F. Glover and G. Kochenberger, Eds. Norwell, MA: Kluwer, 321-353, 2002.

L. Paquete and T. Stützle. "Experimental investigation of iterated local search for coloring graphs." In S. Cagnoni et al., editor, *Applications of Evolutionary Computing, Proceedings of EvoWorkshops 2002*, 2279:122-131. Springer Verlag, Berlin, Germany, 2002.

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

W. D. Potter, R. W. Robinson, J. A. Miller and K. J Kochut, "Using the Genetic Algorithm to Find Snake-In-The-Box Codes." In the 7th International Conference On Industrial & Engineering Applications of Artificial Intelligence and Expert Systems, Austin TX, 421-426, 1994.

M. S. L. Prasanna and C. Swapna, "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 Nov. 1, 2004).

P. Preux, E.G. Talbi, "Towards Hybrid Evolutionary Algorithms", *International Transactions in Operational Research*, 6:557-570, 1999.

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

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

O. Rossi-Doria, M. Samples, M. Birattari, M. Chiarandini, J. Knowles, M. Manfrin, M. Mastrolilli, L. Paquete, B. Paechter and T. Stutzle. "A Comparison of the performance of different metaheuristics on the timetabling problem." In Proceedings of *PATAT 2002: The 4th international conference on the Practice and Theory of Automated Timetabling*, 115-119, Gent, Belgium, August 2002.

K. G. Scheidt, "Searching for patterns of maximal snakes." In *Proceedings of the Southeastern Conference of the Consortium for Computing in small colleges*, November 2000.

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

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

T. Stützle, "Applying Iterated Local Search to the permutation flow shop problem". Technical Report AIDA-98-04, FG Intellektik, TU Darmstadt, 1998.

T. Stutzle. *Local Search Algorithms for Combinatorial Problems – Analysis, Improvement, and New Applications.* PhD thesis, Darmstadt University of Technology, Department of Computer Science, 1998**.**

T. Stützle, A. Grün, S. Linke, and M. Rüttger, "A Comparison of Nature Inspired Heuristics on the Traveling Salesman Problem*"*. In Deb et al, editors, *Proceedings of PPSN-VI, Sixth International Conference on Parallel Problem Solving from Nature, volume 1917 of LNCS*, 661-670, 2000.

E. D. Taillard, L. M. Gambardella, M. Gendreau and J. Y. Potvin, "Adaptive Memory Programming: A Unified View of Meta-Heuristics", *EURO XVI Conference Tutorial and Research Reviews booklet* (semi-plenary sessions), Brussels, July 1998.

