An Ant Colony approach to the Snake-In-the-Box Problem

by

Shilpa Prakash Hardas

(Under the direction of Walter D. Potter)

Abstract

In this thesis we present a new approach to the Snake-In-the-Box (SIB) problem using Ant Colony Optimization (ACO). ACO refers to a class of algorithms that model the foraging behavior of ants to find solutions to combinatorial optimization problems. SIB is a well-known problem, that involves finding a Hamiltonian path through a hypercube which follows certain additional constraints. This domain suffers from severe combinatorial explosion and has been the subject of various search techniques which include genetic algorithms [Potter et al., 1994; Casella, 2005], exhaustive search [Kochut, 1994], mathematical logic [Rajan and Shende, 1999] and iterated local search [Brown, 2005]. After making certain problem specific customizations a variation on the MIN-MAX Ant System, MMAS_SIB, has shown very promising results when applied to the SIB problem. The length of the longest known snake in dimension 8 was matched, using much less computation and time than the best known methods for this problem.

Index words: Ant Colony Optimization, Snake-In-the-Box, Swarm intelligence

An Ant Colony approach to the Snake-In-the-Box Problem

by

Shilpa Prakash Hardas

B.B.A., The University of Georgia, 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 Ant Colony approach to the Snake-In-the-Box Problem

by

Shilpa Prakash Hardas

Approved:

Major Professor:	Walter D. Potter
Committee:	Robert Robinson
	Khaled Rasheed

Electronic Version Approved:

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

Dedication

The entire credit for the completion of this thesis and my degree goes to my husband, Kunal Verma. Colloqially, the wife is the better half of a husband, but in this case I truly attribute the better part of me to him. He motivates me, by example, to be a more patient and focused person. His belief in me has made it possible for me to take on tough challenges and see them through. I also can not thank him enough for holding my hand through the initial process of learning programming.

My Masters degree will be the culmination of my 7 year long education in the United States. If my dedication were to the include people who sacrificed a lot for the sake of my academic endeavour, the first ones on the list would be my family. Over the years they have supported me in all my endeavours by providing me with the resources and ample amounts of faith required to live and work alone away from home .

This thesis grew out of a class project that I did in an Expert Systems class with Makiko Kaijima. Tons of credit goes to her for not only being a hard working team mate, but also for being a wonderful friend. I also want to thank her for all the help she has extended to me, during preparations for my oral comprehensive exam and my thesis writing.

Acknowledgments

I wish to thank all my fellow students at the Artificial Intelligence Center, who made the excruciatingly long hours spent in the basement much more pleasant. In particular, mention is due to Julian Bishop and Daniel Tuohy, who have graciously extended their time to help me grasp certain “snake” concepts and discussed some of their work with me. Dr. Potter, with his great enthusiasm for the snake problem, has largely been the reason for the origin of this thesis. I would like to thank him for all his guidance not only in my thesis, but also in helping me understand the fundamental concepts of Artificial Intelligence during all the classes I took with him.

TABLE OF CONTENTS

List of Figures

List of Tables

Chapter 1

Introduction

The Snake-In-the-Box (SIB) problem consists of finding "snakes" and "coils" in a hypercube. Snakes and coils are paths in a hypercube which share the two following properties: 1) there are no repeated nodes in the path, 2) any two nodes that are not connected by an edge in the hypercube cannot be adjacent in the path. While snakes are open paths (a snake cannot travel to a node that is adjacent to any other node in the snake, except the previous node) coils are closed paths (a coil starts and ends at the same node). The applications of this problem range from electronic locking schemes [Kim, 2000] to error correction in analog to digital codes [Klee, 1970]. The longer the snakes and coils, the more useful they are in the application. Since the SIB problem was first introduced in 1958 [Kautz, 1958] it has been the subject of a great deal of research. The search space for this problem grows at an exponential rate with the increase in dimension of the hypercube. This makes it impractical to apply traditional search techniques to this problem. As a result, different heuristic approaches have been proposed by researchers to find the longest snakes in dimensions higher than 7.

1.1 Thesis Motivation

The most successful heuristic approaches for the SIB problem so far have been population based evolutionary approaches like genetic algorithms [Potter et al., 1994; Bitterman, 2004] and a population based stochastic hill climber [Casella, 2005]. This thesis attempts to move away from the conventional wisdom regarding this problem and focuses on applying Ant Colony Optimization (ACO) to the SIB problem. This is with the objective of gaining additional insight into the problem as well as developing a problem-specific ACO algorithm. The

ACO framework refers to algorithms which solve combinatorial optimization problems by simulating the foraging (food gathering) behavior of an ant swarm. The term ACO is used in the remainder of this thesis to refer to properties, techniques and algorithms that belong to this framework. We chose to apply ACO to this problem for two reasons: 1) ACO models the path finding behavior of ants and thereby lends itself naturally to path finding problems like the SIB problem. 2) We also expect that ACO will offers certain computational advantages (like less time and memory requirements) over the population based evolutionary approaches that have been used before.

1.2 Creating an ACO Framework for the SIB Problem

In this thesis, the SIB problem has been tackled using three different ACOs: 1) The Ant Colony System (ACS) [Dorigo and Gambardella, 1997], 2) The Min-Max Ant System (MMAS) [Stützle and Hoos, 1997] and 3) our variation on the MMAS, which we will call MMAS_SIB. Due to the similarities between the Traveling Salesman Problem (TSP) and the SIB problem we chose to focus on the ACS and the MMAS; which are currently the most experimentally successful ACO algorithms [Dorigo and Stützle, 2004:150] for the TSP. Subsequently, we created MMAS_SIB to combine the best features of the two. To improve the performance of the algorithms a seeding strategy (which used the longest snake in the previous dimension as a starting point) was employed in dimensions higher than 7. Testing done with all three algorithms in dimension 5-9 yielded encouraging results. Our empirical evaluation proves that MMAS_SIB is the most successful ACO algorithm across all the dimensions that were tested.

1.3 Contributions and Outline

The contributions of this thesis are as follows: 1) To the best of our knowledge, this is the first research work to address the SIB problem using ACO. 2) It introduces a new ACO algorithm, MMAS_SIB, which combines the best features of the ACS and the MMAS. 3)

Two of the algorithms presented in this paper have matched world records for the maximum length snakes in dimensions 5-8 found by other computing and time intensive approaches. 4) Results from this research prove the feasibility of an ACO approach to the SIB problem. The rest of this thesis is organized as follows: Chapter 2 explains the SIB problem and its domain in greater detail, Chapter 3 provides a detailed explanation of the biological basis of ACO and the computational components of the ACO algorithm, Chapter 4 describes our implementations of ACO and problem specific changes that were made, Chapter 5 presents results for all three ACOs and Chapter 6 discusses conclusions and future work.