J. Wojciechowski, "A new lower bound for Snake-in-the-Box codes," *Combinatorica*, 9:91-99, 1989.

APPENDIX A JAVA CODE

```
import java.util.Arrays;

/**
 * The ASnakeAttribute class provides a skeletal implementation
 * of the ITabuListElement and ISolutionAttribute
 * interfaces defined in package tabusearch.  ASnakeAttribute
 *  defines an attribute to a Snake object (an induced
 * path in a hypercube).
 *
 * @author William Brown
 * @version 1.01, 10/10/2004
 */
public abstract class ASnakeAttribute extends ASolutionAttribute implements
Comparable{

    /**
     * Compares two ASnakeAttribute objects using the fitness values
     * of the ASnakeAttribute objects.  If the object passed in is
     * not of type ASnakeAttribute, then a ClassCastException
     *  is thrown.  The result is exactly zero when the equals(Object)
     *  method would return true.  Otherwise, returns this.fitness
     *  - o.fitness.
     *
     * @param o the ASnakeAttribute
     * @return    int
     * @throws ClassCastException
     */
      public int compareTo(Object o){
           if( o instanceof ASnakeAttribute ){
                 return (int)(1000 * ( fitness -
                       ((ASnakeAttribute)o).getFitness() ));
           }else{
                 throw new ClassCastException("ClassCastException in
                             ASnakeAttribute.compareTo(o): " + o);
           }
      }

     /**
      *    Compares this ASnakeAttribute to the object passed in..
      *  Returns true if and only if the object to compare with is a
      *  ASnakeAttribute object with the same values
      *
      *    @param o - the Object to compare with
      *    @return boolean
      */
      public abstract boolean equals(Object o);
```

```
/** hashCode()
        *    returns the hashCode value of this ITabuListElement
        */
        public int hashCode(){
             long h = 1234;
             int sum = 0;
             for( int i = 0; i < values.length; i++ )
                   sum += values[i];
         h ^= sum;
             return (int)((h >> 32) ^ h);
        }

}// class ASnakeAttribute
```

```
/**
 * The ASolutionAttribute class provides a skeletal implementation
 * of the ITabuListElement and ISolutionAttribute interfaces.  This class is
 * utilized when a tabu search chooses to give solution modifications
 * tabu status, as opposed to giving entire solution tabu status.
 *
 * @author William Brown
 * @version 1.08, 09/28/2004
 */
 public abstract class ASolutionAttribute implements ISolutionAttribute{

      // values representing the attribute
      protected int[] values;
      // objective function value
      protected double fitness;
      /**
       * Given an ISolution s, applies this ASolutionAttribtue
       * object to s.  Implementation details are left to the programmer.
       * computation may be done in a class that extends ISolution or
       * in this class itself.
       *
       * @param s the ISolution
       */
      public abstract ISolution applyAttribute(ISolution s);

      /**
       *    Compares this ASolutionAttribute to the object passed in..
       *  This method may be invoked by compareTo(Object), thus
       *  correct implementation is important for the TabuList to perform
       *  correctly.
       *
       *    @param o the Object to compare with
       *    @return boolean
       */
      public abstract boolean equals(Object o);

      /**
       * Computes a hashcode for this object based on its array values.
       *
       * @return int the hashcode
       */
      public int hashCode() {
            long h = 1234;
            for (int i = values.length; --i >= 0; )
                  h ^= values[i] * (i + 1);
            return (int)((h >> 32) ^ h);
      }

      /**
       *  Returns the objective function value of this ISolution
       *
       *    @return double the fitness.
       */

      public double getFitness(){
            return fitness;
       }
```

```
	/**
	 * Returns a reference to the int[] representing this
	 * ASolutionAttribute's values.
	 *
	 * @return int[] the values
	 */
	public int[] getValues() {
		return values;
	}

	/**
	 * Returns a String representation of this ASolutionAttribute
	 * .  The format is as follows:
	 *    "<classname>      {values of the array, comma delimited}
	 *    <fitness>"
	 *    where <classname> is equivalent to the call
	 *    this.getClass().getName()
	 *
	 * @return String the object as a String
	 *
	 */
	public String toString() {
		String result = this.getClass().getName() + "\t{";
		for (int i = 0; i<values.length-1; i++) {
			result += values[i] + ",";
		}
		result += values[values.length-1] + "}\t" + fitness;
		return result;
	}

}// class ASolutionAtribute
```

```
import java.util.*;
import java.io.*;

/**
 * The Snake class provides an implementation of the ISolution
 *  interface in the tabusearch package.  A Snake is defined
 * mathematically as
 *          an induced path in a hypercube, or
 *          a path in a hypercube with no chords, or
 *          a code of spread 2.
 *
 * @author William Brown
 * @version 1.26, 2/11/2005
 */

class Driver{


      private static String outputDir;
      private static int startDim;        // smallest dimension to explore
      private static int endDim;          // largest dimension to explore
      private static int listSize;        // size of solution pool list
      private static int ilsIteration;    // number of ILS iterations

      public static void main( String args[] ){

            ilsIteration = Integer.parseInt( args[0] );
            listSize = Integer.parseInt( args[1] );
            startDim = Integer.parseInt( args[2] );
            endDim = Integer.parseInt( args[3] );
            Util.gen = new Random( Integer.parseInt( args[4] ) );

            // Create solution pools
            SnakeList[] seeds = new SnakeList[endDim-startDim+1];
            for (int k = 0; k < seeds.length; k++) {
            seeds[k] = new SnakeList( listSize );
          }

          StopWatch sw = new StopWatch();
            int best8 = 0;
            int best9 = 0;

          // for set amount of time
          for( int iteration = 1; sw.elapsedTime() < 1800000; iteration++ ){

                  // create initial solution
                  int[] v = {0,1,2};
                  Snake s = new Snake( v,startDim );

                  // for each dimension
                  for( int k = 0; k < endDim-startDim+1; k++ ){

                        // perform ILS search
                        Snake s2 = getSnake( s );
                        if( k != 0 ){
                        // update the solution pool based on the solution's
                        // performance
```

```
                        seeds[k-1].update( s, s2.getSnakeLength() );
                        seeds[k-1].sort();
                  }
                  // add the solution to the solution pool
                  seeds[k].add( s2.reduce() );

                  // need index in range [0,seeds[i].size()-1]
                  int index = Util.gen.nextInt( seeds[k].size()+1)+1;
                  // currently in range [1,seeds[i].size()+1]
                  index = Util.gen.nextInt( index );
                  // currently in range [0,seeds[i].size()]
                  index = seeds[k].size() - index;
                  // currently in range [0,seeds[i].size()]
                  if( index != 0 ){
                        index--;
                  }

                  // print results as they improve in dimensions 8 & 9
                  s = new Snake( seeds[k].get(index).getValues(),
                        k+startDim+1 );
                  if( k == 2 && seeds[2].getBest().getSnakeLength() >
                        best8 ){
                        best8 = seeds[2].getBest().getSnakeLength();
                        System.out.println (iteration + "\t" +
                              sw.elapsedTime());
                        System.out.println (seeds[2].getBest());
                  }else if( k == 3 &&
                        seeds[3].getBest().getSnakeLength() > best9 ){
                        best9 = seeds[3].getBest().getSnakeLength();
                        System.out.println (iteration + "\t" +
                              sw.elapsedTime());
                        System.out.println (seeds[3].getBest());
                  }
            }// for

      }// for

}

/* Performs an ILS on the Snake object passed in for ilsIteration
   iterations */
private static Snake getSnake( Snake sn ){
      ISolution sCurrent, s0 = sn;

      // Create ILS instance
      SnakeILS s = new SnakeILS( sn.getDimension() );

      s.startingLength = sn.getSnakeLength();

      // generate starting solution
      sCurrent = s.localSearch( s0 );
      for (int i = 0; i < ilsIteration; i++) {
            // perturb
            ISolution sPerturb = s.perturb( sCurrent );
            // local search
            ISolution sTemp = s.localSearch(sPerturb);
            // acceptance criteria
```

```
                if( s.accept( sTemp ) ){
                      sCurrent = sTemp;
                }
            }
            return (Snake)(s.getBest());
      }

}// class Driver
```

```
import java.io.*;

/**
 * The Snake class provides an implementation of the ISolution
 *  interface in the tabusearch package.  A Snake is defined
 *  mathematically as
 *          an induced path in a hypercube, or
 *          a path in a hypercube with no chords, or
 *          a code of spread 2.
 *
 * @author William Brown
 * @version 1.10, 2/17/2005
 */

class ILSOnly{

      public static void main( String args[] ){

            try{
                  FileWriter fw = new FileWriter( new File("ILSOnly_" +
                        args[0] + ".out") );
                  StopWatch sw = new StopWatch();
                  int dimension = Integer.parseInt( args[0] );

                  // Initial Solution
                  int[] vals = {0,1,2};
                  Snake s0 = new Snake( vals, dimension );

                  // Best values known for dimensions 6,7,8,9
                  int[] bestVals = {26,50,97,186};

                  ISolution sCurrent;
                  Snake sn = s0;

                  SnakeILS s = new SnakeILS( sn.getDimension() );
                  s.startingLength = sn.getSnakeLength();

                  // Generate Initial Solution
                  sCurrent = s.localSearch( s0 );
                  int counter = 0;

                  for (;sw.elapsedTime() < 1000 * 60 * 30 &&
                        s.getBest().getFitness() < bestVals[dimension-6];) {
                        counter++;

                        // Perturb Solution
                        ISolution sPerturb = s.perturb( sCurrent );

                        // Perform Local Search
                        ISolution sTemp = s.localSearch(sPerturb);

                        // Check with Acceptance Criteria
                        if( s.accept( sTemp ) ){
                              sCurrent = sTemp;
                        }
                        if( counter % 1000 == 0 )
```

```
                        fw.write ( counter + "\t" + sw.elapsedTime() +
                        "\t" + s.getBest() + "\t" + sCurrent + "\n" );
                    if( counter % 10000 == 0 )
                        System.out.println (s.getBest());
                }
                fw.write ("\n");
                fw.write ( counter + "\t" + sw.elapsedTime() + "\t" +
                  s.getBest() );
                fw.flush();
                System.out.println (s.getBest());
            }catch( Exception e ){
              e.printStackTrace();
            }// catch

        }// main()
}// ILSOnly
```

```
import java.util.Iterator;

/**
 *  This interface defines the behavior necessary to implement a
 *  neighborhood around
 *  an ISolution object.  Note that this interface does NOT provide any
 *  methods for constructing the neighborhood.  Instead, the behaviour is
 * defined in such a way that elements may be added, removed, and retrieved
 * in a set-like (no repeats) fashion.
 *
 *  @author William Brown
 *  @version 1.01, 08/28/2004
 */
public interface INeighborhood{

      /**
       *  Adds the Object to this INeighborhood.
       *
       *  @param o the Object to add
       */
       public void add( Object o );

      /**
       * Returns the last (largest) element in the set.  The largest element
       * is the element that satisfies the following statement for all other
       * elements in the INeighborhood:
       *          largestElement.compareTo(anyOtherElement) > 0
       *
       * @return Object the largest element
       */
      // public Object best();

      /**
       * Removes all ISolution elements from this Neighborhood.
       */
       public void clear();

      /**
       *    Returns true if and only if this INeighborhood
       *  contains no elements.
       *
       *    @return boolean
       */
       public boolean isEmpty();

      /**
       * Returns a java.util.Iterator object over the elements of the
       * Neighborhood, preserving their natural ordering.
       *
       * @return Iterator over the set
       */
       public Iterator iterator();

      /**
       *  Removes the Object from this Neighborhood.
       *
       *  @param o the Object to remove
```

```
         */
         public void remove( Object o );

        /**
         * Returns the number of elements in this Neighborhood
         *
         * @return int the number of elements present in this Neighborhood.
         */
         public int size();

        /**
         * Returns a String representation of this Neighborhood
         *
         * @return String
         */
         public String toString();

}// interface INeighborhood
```

```
/**
 * This interface defines the methods necessary for an object to be
 * considered a Solution in a TabuSearchTabuSearch implementation.
 * Classes implementing ISolution must also implement the Comparable
 * interface.  Otherwise, there can be no ordering at the time that a
 * Neighborhood is created.
 *
 * @author William Brown
 * @version 1.01, 08/28/2004
 */
public interface ISolution extends Comparable{

      /**
     * Returns true if and only if the object passed in is another ISolution
     * and is equal to this ISolution.  This method is utilized in the
     * ITabuList to determine ISolution membership in the
     * tabu list.  It may also be used inside compareTo(Object) to determine
     * the 0 case.
     *
     * @param     o the Object to compare with
     * @return    boolean true if equal, false otherwise
     */
     public boolean equals(Object o);

      /**
       *  Returns the objective function value of this ISolution
       *
       *    @return double the objective funciton value
       */