Chapter 2

Snake-In-the-Box

2.1 Snakes and Coils

A snake is an acyclic achordal path in a hypercube. A snake of length 4 in dimension 3 is shown in Figure 2.1. A hypercube usually refers to an undirected graph on dimensions greater

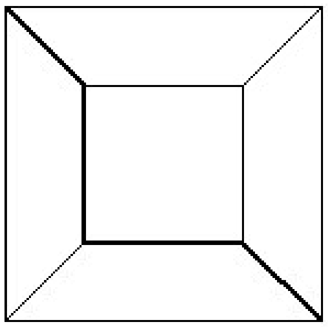

Figure 2.1: A snake of length 4 in dimension 3

than 3, which has 2^d vertices (nodes), $d(2^{d-1})$ edges and each vertex has exactly d neighbors. An achordal path through this undirected graph means a path which does not repeat nodes; has the property that any two nodes that are adjacent in the path are necessarily adjacent in the hypercube and; cannot visit a node that is adjacent to any previously visited nodes, except the node it just came from. A coil is a cyclic path which which does not repeat nodes; has the property that any two nodes that are adjacent in the path are necessarily adjacent in the hypercube and; whose last node is adjacent to its first node. The SIB problem refers to the open problem of finding longest length snakes and coils in dimensions greater than

7. Henceforth we will refer to a hypercube in dimension d as Q_d, the length of the longest snake in dimension d as $s(d)$ and the length of the longest coil in dimension d as $c(d)$.

2.2 Hypercube Representation

From the definition of a hypercube it follows that all nodes within a hypercube can be represented by binary numbers from 0 to $2^d - 1$, using a maximum of d bits to represent each node. Gray codes are binary number labels used for nodes, where adjacent nodes differ by only 1 bit. This property proves to be very helpful in determining adjacencies and thereby

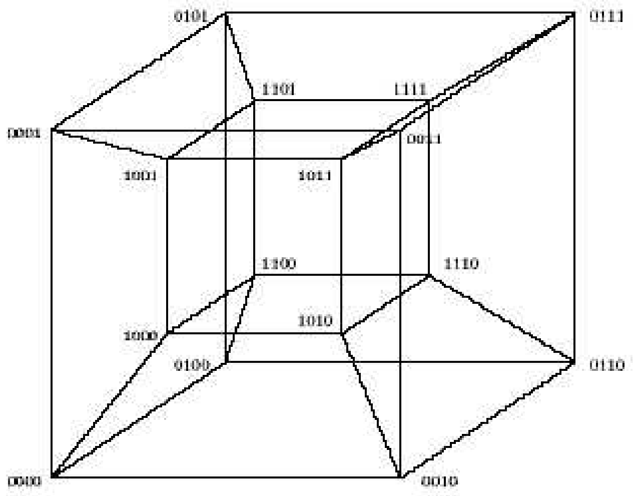

Figure 2.2: A 4-D hypercube with Gray code node representation

the validity of a path through a hypercube. Using Gray code all 16 nodes in a hypercube of dimension 4 (Q_4) are represented as follows:

0000 0001 0010 0011 0100 0101 0110 0111

1000 1001 1010 1011 1100 1101 1110 1111

Notice in Figure 2.2 that nodes 1, 2, 4 and 8 are adjacent to node 0, as they differ by one bit. Similarly nodes 0, 3, 5 and 9 are adjacent to node 1. So a valid snake through Q_4 can be represented as:

[0000, 0001, 0011, 0111]

When the binary numbers are converted to integers we get the node sequence of this snake.

[0,1,3,7]

Node sequences have been used in several construction techniques for snakes [Casella, 2005; Brown, 2005]. Another type of representation that is commonly used to denote a snake is a transition sequence [Potter et al., 1994]. A transition sequence denotes the bit position that has to be flipped in the binary representation of a node to get to the next node in the snake. The length of a transition sequence is always one less than the length of the corresponding node sequence. The advantage of using a transition sequence to represent a snake is that a single transition sequence can represent many different snakes, depending upon the starting node of the snake. Due to this property, transition sequences have been widely used in snake hunting [Potter et al., 1994; Bitterman, 2004].

2.3 Related Research

The body of work about the SIB problem can be divided into two categories. The first would include the theoretical and analytical work based upon graph theory and logic. Some of this work was aimed at getting a better understanding of the problem, and some presented new lower and upper bounds for the problem [Kautz, 1958; Deimer, 1985; Abbott, 1991; Snevily, 1994; Paterson, 1998; Rajan and Shende, 1999]. The lower bound refers to the longest snake or coil found in a particular dimension which serves as a benchmark for future research. The upper bound refers to the longest snake that in theory can be found in that dimension. Bridging the gap between snake lengths that mark the bounds, until the lower and upper bound are the same, constitutes solving the problem for that particular dimension. The second type of research focused on finding new snakes and tightening the previously proposed upper and lower bounds on the problem. This was done using exhaustive search

[Kochut, 1996], and by using heuristic search [Potter et al., 1994; Bitterman, 2004; Casella, 2005].

Table 2.1: Longest known snakes and coils dimensions 3-7.

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

Table 2.2: Lower (lb) and upper (ub) bounds for dimension 8-11

d	$lb\ s(d)$	$ub\ s(d)$
8	97 [Rajan and Shende, 1999]	123 [Snevily, 1994]
9	186 [Casella, 2005]	250 [Snevily, 1994]
10	358 [Casella, 2005]	504 [Snevily, 1994]
11	680 [Casella, 2005]	1012 [Snevily, 1994]

Table 2.1 shows the maximum lengths of snakes and coils found in dimensions 3 through 7. The exhaustive search in dimension 7 took over a month on 5 SUN Microsystems Sparc 1000's with two processors each [Kochut, 1996]. Scaling this up to dimension eight or higher poses very serious practical issues. This is the primary reason why heuristic searches have proved to be much more effective in this area as compared to traditional techniques. [Potter et al., 1994] used a genetic algorithm to find a snake of length 89 in dimension 8. [Bitterman, 2004] matched the record of 97 in dimension 8 by [Rajan and Shende, 1999] and found a new lower bound of 182 in dimension 9, using genetic algorithms with the narrowest path heuristic. A population based stochastic hill climber was used by [Casella, 2005] to improve these bounds further as shown in Table 2.2. Both [Casella, 2005] and [Bitterman, 2004] used a seeding scheme of longest[1] snakes from the previous dimension based on the conjecture

[1]Longest snake refers to the maximum length snake that can be found in a dimension.

[Rajan and Shende, 1999] that there exists a longest snake in dimension d that contains a maximal snake[2] from dimension $d-1$. This conjecture was later disproved by the authors themselves, but led to the construction of many new snakes.