       public double getFitness();

      /**
       *    Returns the values of this ISolution
       *
       *    @return int[] the values that define this ISolution
       */
       public int[] getValues();

      /**
       * Computes a hashcode for this ISolution.  Utilized by the
       * TabuList class.
       *
       * @return int the hashcode
       */
       public int hashCode();

      /**
       *    Returns a String representation of this ISolution
       *
       *    @return String
       */
       public String toString();

}// interface ISolution
```

```
/**
 * This interface defines the methods necessary to Implement an
 * ISolutionAttribute
 * An ISolutionAttribute is an Attribute of an ISolution.  How this
 * statement is interpreted is dependent upon the specific implementation.
 *
 * @author William Brown
 * @version 1.08, 09/28/2004
 */
public interface ISolutionAttribute {

      /**
       * Given an ISolution s, applies this ASolutionAttribtue
       * object to s.  Implementation details are left to the programmer.
       * Actual computation may be done in a class that extends ISolution or
       * in this class itself.
       *
       * @param s the ISolution
       */
      public ISolution applyAttribute(ISolution s);

      /**
       * Returns a String representation of this ASolutionAttribute
       * .  The format is as follows:
       *    "<classname>      {values of the array, comma delimited}
       *    <fitness>"
       *    where <classname> is equivalent to the call
       *    this.getClass().getName()
       *
       * @return String the object as a String
       */
      public String toString();

      /**
       * Returns a reference to the int[] representing this
       * ASolutionAttribute's values.
       *
       * @return int[] the values
       */
      public int[] getValues();

      /**
       *    Compares this ASolutionAttribute to the object passed in..
       *  This method may be invoked by compareTo(Object), thus
       *  correct implementation is important for the TabuList to perform
       *  correctly.
       *
       *    @param o the Object to compare with
       *    @return boolean
       */
      public boolean equals(Object o);

      /**
       * Computes a hashcode for this object based on its array values.
       *
       * @return int the hashcode
       */
```

```
        public int hashCode();

        /**
         *  Returns the objective function value of this ISolution
         *
         *    @return double
         */
        public double getFitness();

}// ISolutionAttribute
```

```
/**
 * This interface defines the methods necessary for an Iterated Local Search
 * procedure.
 * The IteratedLocalSearch algorithm acts as follows:
 *
 *    For a set number of iterations, or until termination condition is met:
 *          1. Generate a starting solution, s.
 *          2. Change s in some way, resulting in s*
 *          3. local search on s* until a local optimum is reached, s*'
 *          4. Make sure that s* meets acceptance criteria
 *          5. If s*' meets the criteria, set s = s*'
 *          6. Repeat from 2
 *
 *
 * @author William Brown
 * @version 1.0, 12/28/2004
 */
public interface IteratedLocalSearch{

      /**
     * Returns an ISolution to be used as the starting solution in the
     * Iterated Local Search
     * @return ISolution
     */
      public ISolution generateInitialSolution();

      /**
     * Returns an ISolution that is a local optima in the search.  This local
     * search procedure begins with starting ISolution, s, and improves upon
     * it until a the local optima is reached.
     * @param ISolution the starting solution
     * @return ISolution the local optimum
     */
      public ISolution localSearch( ISolution s );

      /**
     * Returns an ISolution that is a perturbation of the solution s.
     * @param ISolution the solution to perturb
     * @return ISolution the resulting solution after perturbation
     */
      public ISolution perturb( ISolution s );

      /**
     * Returns true if the ISolution passed in meets certain criteria.
     * The ISolution is typically then used in the method localSearch().
     * @param ISolution the ISolution to check against the
     *       criteria
     * @return true if the ISolution passes
     */
      public boolean accept( ISolution s );

}// class IteratedLocalSearch
```

```
import java.util.Iterator;
import java.util.Set;
import java.util.TreeSet;
import java.util.SortedSet;

/**
 * This class implements the INeighborhood interface.  It represents the
 * neighborhood to a solution in a Local Search.  Note that this class does
 * NOT provide any methods for constructing the neighborhood.  This
 * particular implementation uses a
 * java.util.SortedSet Tree to keep the elements so that an
 * ordering over the elements of the neighborhood is maintained.  This
 * ordering is dependent on the compareTo(Object) method defined in the class
 *  implementing ISolution.
 *
 *  @author William Brown
 *  @version 1.01, 08/28/2004
 */
public class Neighborhood implements INeighborhood{

       // The set of neighboring elements
       private SortedSet neighbors;

       /**
        * Constructs a new Neighborhood that acts as a set to hold
        * neighboring elements of a Solution.  Uses a java.util
        * .TreeSet object to hold the elements.
        */
       public Neighborhood( ){
            this( new TreeSet() );
       }

       /**
        * Constructs a new Neighborhood that acts as a set to hold
        * neighboring elements of a Solution.  Uses the java.util
        * SortedSet object to hold the elements.
        */
       public Neighborhood( SortedSet s ){
            neighbors = s;
       }

      /**
       *  Adds the Object to this Neighborhood.
       *
       *  @param o the Object to add
       */
       public void add( Object o ){
            neighbors.add( o );
       }

      /**
       * Returns the last (largest) element in the set.  The largest element
       * is the element that satisfies the following statement for all other
       * elements in the Neighborhood:
       *          largestElement.compareTo(anyOtherElement) > 0
       *
       * @return Object the largest element
```

```
 */
 public Object best(){
      return ((SortedSet)neighbors).last();
 }

/**
 * Removes all ISolution elements from this Neighborhood.
 */
 public void clear(){
      neighbors.clear();
 }

/**
 *    Returns true if and only if this Neighborhood
 *  contains no elements.
 *
 *    @return boolean
 */
 public boolean isEmpty(){
      return neighbors.isEmpty();
 }

/**
 * Returns a java.util.Iterator object over the elements of the
 * Neighborhood, preserving their natural ordering.
 *
 * @return Iterator over the set
 */
 public Iterator iterator(){
      return neighbors.iterator();
 }

/**
 *  Removes the Object from this Neighborhood.
 *
 *  @param o the Object to remove
 */
 public void remove( Object o ){
      neighbors.remove(o);
 }

/**
 * Returns the number of elements in this Neighborhood
 *
 * @return int the number of elements present in this Neighborhood.
 */
 public int size(){
      return neighbors.size();
 }

/**
 * Returns a String representation of this Neighborhood
 *
 * @return String
 */
 public String toString(){
      String result = "";
```

```
            if( !neighbors.isEmpty() ){
                  for( Iterator i = neighbors.iterator(); i.hasNext(); ){
                        result += i.next().toString() + "\n";
                  }
            }
            return result;
      }
}// class Neighborhood
```

```
/**
 * The NeighborhoodBuilder class provides an implementation of
 * the generateNeighborhood method.  This provides a way for an
 * INeighborhood object to be filled with ISolutionAttributes.
 *
 * @author William Brown
 * @version 1.12, 11/01/2004
 */
public class NeighborhoodBuilder {

      /**
       * Constructs an INeighborhood of ASolutionAttributes.
       * This is performed by constructing a neighborhood of SnakeEnds and
       * SnakeSwap elements.
       * @param s - the ISolution to construct the neighborhood around
       * @param n - the INeighborhood to fill
       */
      public static void generateNeighborhood( ISolution s,
                                    INeighborhood n ){

            Snake sn = (Snake)s;
            int dimension = sn.getDimension();

            // SnakeEnds
            for (int i = 0; i < dimension; i++) {
                  int[] ea = new int[2];
                  ea[0] = sn.getSnakeLength();
                  ea[1] = i;
                  // see if it 'makes sense' to add this SnakeEnd
            if( sn.tryAddEnd( ea[1] ) ){
                  ISolutionAttribute end = new SnakeEnd(ea);
                  // add it to the INeighborhood
                  ((SnakeEnd)end).setFitness( Util.gen.nextDouble() );
                  n.add( end );
                }
           }

      }// generateNeighborhood

}// class NeighborhoodBuilder
```

```
import java.util.Arrays;

/**
 * The Snake class provides an implementation of the ISolution
 *  interface in the tabusearch package.  A Snake is defined
 * mathematically as
 *          an induced path in a hypercube, or
 *          a path in a hypercube with no chords, or
 *          a code of spread 2.
 *
 * @author William Brown
 * @version 1.26, 2/11/2005
 */
 final class Snake extends Solution{

      // Fields

      // the dimension of the hypercube this snake is defined on
      private int dimension;
      // array representing the path as a series of movements
      private int[] transitionSequence;
      // array representing the path as a series of nodes
      private int[] nodeSequence;
      // adjacency graph of the hypercube with path information
      private InducedNode[] inducedGraph;
      // current length of the snake
      private int snakeLength;

      // protected double fitness
      // protected int[] values

   /**
    * Constructs a new Snake with the given transition sequence array
    * on a hypercube of the passed in dimension.
    */
      public Snake( int[] v, int d ){
            super(v);
            transitionSequence = values;
            dimension = d;

            // Reduces the Snake
            int[] map = new int[dimension];
            for( int i = 0; i < map.length; i++ )
                  map[i] = -1;
            int current = 0;
            for( int i = 0; i < values.length; i++ ){
                  int x = values[i];
                  if ( map[ values[i] ] == -1 ){
                        map[x] = current;
                        current++;
                  }
            values[i] = map[ values[i] ];
            }
            // Reduces the Snake

            // create node sequence
            nodeSequence = new int[ values.length ];
```

```
        constructNodeSequence();
        // create the induced subset of the hypercube
        inducedGraph = new InducedNode[ 1 << dimension ];
        for (int i = 0; i < inducedGraph.length; i++)
              inducedGraph[i] = new InducedNode();
        constructInducedGraph();
        snakeLength = values.length;
        fitness = snakeLength;
  }

/**
 * Constructs a new Snake with the given data for use when all
 * instance data is computed outside the class.  Utilized by the clone()
 *  method.
 */
  public Snake( int[] v, int[] ns, int d, double f ){
        super(v);
        this.transitionSequence = values;
        this.nodeSequence             = ns;
        this.dimension                = d;
        this.inducedGraph             = new InducedNode[1<<d];
        for (int i = 0; i < inducedGraph.length; i++)
              inducedGraph[i] = new InducedNode();
        this.constructInducedGraph();
        this.fitness                  = f;
        this.snakeLength        = (int)fitness;
  }

  /**
   * Creates a deep copy of this Snake object.
   *
   * @return the copy of this Snake
   */
  public Object clone(){
        int[] v = new int[values.length];
        System.arraycopy(values,0,v,0,values.length);
        int[] ns = new int[ nodeSequence.length ];
        System.arraycopy(nodeSequence,0,ns,0,ns.length);
        return new Snake( v, ns, dimension, fitness );
  }

  /**
 * Reduces this snake to one that is in its isomorphism class.  It does
 * so by replacing the sequence with another one in which the first
 * appearance of any transition node is not preceded by another node of
 * greater value.
 *  For example, if the starting transition sequence is:
 *            {2, 1, 3, 2, 0}
 *  The resulting sequence will be:
 *            {0, 1, 2, 0, 3}
 *
 * @return a new Snake with the reduction performed on its values
 */
  public Snake reduce(){
        int[] map = new int[dimension];
        for( int i = 0; i < map.length; i++ )
              map[i] = -1;
```

```
      int current = 0;
      for( int i = 0; i < values.length; i++ ){
            int x = values[i];
            if ( map[ values[i] ] == -1 ){
                  map[x] = current;
                  current++;
            }
      values[i] = map[ values[i] ];
      }
      return new Snake( values, dimension );
}

/**
* Returns a Snake with the path reversed
*
* @return the reversed Snake
*/
public Snake reverse(){
      int[] rev = new int[ values.length ];
      System.arraycopy(values,0,rev,0,values.length);
      for (int i = 0; i < rev.length/2; i++) {
            int temp  = rev[i];
            rev[i] = rev[ rev.length-1-i ];
            rev[ rev.length-1-i ] = temp;
      }
      Snake s = new Snake( rev, dimension );
      return s;
}

/* Constructs the node sequence array representation */
private void constructNodeSequence(){
      nodeSequence = new int[ transitionSequence.length+1 ];
      nodeSequence[0] = 0;
      for( int i = 1; i < nodeSequence.length; i++ )
            nodeSequence[i] = nodeSequence[i-
                  1]^(1<<transitionSequence[i-1]);
}

/* Constructs a graph of InducedNodes representing the hypercube
   Each node remembers if it was a part of the path AND
   the identity of all other nodes adjacent to it that ARE a part of
   the path. This information can be utilized when determining the
   fitness value of any modification to the Snake */
private void constructInducedGraph(){
      for (int i = 0; i < inducedGraph.length; i++) {
            inducedGraph[i].inPath = false;
            inducedGraph[i].position = -1;
    }

      // population
      for( int i = 0; i < nodeSequence.length; i++ ){
            inducedGraph[ nodeSequence[i] ].inPath = true;
            inducedGraph[ nodeSequence[i] ].position = i;
            for( int j = 0; j < dimension; j++ )
               inducedGraph[nodeSequence[i]^(1<<j)].values[j] = true;
      }
}
```

```
/**
 * @author William Brown
 * @version 1.00, 09/24/2004
 *