2.4 Search Space for the SIB Problem

The search space for the SIB problem has proved to be a challenge for traditional as well as heuristic searches. The size of the search space is so enormous that examining every possibility is virtually impossible, thereby ruling out using traditional search techniques. Heuristic techniques face challenges in this domain due to the lack of knowledge about the search space for the SIB problem. It is difficult to foresee which operators will perform well in a search space without knowing the charachteristics of the search space. Some of the difficulties with this domain are:

- Neighbourhood structure: Not only is it impossible to judge what the length of the optimal solution would be, but it is also difficult to decide the neighborhood structure. This refers to all the problem states that lie next to a given problem state, meaning all the states that could lead to the current state and all the states that the current state can lead to. For example [Stützle and Hoos, 2000] comment about the search space for the TSP by stating that better solutions can be found close to good solutions. Making this type of assertion for the SIB problem would be impossible. The notion of a problem state can be better illustrated if the search space is imagined as a graph that contains all the partial and complete solutions to a problem. The problem consists of achieving certain values for say x number of criteria. A problem state is a point in the graph that has a certain set of values on all x criteria. An operator is defined as a set of operations to be performed on a problem state that changes the values of some or all of the criteria. A goal can be imagined as a point in the graph that has the desired

[2]Maximal snakes are snakes that cannot be extended in either direction. Using this definition longest snakes are also maximal snakes.

values of all x criteria. How close a problem state is to a goal state is determined by how many times the operator has to be applied in order to reach the goal state from the problem state. Establishing this sort of detail for the SIB problem poses several issues. Firstly, in dimensions above 7 the goal state of the problem is not known since the length of the longest snakes in these dimensions is not known. Secondly, even if the length to be achieved was known there is no way to tell for sure whether a particular snake can make it from its current length to the goal length, unless the entire snake is constructed. Thirdly, even if the effort was taken to construct the entire snake we could not always choose the correct operator to get closer to the goal (it would essentially be trial and error to see which nodes take us to the longest snake). Therefore in the case of the SIB problem we cannot establish a neighborhood structure (this refers to the proximity map of the various problem states and solutions in relation to each other).

- Local optimum: Local optimum refers to a solution that is considerably better than other solutions in its surrounding area but might not be the globally best solution. Knowledge of local optima is important to keep a search from converging on a sub-optimal solution. Due to the difficulty with establishing a neighborhood structure speculating about the presence or location of local optima is also a lost cause.

- Local search: Designing local search procedures[3] for the SIB problem is tedious and computationally intensive. Unlike local searches for the TSP, which change a specified number of edges to get a better solution, any local search for the SIB problem will essentially have to rebuild the snake from the first node that was changed. This is because changing a node will change the validity of all future nodes in the snake. Also, the benefit of the procedure would come not from rebuilding the snake, but only from extending it which will require further computation.

[3]A local search procedure takes a complete solution and either alters or adds to the solution to make it better.

- Heuristic look ahead: An important component of heuristic searches is estimating the potential of solution components for guiding future search efforts. In searches for the SIB problem nodes cannot be assigned a static heuristic value.[4] This is due to two reasons. Firstly, every snake is different, in terms of the nodes in its previous path and choices of nodes it will encounter in the future. This necessitates that a node have different heuristic values for different snakes. Secondly, even after dealing with the first stipulation, the choices of nodes the snake makes in the future can change the heuristic value of the node. This means that the search techniques can evaluate snakes that are already built but cannot put an estimated measure of goodness on the snakes that will be constructed in the future using the same nodes.

Our choices of algorithms and modifications made to them were governed to a large extent by the constraints that were just discussed.

[4]The term heuristic value is used here to mean a numerical value on the potential of a node to produce good solutions in the future. This choice of terminology roots from the literature on ACO, which uses the same term.

Chapter 3

The ACO Meta-heuristic

3.1 Overview of Ant Colony Optimization

Ant colonies demonstrate various group behaviors like foraging (food gathering), sorting dead bodies and nest building. The self-governed complex organizational patterns these ants demonstrate were studied by researchers to gain a better understanding of how ant colonies work. An experiment run by [Deneubourg et al., 1990] showed that foraging in a swarm of Argentine ants was a self-organizing behavior. When a swarm of real ants are given access to a food source by different paths of varying length the traffic of ants quickly converges on the shortest path from the nest to the food source [Goss et al., 1989].

Figure 3.1 shows the convergence of ants on the shorter path. This and such other organized behaviors of ant swarms have inspired various heuristic search techniques which have been loosely termed “ant algorithms”. Ant algorithms are multi-agent systems in which the behavior of each agent (artificial ant) is modeled by the behavior of real ants [Bonabeau et al., 1999]. Ant Colony Optimization (ACO) which is the subject of this work is an umbrella term for a family of ant algorithms inspired by the foraging behavior of ants, that have been used to solve discrete combinatorial optimization problems. ACO algorithms have been applied to a variety of problems like the TSP [Dorigo and Gambardella, 1997], the job shop problem (JSP) [Colorni et al., 1994], the capacitated vehicle routing problem (CVRP) [Bullnheimer et al., 1999], the quadratic assignment problem (QAP) [Stützle and Hoos, 2000], and the resource constrained project scheduling (RCPSP) [Merkle et al., 2002].

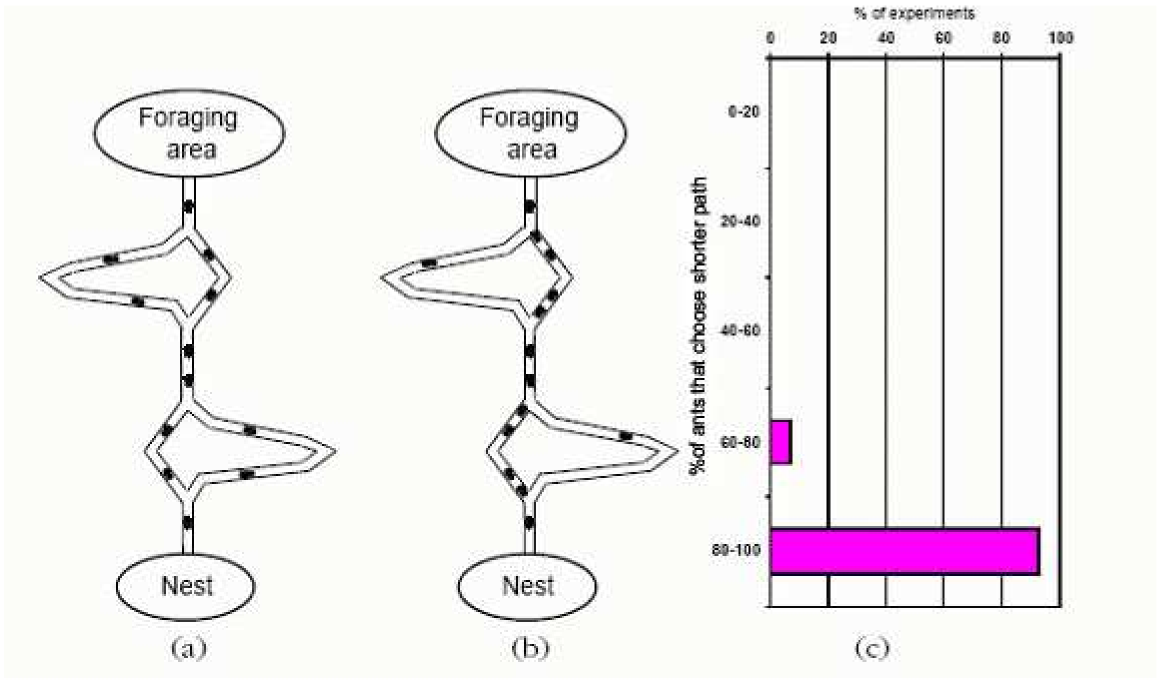

Figure 3.1: Double bridge experiment. (a) Ants start exploring the double bridge. (b) Eventually most of the ants choose the shortest path. (c) Distribution of the percentage of ants that selected the shorter path. After [Goss et al., 1989]

3.2 Autocatalysis, Differential Path Length and Stigmergy

So the question arises, how do small creatures like ants, exhibit such intelligent behavior when put in swarms? The emergence of this problem solving behavior can be attributed to stigmergy, autocatalysis and differential path length. The ants explore until they find the food source. Once they find the source they lay a chemical substance called pheromone on their way back to the nest, to let other ants know where the food is. This indirect communication is called stigmergy. Other ants follow the path with the strongest pheromone. This positive feedback that encourages other ants to follow the pheromone is called autocatalysis. The ants taking the shorter path can make more trips in the same amount of time as compared to other ants. This leads to a build up of pheromone on the shorter path which causes more ants to follow, thereby adding more pheromone and attracting more ants [Deneubourg et al., 1990]. Differential path length (different path lengths) between the available paths is what makes one particular path better than the others. While stigmergy, autocatalysis and differential path length help find the shortest path, there is also a need to avoid longer paths and abandon bad paths that have been explored before. Depending upon the species, the mechanism used to avoid longer paths could differ. Most species use more than one type of pheromone. Some pheromones signal positive reinforcement, and some dissuade other nest mates from following a path. *Lasius niger*, another type of ant makes U-turns on long paths when it realizes that it is going almost perpendicular to the source of food [Beckers et al., 1992]. However, the most important factor which influences the abandoning of bad paths is the evaporation of pheromone. This happens naturally just like the dissipation of scents in air. Evaporation guarantees that over a course of time ants forget paths which have not been reinforced. Thus the swarm collectively moves away from the longer paths. The experimental model [Deneubourg et al., 1990] illustrated in Figure 3.1 consisted of a swarm of ants, which was given access to a food source by a double bridge (two different paths). The two paths are referred to as "the upper branch" and "the lower branch". An interesting pattern occurred when the two branches were of different lengths. Over time the ants converged on the shorter path. The choice equation which shows the probability of an ant choosing one branch over the other [Goss et al., 1989] is shown in Equation 3.1. Equation 3.1 does not consider the evaporation of pheromone, since the experiments were run over the period of an hour, which is not long enough for evaporation to take place.

$$P_U(m) = \frac{(U_m + k)^h}{(U_m + k)^h + (L_m + k)^h} \tag{3.1}$$

where,

L_m = Number of ants that have taken the lower branch

U_m = Number of ants that have taken the upper branch , $m = L_m + U_m$

$P_U(m)$ = Probability that the $(m+1)^{th}$ ant will choose the upper branch

$P_L(m) = 1 - P_U(m)$, probability that the $(m+1)^{th}$ ant will choose the lower branch

The ant choice rule then becomes $U_{m+1} = U_m + 1$ if $\psi < P_U(m)$, else $U_{m+1} = U_m$, where ψ

is a random variable uniformly distributed over [0,1]. The variables h and k allow to fit the model. Equation 3.1 shows that the probability of choosing a branch is positively correlated to the number of ants that have chosen that path before, which in turn is positively correlated to the pheromone on the path. Equation 3.1 provided the biological basis for the ACO choice rules which will be discussed in Chapter 4.

3.3 Algorithm for the ACO

The computational equivalent of the model discussed in Chapter 3.2 consists of artificial ants, which we will call ants. Since this work deals specifically with Ant Colony Optimization, we will discuss the general procedure followed by ACO. Ants travel from the starting state of a problem to the goal state while incrementally building a solution. Each ant uses a probabilistic rule similar to the one in Equation 3.1 to choose its next step. The choice rule can take different forms and will be discussed in detail in Chapter 4.1. This process models the search for the food source, which real ants undertake. The return journey of real ants is implied in a computational framework. However, pheromone laying which happens on the return journey is accounted for. Once all the ants construct a solution, pheromone is deposited on the steps that comprise the good solutions. The pheromone from all steps is also evaporated to simulate the effect of time. Once the solution construction and pheromone functions have been executed, other functions that might be necessary to ensure the proper functioning of the program are done. The steps mentioned above are repeated until one of the stopping criteria for the problem is fulfilled (these can include finding an optimal solution, no improvement for a considerable time, or completion of the specified number of program iterations). This ACO algorithm is outlined in Figure 3.2. It will serve the dual purpose of supporting the discussion about ACO in the following chapters as well as presenting the algorithm for all three ACOs (ACS, MMAS and MMAS_SIB) implemented during the course of this work. Different authors might follow different naming conventions for the procedures, but the algorithm presented here includes all the necessary components of ACO. For a more detailed version of the ACO meta-heuristic refer to [Dorigo and Di Caro, 1999].

```
ACO_snake_in_box (max_gen, num_ants)
    Num_gens = 0
    num_ants = m
    Initialize_pheromone_table()
    createAnts (num_ants)
    Adjacency_table()
    While (Num_gens < max_gen)      //This is a generation
       Do                           //This is an iteration
        For each ant 1 to m
           Next_ant_move(ant_id)    //Uses Equation 4.1
           Update_ant_path(ant_id)
           Local_pheromone_update(ant_id)   //Uses Equation 4.4
       While some ant has not constructed a maximal snake
       Global_pheromone_update()         //Uses Equation 4.3
       Reset_ant_paths()
   End While
```

Figure 3.2: ACO algorithm for the SIB problem

Depending upon the nature of the problem and the search space various diversification and intensification mechanisms are used to guide the search. Diversification opens up previously unexplored parts of the search space and introduces new solutions; while intensification focuses the search around previously found good solutions. A balance of both these mechanisms is necessary for the search to yield good results. Too much diversification means that ants will continue their random walk forever and will not converge on a good solution. On the other hand if the intensification of the search happens too fast then the search will stagnate. This means that it will not have enough time to find good solutions and all ants will prematurely converge on a sub-optimal solution. The use of these mechanisms (to guide the search) combined with some other parameters (which will be discussed in Chapter 4) make for very different approaches towards exploring the search space. Various ACOs differentiate themselves by how they use these factors to influence their searching capabilities. We devote

Chapter 4.6 to comparing and contrasting these using the ACS, MMAS and MMAS_SIB as examples.