 * Class that represents a node in the induced hypercube subgraph,
 * inducedGraph defined in class Snake.  Stores information regarding
 *            which adjacent nodes (if any) are in the path --> int[]
 *            whether the node is in the path --> boolean
 *            which number node is the node --> int
 */
  class InducedNode{
        public boolean[] values = new boolean[ dimension ];
        public boolean inPath = false;
        public int position = -1;
  }

  /**
   * Returns the fitness of this Snake after it has been extended
   * by one node.  If the extension causes the snake to violate its
   * properties, it returns 0;
   *
   * @param int the node to add
   * @return double
   */
  public boolean tryAddEnd( int end ){
        // the transition to add, converted to a node
        int endNode = nodeSequence[ nodeSequence.length-1 ]^(1<<end);
        // for each dimension
        for (int i = 0; i < dimension; i++) {
              // check to see if surrounding nodes are occupied
              if( i != end && inducedGraph[ endNode ].values[i] ){
                    return false;
              }
      }
        return true;
  }

  /**
 * Appends a transition to the end of the transition sequence in this
 * snake.
 *
 * @param int - the next "step" to take in the transition sequence
 */
  public void addEnd( int end ){
        // extend the transitionSequence array by one
        int[] tranSeq = new int[ transitionSequence.length+1 ];
        System.arraycopy( transitionSequence, 0, tranSeq, 0,
                                        transitionSequence.length );
        transitionSequence = tranSeq;
        // add the new node
        transitionSequence[ transitionSequence.length-1 ] = end;
        // update values array
        values = transitionSequence;

        // extend the nodeSequence array by one
        int[] nodeSeq = new int[ nodeSequence.length+1 ];
```

```
        System.arraycopy( nodeSequence, 0, nodeSeq, 0,
                nodeSequence.length );
        nodeSequence = nodeSeq;
        int endNode = nodeSequence[ nodeSequence.length-2 ]^(1<<end);
        // add the new node
        nodeSequence[ nodeSequence.length-1 ] = endNode;

        // update the induced graph
        inducedGraph[ endNode ].inPath = true;
        inducedGraph[ endNode ].position = snakeLength+1;
        // for each neighboring node, leave endNode's 'mark'
        for( int i = 0; i < dimension; i++ )
                inducedGraph[ endNode^(1<<i) ].values[i] = true;

        // update fitness & snake length
        fitness += 1;

        snakeLength += 1;
        //evalFitness();
}

/**
 * Returns the dimension of the hypercube that this Snake is defined
 * on.
 *
 * @return int the dimension of the hypercube
 */
public int getDimension() {
        return dimension;
}

/**
 * Returns the length of the Snake formed by this Snake object.
 *
 * @return int the length of the snake
 */
public int getSnakeLength() {
        return snakeLength;
}

/**
 * Returns a double representing the objective function value of this
 * Snake object.
 *
 * @return double the objective function value
 */
public double getFitness() {
        return snakeLength;
}

/**
 * Subtracts a node from the end of the node sequence of this snake.
 * The transition sequence is also updated, as well as the induced
 * graph.  The net result of the operation is that
 * getSnakeLength() will be 1 smaller after this method has executed.
 */
public void subtractEnd(){
```

```
            int endNode = nodeSequence[ nodeSequence.length-1 ];

            //update transition sequence
            int[] tranSeq = new int[ transitionSequence.length-1 ];
            System.arraycopy( transitionSequence, 0, tranSeq, 0,
                  tranSeq.length );
            transitionSequence = tranSeq;
            values = transitionSequence;

            //update node sequence
            int[] nodeSeq = new int[ nodeSequence.length-1 ];
            System.arraycopy( nodeSequence, 0, nodeSeq, 0, nodeSeq.length );
            nodeSequence = nodeSeq;

            //update induced graph
            inducedGraph[ endNode ].inPath = false;
            inducedGraph[ endNode ].position = -1;
            // for each neighboring node, leave endNode's 'mark'
            for( int i = 0; i < dimension; i++ ){
                  inducedGraph[ endNode^(1<<i) ].values[i] = false;
            }
            // update fitness & snakeLength
            fitness -= 1;
            snakeLength -= 1;

      }

}// class Snake
```

```
import java.util.Arrays;

/**
 * The SnakeEnd class represents an additional node that can be
 * appended to a Snake (an induced path in a hypercube).
 *
 * @author William Brown
 * @version 1.02, 10/29/2004
 */
public final class SnakeEnd extends ASnakeAttribute{

      /**
     * Constructs a new SnakeEnd with the given array, the given
     * Snake object and the new fitness value.
     */
       public SnakeEnd( int[] vals ){
            values = new int[vals.length];
            System.arraycopy(vals,0,this.values,0,vals.length);
       }

      /**
       * Given a Snake s, applies this SnakeEnd object
       * to s.
       * WARNING:  It should be noted that this application is performed
       * "blindly".
       * That is, this method should not be invoked on a Snake unless
       * the method tryAddEnd in class Snake has returned
       * a value greater that the fitness of that Snake.
       *
       * @param s - the ISolution
       * @throws ClassCastException if s is not of type Snake
       */
       public ISolution applyAttribute( ISolution s ){
            if( s instanceof Snake ){
                  ((Snake)s).addEnd( values[1] );
                  return s;
            }else
                  throw new ClassCastException();
       }

      /**
     * Compares two SnakeEnd objects using the fitness values of
     * the SnakeEnd objects.  If the object passed in is not of type
     * SnakeEnd, then a call to super.compareTo(Object)
     * is made.  The result is exactly zero when the equals(Object)
     * method would return true.  Otherwise, returns a negative if this
     * SnakeEnd object is "less than" the other SnakeEnd
     * object and a positive int otherwise.
     *
     * @param o the SnakeEnd
     * @return    int
     */
       public int compareTo( Object o ){
            if( o instanceof SnakeEnd ){
                  if( this.equals( o ) )
                        return 0;
                  // multiply by factor b/c fitnesses differ by less than 1
```

```
                return (int)( 10000 * (fitness -
                      ((SnakeEnd)o).getFitness()) );
          }else
                return super.compareTo(o);
     }

     /**
      *    Compares this SnakeEnd to the object passed in..
      *  Returns true if and only if the object to compare with is a
      *  SnakeEnd object with the same values  (the same node is being
      *  replaced with the same node number).
      *
      *    @param o - the Object to compare with
      *    @return boolean
      */
     public boolean equals( Object o ){
          if( o instanceof SnakeEnd ){
                // compare the values of the two SnakeEnds
                int[] other = ((SnakeEnd)o).getValues();
                return ( other[0] == values[0] && other[1] == values[1] );
          }else
                return false;
     }

     /**
     * Returns an int indicating the transition sequence node that this