3.4 Properties of ACO

ACO is a relatively new meta-heuristic and thus much of the work for the theoretical basis of ACO has just begun. [Gutjahr, 2000] proved the convergence in solution for an experimental setup called GBAS (Graph Based Ant System). More relevant work comes from [Stützle and Dorigo, 2002] in reference to a class of ACO's called ACO_{gb}, τ_{min}. These are defined as ant algorithms which use the global best pheromone update rule and have a lower bound on the amount of pheromone. In short this refers to algorithms in which only the best solution found so far is reinforced and the pheromone present on all nodes satisfies a minimum value condition. The following properties were proved with regard to this class of algorithms:

- Convergence in result: "ACO_{gb}, τ_{min} is guaranteed to find an optimal solution with a probability which can be made arbitrarily close to one if given enough time" [Stützle and Dorigo, 2002]. What this really means is that the heuristic can find an optimal solution to the problem but the time required to do so could be very large.

- Convergence in solution: "After a fixed number of iterations has elapsed since the optimal solution was first found, the pheromone trails on the connections of the optimal solution are larger than those on any other connection" [Stützle and Dorigo, 2002]. The above result is also extended to give a lower bound on the probability of the optimal solution being constructed.

Suffice it to say that these results apply to all three of the algorithms which will be discussed during the course of this paper since the ACS, the MMAS and MMAS_SIB all use the global best pheromone update rule and have a lower limit on the pheromone trails.

Chapter 4

Adapting ACO to the SIB Problem

This chapter focuses on strategies and computational techniques used to build an ACO framework for the SIB problem. Since there has been no prior work in this area; these modifications were inspired by various ACO implementations for the TSP combined with the problem specific knowledge for the SIB problem. [Bonabeau et al., 1999] state that it is possible to apply ACO to any combinatorial problem as long as it is possible to define:

1. Problem representation which allows ants to incrementally build/modify solutions by using a construction/modification probabilistic transition rule.
2. Heuristic desirability of edges.
3. Constraint satisfaction method which forces construction of feasible solutions.
4. Pheromone update rule.

All three of the ACO algorithms have been implemented keeping in mind these four criteria. The following discussion which provides details of the ACOs is also structured around these four components.

4.1 Problem Representation

In our implementation the ants build a sequential solution by traveling from one node of the hypercube to an adjacent node. These nodes are represented by numbers ranging from 0 to $2^n - 1$, where n is the dimension of the hypercube. For example, in dimension 8 the nodes are numbered from 0 to 255. When the ant has no more feasible nodes left it returns a solution starting from node 0. When all ants have exhausted their feasible nodes the generation is said to be over. The ants then reset their paths to start again at 0. The term generation in Figure 3.2 has been introduced to maintain a distinction between the two main loops used in the program and is closer in meaning to iteration. It should not be confused with the same term used in reference with evolutionary techniques like genetic algorithms. At the end of each generation every ant will return a feasible solution of varying length depending upon the choices of nodes it has made through out its journey. Since paths of different lengths are found the desirability of good solutions is established as per the differential path length effect (discussed in Section 3.2). Solutions are represented as integer node sequences. The most important part of this process of building a solution is the choice of the ant's next step which is referred to as `Next_ant_move(ant_id)` in Figure 3.2. Equation 4.1 [Dorigo et al., 1996] shows the state transition rule which is also called the random-proportional rule.

$$p_k(r,s) = \begin{cases} \frac{[\tau(r,s)]^{\alpha} \cdot [\eta(r,s)]^{\beta}}{\sum_{u \in J_k(r)} [\tau(r,u)]^{\alpha} \cdot [\eta(r,u)]^{\beta}} & \text{if } s \in J_k(r) \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

where,

$J_k(r)$ = list of all feasible nodes at r, according to the problem constraints in Chapter 4.3

$p_k(r,s)$ = probability of choosing node s after node r for the k^{th} ant

$\tau(r,s)$ = pheromone on edge (r,s)

$\eta(r,s)$ = heuristic desirability of going from node r to s according to Chapter 4.2

α = intensity of the pheromone trail

β = visibility of the heuristic information

The pheromone is the memory of the past search efforts of the ants; it quantifies the quality of the solutions explored so far using the node. The heuristic information provides a way to measure the potential of a node to yield good solutions in the future. The desirability of a node is the combination of its past contributions to good solutions (pheromone) and an estimate of its future worth (heuristic value). The two parameters α and β control the relative importance of pheromone *versus* heuristic information. The selection of the next node using Equation 4.1 is very much like roulette wheel selection in evolutionary computation. A random number between [0,1] is generated and a node is chosen according to the probability distribution in Equation 4.1. Nodes that have a higher probability $p_k(r, s)$ have a better chance of being chosen. Equation 4.1 was used in our implementation of the MMAS. The ACS and MMAS_SIB use a different version of this rule which is shown in Equation 4.2 [Dorigo and Gambardella, 1997]

$$s = \begin{cases} argmax_{u \in J_k(r)} \left\{ [\tau(r,u)]^{\alpha} \cdot [\eta(r,u)]^{\beta} \right\} & \text{if } q \leq q_0 \\ S & \text{otherwise} \end{cases} \tag{4.2}$$

where,

q is a random number uniformly distributed in $[0, 1]$ and q_0 is a parameter such that $0 < q_0 < 1$. S is a random variable according to the distribution in Equation 4.1. In effect, when $q < q_0$ the best node will be chosen with the probability of one. This rule is called the pseudo-random proportional rule [Dorigo and Gambardella, 1997]. This rule affords the ACS and MMAS_SIB a parameter that can be used to directly exploit good solutions. Fine tuning of the exploitation parameter q_0 is required to give the algorithm a balance between "exploitation" and "biased exploration". [Dorigo and Gambardella, 1997] refer to always choosing the best node as "exploitation" and the probabilistic selection in Equation 4.1 as "biased exploration".