     * SnakeEnd represents.
     *
     * @return    int
     */
      public int getEndValue(){
          return values[1];
     }

     /**
      * Returns an int representing the position that this SnakeEnd
      * takes in the Snake object.
      * @return   int
      */
      public int getPosition(){
          return values[0];
      }

     /**
      * Sets the fitness of this SnakeEnd to the double passed
      * in.
      * @param double
      */
      public void setFitness( double f ){
          fitness = f;
      }
}// class SnakeEnd
```

```
import java.util.Iterator;

/**
 * This class defines an Iterated Local Search procedure for finding maximal
 * snakes in a hypercube.
 * The IteratedLocalSearch algorithm acts as follows:
 *
 * @author William Brown
 * @version 1.0, 12/28/2004
 */

public class SnakeILS implements IteratedLocalSearch{

      // FIELDS

      private int dimension;  // The dimension of the hypercube to search in
      private Snake best;           // The best solution found so far
      public int startingLength;

      /**
     * Constructs a new SnakeSearch object set to find Snakes in dimension d
     * @param int the dimension
     */
      public SnakeILS( int d ){
            dimension = d;
            best = (Snake)generateInitialSolution();
      }

      /**
     * Returns an ISolution to be used as the starting solution in the
     * Iterated Local Search
     * @return ISolution
     */
      public ISolution generateInitialSolution(){
            int[] values = {0,1,2};
            Snake s = new Snake( values, dimension );
            return s;
      }

      /**
     * Returns an ISolution that is a local optima in the search.  This local
     * search procedure begins with starting ISolution, s, and improves upon
     * it until a the local optima is reached.
     * @param ISolution the starting solution
     * @return ISolution the local optimum
     */
      public ISolution localSearch( ISolution s ){
            boolean canExtend = true;
            // create a copy of our Solution
            Snake copy = (Snake)(((Snake)s).clone());
            int counter = 0;

            // Perform the local search here
            // add nodes to the end of the Solution until it is maximal
            while( canExtend ){
                  counter++;
                  // create neighborhood of possible nodes to append
```

```
            Neighborhood neighbors = new Neighborhood();
            NeighborhoodBuilder.generateNeighborhood(copy, neighbors);
            if( neighbors.isEmpty() ){
                  // exit if there are no viable neighboring nodes
                  canExtend = false;
            }else{
                  // otherwise, add one to the end

                  Object o = neighbors.best();
                  copy.addEnd( ( (SnakeEnd)o ).getEndValue() );
            }// if-else
      }// while

      //update the best solution found if necessary
      if( copy.getFitness() >= best.getFitness() )
            best = (Snake)(((Snake)copy).clone());

      // return the extended copy
      return copy;
 }

 /**
* Returns an ISolution that is a perturbation of the solution s.
* @param ISolution the solution to perturb
* @return ISolution the resulting solution after perturbation
*/
 public ISolution perturb( ISolution s ){

      // create a copy
      Snake copy = (Snake)(((Snake)s).clone());

      // choose number of nodes to remove from linear probability
      // distribution
      int removeNumber =   Util.gen.nextInt( Util.gen.nextInt(
            copy.getFitness() - (startingLength-5) + 1) + 1 ) + 3;

      // remove aforementioned number of nodes
      for (int i = 0; i < removeNumber && copy.getFitness() >
            startingLength; i++) {
      copy.subtractEnd();
    }

    return copy;
 }

 /**
* Returns true if the ISolution passed in meets certain criteria.
* The ISolution is typically then used in the method localSearch().
* @param ISolution the ISolution to check against the
*      criteria
* @return true if the ISolution passes
*/
 public boolean accept( ISolution s ){

      // Accept if the solution matches the best within 1
      if( s.getFitness() > best.getFitness()-1 ){
            if( s.getFitness() >= best.getFitness() ){
```

```
                        best = (Snake)(((Snake)s).clone());
                  }
                  return true;
            }else{
                  return false;
            }
      }

      /**
     * Returns the best ISolution found in this instance of SnakeSearch
     * @return ISolution the best solution found so far
     */
      public ISolution getBest(){
            return best;
      }

      /**
       * Returns a String representation of this SnakeSearchObject
       * @return String
       */
      public String toString() {

            String sep = System.getProperty("line.separator");

            StringBuffer buffer = new StringBuffer();
            buffer.append(sep);
            buffer.append("dimension = ");
            buffer.append(dimension);
            buffer.append(sep);

            buffer.append("best = ");
            buffer.append(best);
            buffer.append(sep);

            return buffer.toString();
      }

}// class SnakeILS
```

```
import java.util.*;

/**
 * The SnakeList class provides an implementation of a list of
 * Snake objects.  The list is kept in ascending order based on
 * the result of the getSnakeLength() method from class Snake.
 * Snake objects with the same
 * @author William Brown
 * @version 1.26, 2/11/2005
 */

class SnakeList{

      private int maxSize;
      private ArrayList<SnakeListNode> list;
      private Snake best;

      // Constructors
      public SnakeList(){
            list = new ArrayList();
            maxSize = Integer.MAX_VALUE;
      }

      public SnakeList( int size ){
            list = new ArrayList();
            maxSize = size;
      }

      /**
     * Returns true if a Snake was successfully added to this SnakeList
     * @param Snake
     * @return boolean
     */
      public boolean add( Snake s ){

            for (int i = 0; i < list.size(); i++) {
            if( get(i).equals(s) ){
                  return false;
            }
          }

            if( list.size() == maxSize ){
                  SnakeListNode min = list.get(0);

                  for (int i = 1; i < list.size(); i++) {
                  if( list.get(i).best > 0 && list.get(i).best < min.best &&
                  list.get(i).snake.getSnakeLength() <
                        min.snake.getSnakeLength() ){
                        min = list.get(i);
                  }
                }
                list.remove(min);
            }

          int i = 0;
          while( i < list.size() && list.get(i).snake.getSnakeLength() <
                        s.getSnakeLength() )
```

```
      i++;
     list.add( i, new SnakeListNode(s) );
      return true;
 }

 /**
* Sets the maximum number of elements this SnakeList will hold
* @param int
*/
 public void setMaxSize( int ms ){
      maxSize = ms;
      while( list.size() > maxSize ){
            list.remove(0);
      }
 }

 /**
* Removes all elements in this SnakeList
*/
 public void clear(){
      list.clear();
      best = null;
 }

 /**
* Returns the maximum number of elements this SnakeList will hold.
* @return int
*/
 public int getMaxSize(){
      return maxSize;
 }

 /**
* Returns the Snake at the specified index
* @param int
* @return Snake
*/
 public Snake get( int i ){
      return (list.get(i)).snake;
 }

 /**
* Returns the Snake in this list with the best fitness
* value
* @return Snake
*/
 public Snake getBest(){
      return (Snake)(list.get( list.size()-1 ).snake);
 }

 /**
* Removes the Snake element from this SnakeList
* @param Snake
*/
 public void remove( Snake s ){
      list.remove( new SnakeListNode(s) );
 }
```

```
 /**
* Returns an int representing the number of elements currently
* in this SnakeList.
* @return    int
*/
 public int size(){
       return list.size();
 }

 /**
* Returns a double representing the average fitness
* of a Snake in this SnakeList.
* @return double
*/
 public double getAverage(){
       int sum = 0;
       for (int i = 0; i < list.size(); i++) {
       sum += get(i).getSnakeLength();
     }
     return (double)sum/list.size();
 }

 /**
* Updates ordering of the Snake in the list.
* @param Snake
* @param int
*/
 public void update( Snake s, int length ){
       for( int i = 0; i < list.size(); i++ ){
             if( list.get(i).snake.equals( s ) ){
                   if(list.get(i).best < length){
                         list.get(i).best = length;
                   }
                   list.get(i).iterations ++;
             }
       }
 }

 /**
* Returns a String representation of this SnakeList
* @return    String
*/
 public String toString2(){
       String s = list.get(list.size()-1) + "\t" + getAverage();
       return s;
 }

 /**
* Returns a String representation of this SnakeList
* @return    String
*/
 public String toString(){
       String s = "";
       for (int i = 0; i < list.size(); i++) {
       s += list.get(i).snake + " " + list.get(i).best + "\n";
     }
```

```
            return s;
        }

        /**
       * Sorts the elements of this SnakeList.
       */
        public void sort(){
              SnakeListNode[] sln = new SnakeListNode[list.size()];
              list.toArray(sln);
              Arrays.sort(sln);
              list.clear();
              for (int i = 0; i < sln.length; i++) {
              list.add( sln[i] );
            }
        }

        /**
       * Encapsulates the Snakes added to this list
       */
        class SnakeListNode implements Comparable{
              public Snake snake;
              public int iterations;
              public int best;
              public SnakeListNode( Snake s ){
                    snake = s;
                    iterations = 0;
                    best = 0;
              }

              public int compareTo( Object o ){
                    SnakeListNode sln = (SnakeListNode)o;
                    if( snake.compareTo(sln.snake) == 0 )
                          return (int)(this.best-sln.best);
                    else
                          return snake.compareTo(sln.snake);
              }

              public boolean equals( Object o ){
                    return snake.equals( ((SnakeListNode)o).snake );
              }

              public String toString(){
                    return snake.toString() + "\t" + best + "\t" + iterations;
              }

        }

}// class SnakeList
```

```
import java.util.Arrays;

/**
 * The ASolution class is a skeletal implementation of the
 * ISolution and ITabuListElement interfaces in package
 * tabusearch.  More specifically, it gives a simple implementation
 * for a solution that would be used in a tabu search setting where entire
 * solutions are kept in the tabu list.
 *
 * @author William Brown
 * @version 1.03, 09/09/2004
 */
 public class Solution implements ISolution{

      // values representing this solution
      protected int[] values;
      // objective function value
      protected double fitness;

      // CONSTRUCTOR
      public Solution( int[] values ){
            this.values = new int[values.length];
            System.arraycopy(values,0,this.values,0,values.length);
      }

      /**
     * Compares two Solution objects using the fitness values of
     * the Solution objects.  If the object passed in is not of type
     * Solution, then a ClassCastException is thrown.
     * The result is exactly zero when the equals(Object)
     * method would return true.  Otherwise, returns a negative if this
     * Solution object is "less than" the other Solution
     * object and a positive int otherwise.
     *
     * This method is utilized by the Neighborhood class in order to
     * keep an ordered set of Solutions.
     *
     * @param o the Object with which to compare
     * @return int comparison result
     * @throws ClassCastException
     */
      public int compareTo(Object o) {
            if( o instanceof Solution ){
                  // if 2 objects are equal, return 0
                  if( this.equals(o) )
                        return 0;
                  // otherwise, return the difference in fitness values
                  int x = (int)(1000*(fitness-((Solution)o).getFitness() ));
                  return x;
            }else{
                  throw new ClassCastException();
            }
      }

      /**
       *    Compares this Solution to the object passed in..
       *  Returns true if and only if the object to compare with is a
```

```
 *  Solution object with the same values  in the 'values' array.
 *  That is, Arrays.equals(this.values,o.values) == true.
 *  For specific implementations, when different criteria are used to
 *  destinguish two solutions, this can be overridden.
 *
 *    @param o the Object to compare with
 *    @return boolean
 *  @throws ClassCastException
 */
public boolean equals(Object o){
      if( o instanceof Solution )
            return Arrays.equals( values, ((Solution)o).getValues() );
      else
            throw new ClassCastException();
}

/**
 * Returns a double representing the objective function value, or
 * 'fitness' of this Solution.
 *
 * @return double the fitness
 */
public double getFitness() {
      return fitness;
}

/**
 * Returns a new, deep copy of this Solution
 *
 * @return Solution
 */
public Solution getSolution(){
      return new Solution( values );
}

/**
 * Returns a reference to the int[] representing this Solution's
 * values.
 *
 * @return int[] the values that define this Solution
 */
public int[] getValues() {
      return values;
}

/**
 * Computes a hashcode for this object based on its array values.
 *
 * @return int the hashcode
 */
public int hashCode() {
      long h = 1234;
      for (int i = values.length; --i >= 0; )
      h ^= values[i] * (i + 1);
      return (int)((h >> 32) ^ h);
}
```

```
	/**
	 * Returns a String representation of this Solution.
	 * The format is as follows:
	 *    "<classname>      {values of the array, comma delimited}
	 * <fitness>"
	 *    where <classname> is equivalent to the call
	 *  this.getClass().getName()
	 *  .
	 *
	 * @return String the object as a String
	 *
	 */
	public String toString() {
		String result = this.getClass().getName() + "\t{";
		for (int i = 0; i<values.length-1; i++) {
			result += values[i];// + ",";
		}
		result += values[values.length-1] + "}\t" + fitness;
		return result;
	}

}// class Solution
```

```
import java.util.Random;

/**
 *    Provides various utilities.  i.e. --> A static Random Generator for all
classes to share
 */
public final class Util{

      public static Random gen = new Random();

}// class Util
```