4.2 Heuristic Desirability of Edges

The evaluation of a solution/ solution component needs to be done twice during ACO. The first is when the ant attributes an estimated value to each of its possible next steps; this is referred to as the heuristic desirability of the edges. The second is when the quality of the final solution is assessed at the end of each generation. Length, [Potter et al., 1994], "tightness" [Casella, 2005] and narrowest path heuristic [Bitterman, 2004] are some of the criteria that have been used before to determine the fitness of a snake. However, the step by step solution construction in ACO poses some difficulties in using these measures. The heuristic desirability is associated with all possible future nodes of the same snake. This means that any function of length or tightness associated with the same snake will have the same value and hence cancel out for all nodes. The narrowest path heuristic works on the assumption that the more valid nodes the snake has left, the better its chances are of being a longer snake. Even though this is true to some extent, Figure 4.1 will illustrate that it is not only important how many nodes are left but also whether the snake can get to those nodes from the present node. Figure 4.1 graphs at every point through the snake the number of unavailable nodes divided by the length of the snake. We will refer to this as the average number of unavailable nodes per node. The figure includes 3 snakes of length 47, 74 and 97 in dimension 8. As shown in Figure 4.1, the shorter snakes never drop down below an average of 4 unavailable nodes per node while the longest snakes drop down all the way to 2.5 for every new node added. The longer snakes can effectively use many of the available nodes, by "overlapping" many nodes and making node choices that allow access to the remaining unvisited valid nodes. When a new node is visited, $n-1$ other nodes become invalid (assuming that one of the n neighbor nodes of that particular node is already on the snake and thereby invalid), where n is the dimension. Overlapping refers to when a snake marks some nodes invalid that are already invalid thereby saving valid nodes for further expansion. The change in average unavailable nodes is a good measure of how promising the next node is since it accounts implicitly for length and tightness. The change in average unavailable nodes at node n is defined as the difference between average unavailable nodes at length t and the average unavailable nodes at length $(t+1)$, if n is the next node chosen. It also suits ACO rather well, since ACOs have mostly been used for minimization problems, and minimizing this change for every node corresponds to efficient use of the available nodes. In most ACO implementations the heuristic value of each node is a static value that remains unchanged during the course of the program. However in case of the SIB problem this value is dynamic depending upon the previous part of the snake which requests the value.

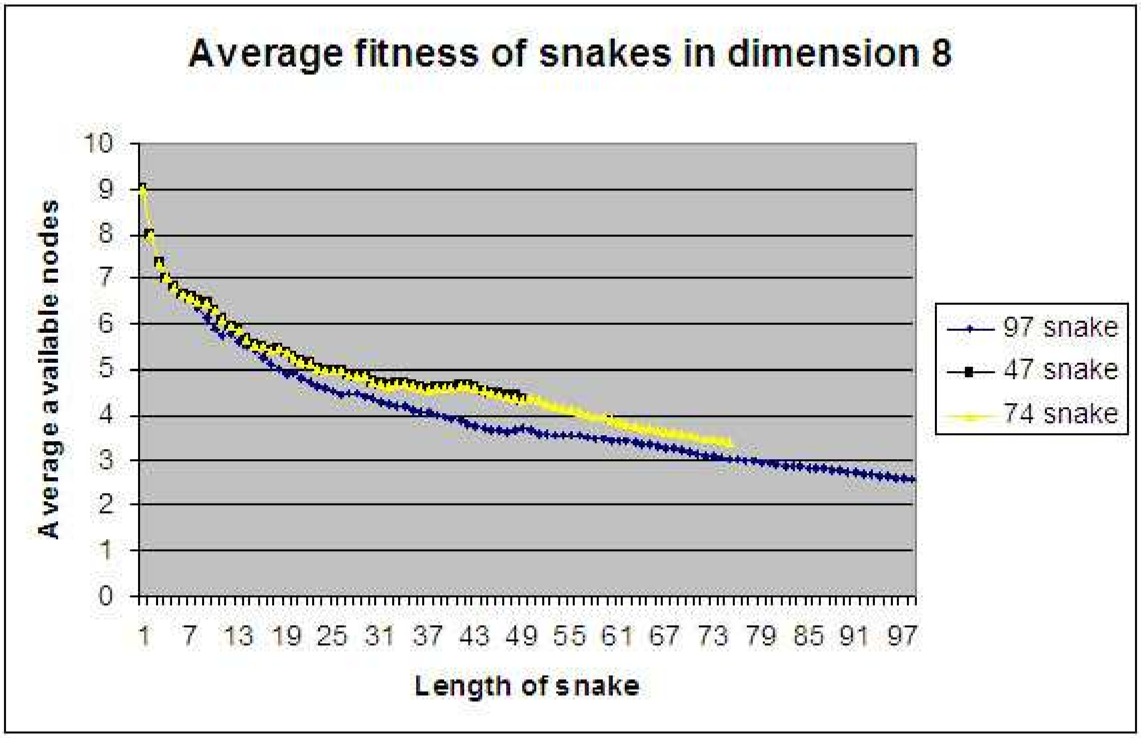

Figure 4.1: Change in average unavailable nodes as snake length increases. Data provided by [Tuohy, 2005]

4.3 Constraint Satisfaction

To construct a valid solution it is important to ensure that the problem constraints are never violated during solution generation. To achieve this in ACO, we need to make sure that every step the ants take is a valid node for the snake. Each ant has a memory of the visited nodes and uses this list to avoid constraint violation. As discussed in Chapter 2.1 in addition to already visited nodes a snake cannot visit any nodes adjacent to a previously visited node. This second constraint is checked using a look-up table which is called an adjacency table. The ants can travel only to other nodes that have connections to the present node but not any other node on its path. These conditions guarantee that a valid snake will be generated by each ant.

4.4 Pheromone Update Rule

Deposition and evaporation of pheromone are the two main reasons behind the functioning of ACO. The deposition of pheromone guides the search from past experience while the evaporation maintains dynamism (change in the desirability of edges) in the search space, by erasing the pheromone trails for bad solutions found in the past. Figure 3.2 refers to two pheromone update rules, the `local_update()` and the `global_update()` that deal with pheromone. There are differences in the way these two rules are implemented and the role they play in ACO.

The `global_update()` rule is an essential part of every ACO algorithm. It reinforces successful solutions found in each generation by laying pheromone on the edges of the best solution while also evaporating pheromone from all edges. The amount of pheromone deposited depends on the quality of the solution found but the evaporation is a constant amount determined by a trail evaporation parameter. Equation 4.4 shows the `global_update()` rule.

$$\tau(r,s) \leftarrow (1-\alpha)\cdot\tau(r,s) + \alpha\cdot\Delta\tau(r,s) \tag{4.3}$$

where,

α is a parameter of trail evaporation

$\Delta\tau(r,s)$ is the change in pheromone at edge (r,s)

To determine how much pheromone the best solutions should deposit, two different measures of change were used. The first one was a function of the length of the snake and the second one was the length of the snake multiplied by the inverse of average unavailable nodes. The second measure would favor snakes that were tighter but also has an inherent advantage for longer snakes. This is because the smaller the value of average unavailabe nodes the longer the snake will be.

All three models of ACO that have been implemented here are elitist. This means that only the best solution lays pheromone at the end of the generation. The best solution maybe the global best (gb) or the iteration best (ib). The global best is the best solution the algorithm has found over all the past generations, while the iteration best is the best solution found in the last generation. In MMAS and MMAS_SIB, we use a schedule of oscillating between the iteration best and global best which favors the global best towards the end. The ACS by nature is built to exploit the best solutions and so performs slightly better when the global best solution is chosen to lay pheromone, even though both gb and ib have been used in trials.

The `local_update()` rule is specific to the ACS. This function is meant to ensure that all ants do not repeat the same path. This is likely to happen if more than one ant is on the same node. In such a case, they would all have to choose between the same neighbor nodes (with the same pheromone value distribution) thereby increasing the likelihood of choosing the same dominant node. The `local_update()` method is performed immediately after each ant takes a step and it reduces pheromone on the node that has been chosen thereby making it less attractive to other ants. This ensures diversification of the search effort. Equation 4.3 shows the `local_update()` rule

$$\tau(r,s) \leftarrow (1-\rho)\cdot\tau(r,s) + \rho\cdot\Delta\tau(r,s) \tag{4.4}$$

where,

ρ is a parameter of trail evaporation

$\Delta\tau(r,s)$ is the change in pheromone at edge (r,s)

Two variations on the `local_update()` rule were tried. The first one set the change in pheromone ($\Delta\tau(r,s)$) to 0, which meant that the only effect on pheromone was that of trail evaporation. The second one was inspired by [Dorigo and Gamberdella, 1997] where $\Delta\tau(r,s)$ was equal to the pheromone at the node, when it was initialized. This makes the initialization values of the pheromone very critical. Trials were conducted with random pheromone values at every node and with pheromone equal to the optimal snake length in every dimension at every node. Setting the change in pheromone to the initial pheromone value at the node proved to be more successful, however there was not a significant difference between the random value versus the optimal snake length value. Therefore all the testing done with the ACS uses the initial pheromone as $\Delta\tau(r,s)$.

In addition to parameters discussed in Chapter 4.1 to 4.4 there are two more parameters that need to be set for each ACO; the number of ants (`num_ants`) and the number of maximum generations(`max_gen`). The number of ants determine how many solutions will be simultaneously explored in each generation. Even though it is important to produce several solutions in a generation the benefit of having many ants is lost when ants start duplicating each others path. So even for large problems there is a definite number of ants after which no incremental improvement is seen in solution quality. The number of maximum generations used in an ACO should be enough to let the search converge. Convergence[1] can occur in different ACOs at different times for different problem sizes. As a general rule for all algorithms the number of maximum generations increase with problem size. The parameters for trail evaporation, number of ants, maximum generations and q_0 for the ACS were optimized during testing. Not surprisingly all three versions of ACO, namely the ACS, the MMAS and MMAS_SIB problem worked well on different parameters.

4.5 Optimizations for the SIB Problem

Some changes were made in our algorithms keeping in mind problem specific information for the SIB problem. Most of these were aimed at reducing the size of the search space. A snake starting at any node is equivalent to another snake with the same transition sequence. Since a hypercube of dimension n has 2^n nodes; fixing the starting point of a snake helps exploit the symmetry of the hypercube to reduce the search space by a factor of 2^n. Additional efficiency can be gained by following the canonical[2] snake scheme used in [Kochut, 1996] which reduces

[1]Convergence in an ACO refers to a state when all ants follow the same path because one node at every step has more pheromone than the others.

[2]Canonical snakes have the constraint that the choice of the next node has to be made by flipping the bits in the Gray code of the present node from least to greatest significant order. This means that a bit in position 5 cannot be flipped unless the bit in position 4 has been flipped at least once.

the space by a factor of $n!$. This combined with fixing a start node reduced the search space by a factor of $n! * 2^n$, where n is the dimension of the hypercube. The interesting thing to note is that it also reduced the number of possible solutions by the same factor. In case of ACO, this did not yield better results because it converted the population based search to essentially a single point search. By restricting the node choices to canonical form the same snake was produced by many ants especially in the earlier parts of the search. The choice of retaining a fixed starting node but eliminating the canonical scheme was made.

Traditionally, all TSP applications of ACO use pheromone to reinforce the relative positions of the edges. This supports the notion that it is not important whether certain nodes come earlier or later in the solution, as long as they have been reached through the least cost path (other cities connected to it), they will make for a good solution. In the case of the SIB problem an alternate reinforcement scheme was also tried. The absolute position of a node within a snake was reinforced so as to encourage ants to pick certain nodes earlier in the snake. It made more sense to use this in conjunction with the canonical version of the algorithm since the first few nodes in a canonical snake are always fixed. For example [0,1,3,7] will always be the first 4 nodes of any canonical snake in dimension higher than 3. This alternate scheme significantly worsened the performance of ACO and was abandoned in favor of the traditional scheme in which the edges connecting two nodes receive pheromone.

The best results for the SIB problem have been obtained by using seeding [Casella, 2005]. This can be done in two different ways. In a genetic algorithm, individuals that are longest snakes in the previous dimension can be introduced into the population. In search techniques that can "grow" the snake, the search can be started from a good node sequence which can be a previously known longest snake in the lower dimension [Brown, 2005]. The benefit gained from seeding in both of the above mentioned schemes is apparent. However, using seeding to our advantage in ACO is a more tricky issue. A longest snake from a previous dimension can be a good start for the first generation of ACO but once the ants finish solution construction and reset their paths the seeding is lost. The pheromone trail will retain the path for some time and unless more long paths with the same nodes are found the trail will eventually evaporate. On the other hand reinforcing the trail too strongly will cause the search to stagnate. For these reasons the seeding scheme for ACO needs to be made more dynamic. For dimensions 7 and higher all snakes were seeded with the longest snake from the previous dimension at the start. Once ACO found a longer snake the seeding scheme was changed. One third of the ants were started from node 0, one third were seeded with the original seed and the remaining were seeded with two thirds of the length of the best snake found so far. The original seeds from the previous dimension were canonical and had all of the bits flipped at least once (the process of finding canonical snakes ends once every bit position has been flipped at least once in the transition sequence of a snake. From then on whatever nodes are added will produce distinct canonical snakes). Meaning that any snakes seeded with these would produce canonical snakes. This provided a good way to exploit this property even though previous efforts had not produced significantly better results. The search space being explored by each set of ants was different after seeding and was only a subset of the entire space. This increased the chances of finding longer snakes.

4.6 Techniques used for Intensification and Diversification

In Chapter 3.3 we discussed the role of diversification and intensification in ACO. We have also discussed the implementation details and parameters of the ACS, the MMAS and MMAS_SIB. Let us now see what properties these algorithms gain due to the interaction of all the components discussed earlier and how these factors can be controlled to get the desired result.

- Trail persistence/evaporation: To prevent ants from converging on a sub-optimal solution, the pheromone trail dissipates much like scents do in air. This allows for the exploration of new parts of the search space since ants resume their random walk in the absence of differences in pheromone. How much pheromone to evaporate decides how quickly the old paths are abandoned and new ones explored; this parameter is called trail evaporation. In the remainder of this paper the trail evaporation parameter refers to the trail evaporation (α) in the `global_update()` rule (Equation 4.4). A very low value of α is used in the ACS to retain memory of the past best solutions for a longer time to exploit them. While a very high value of α is used in MMAS to keep new best solutions in focus and quickly abandon old ones. MMAS_SIB tries to achieve a balance between direct exploitation and biased exploration and needs customized trail evaporation parameters for each dimension.

- Reinforcement of pheromone: This is achieved by how much pheromone is deposited by the good solutions. In the case of large amounts of pheromone being deposited, the good solutions will quickly develop a big difference in the amount of pheromone thereby drastically reducing the chances of other paths being chosen. The amount of pheromone deposited by the best solutions uses the same function of length in all three ACOs.

- Visibility of heuristic information: This parameter might be used advantageously to give nodes that have good potential but have not been explored or reinforced a better chance. The higher the visibility the more weight heuristic information has in the search process. Due to the difficulty in accurately estimating heuristic value (reasons for which are discussed in Chapter 2.4), we expect all three of the ACOs to perform well with very low or no heuristic information.

- Trail strength: Trail strength can achieve the opposite effect of heuristic visibility. It tells the search to rely heavily on the history of good solutions (amount of pheromone) as opposed to the value of future solutions. Having values of trail strength (Equation 4.1) higher than 1, will lead to the differences in pheromone between edges increasing exponentially. Due to the dangers of stagnation associated with this situation, no values other than 1 were tried. The differences in trail strength are expected to make the most difference in the ACS and versions of MMAS_SIB which have a high exploitation parameter(q_0). Since for the SIB problem the importance of heuristic information is expected to be very little, it is not expected to affect the probability of a node being chosen. This gives the trail strength complete control over establishing the dominant node (node with the highest probability of being chosen).

4.7 ACO Implementations for the SIB Problem

The following section outlines the three algorithms used for the SIB problem, and gives reasons for the development of each algorithm. It explains the distinct features of each algorithm and gives a synopsis of the performance of each algorithm by correlating it to domain charachteristics.

4.7.1 Ant Colony System

The two main aspects of the ACS are: the state transition rule, which allows for both the exploitation of the best edges in the past and exploration of new edges; and the local updating rule which maintains the dynamism in the pheromone trail. However, with greedy exploitation of the best solutions, the algorithm might stagnate. The pheromone on the best solution might become so strong that all ants take the same path; this is remedied by the `local_update()` rule. The function `local_update()` decreases pheromone on edges that have been chosen before; thereby increasing the likelihood of each ant constructing a distinct solution. To compensate for its voratious exploitation of the best edges the ACS provides very fast change in their desirability, if they do not belong to another best tour. Work done by [Dorigo and Gamberdella, 1997] proves this quality of the algorithm by doing an analysis of the pheromone levels on the best[3], recent best[4] and uninteresting edges[5]. When applied to the SIB problem the greedy exploitation characteristic of the ACS might keep it from

[3]Best edges are edges that belong to the best solution found in the last generation.

[4]Recent best edges are edges that have belonged to the best solution within the last two generations.

[5]Uninteresting edges are edges that have not been part of a good solution for over two generations.

completely exploring the search space as the problem size increases. Another concern would be that snakes which deposit pheromone are always maximal snakes (since ACO produces only maximal snakes at the end of each generation).Therefore following these maximal snakes too closely might lead the search down a dead end path. The exploitation parameter q_0 will need to be low especially in higher dimensions so that edges not belonging to the best solution have a chance of being chosen. Previously known good results using the ACS for the TSP have been obtained by combining the ACS with a local search procedure called 3-opt [Dorigo and Gambardella, 1997]. The 3-opt local search takes a complete solution and by changing 3 connections (TSP involves connecting a set of cities with a least cost path) produces a lower cost solution. Applying local search procedures to the SIB problem is not a very practical approach (reasons for which are discussed in Chapter 2.4) and this might hurt the performance of the ACS considerably. However, the seeding strategy is expected to significantly boost the performance of the ACS since the algorithm is built to exploit promising solutions.

4.7.2 Min Max Ant System

The Min-Max Ant System (MMAS) was explored as a possible solution to some initial problems that the ACS was facing in this domain. The MMAS was introduced [Stützle and Hoos, 1997] to provide a simple mechanism to prevent stagnation. Here is how it differs from the ACS: it has explicit limits on the strength of the pheromone trail and; it oscillates between reinforcing the global best and the iteration best solution so as to avoid only one solution being reinforced in every generation. The MMAS monitors the trail limits [6] and gradually increases them. This is done by having an explicit minimum (τ_{min}) and maximum (τ_{max}) limit on the amount of pheromone at each node and then gradually increasing the upper limit as better solutions are found. Stützle and Hoos point out that by dynamically changing the maximum limit and setting it to the best solution found so far, convergence

[6]Trail refers to the amount of pheromone on each node.

can be achieved. Convergence in terms of the MMAS means that at each step one node has the pheromone value τ_{max}, while all others have the value τ_{min} . The effect of all these strategies is that there is a constant increase in the pheromone on good edges and a constant decrease in the pheromone on edges that have not been part of the best solution. However, the trail limits ensure that no node ever has a zero probability of being chosen. The trail limits are enforced by first initializing all edges to a very large number and then setting them all to the max limit at the end of the first iteration. This ensures that the algorithm focuses on exploration in the beginning. At the end of each iteration the edges that have less pheromone than τ_{min} are set to τ_{min} and those that have more pheromone than τ_{max} are set to τ_{max}. MMAS uses iteration best solution (ib) to reinforce pheromone in the beginning of the search, and global best solution (gb) towards the end of the search. Since ib changes more frequently than gb, this ensures diversity in the solutions being reinforced. Another strategy to ensure changes in the best solution is a high trail evaporation parameter. This helps the search forget old solutions, thereby producing iteration best solutions that are significantly different from the global best. There were also some additional reasons for the choice of the MMAS: 1) Local search operators have been known to improve MMAS performance but its performance does not rely too heavily on them (unlike the ACS). 2) Initial tests with the ACS revealed that heuristic or problem specific information, which helps ascertain the potential of future solution components, was not very useful in case of the SIB problem. The model for MMAS works on the assumption that the value of heuristic information is low [Stützle and Hoos, 2000] making MMAS well suited to the problem. The expectation is that the MMAS will make up for the shortcomings of the ACS in this domain. The exploratory nature of the MMAS should give it good searching capability in the higher dimensions.

4.7.3 MMAS_SIB

Initial testing indicated that while the ACS did well on some dimensions the MMAS was better on others. The ACS found longer snakes, but the MMAS was far more consistent in the results it produced. By now it was clear that the search space in the SIB problem worked differently for different dimensions. The search for an algorithm that worked well on all the dimensions, led to MMAS_SIB. This ACO combines the aggressive exploration of the ACS with the more exploratory nature of the MMAS. Since the ACS and MMAS have complimentary searching capabilities MMAS_SIB should get the best of both worlds. MMAS_SIB uses the exploitation parameter (q_0) from the ACS combined with the trail limits (τ_{min} and τ_{max}) and reinforcement schedule (ib and gb) of the MMAS. The q_0 parameter of the ACS is used to gain a better chance of exploiting the best solutions while the trail limits and reinforcement schedule avoid searching in the same area. We ascertained earlier that heuristic information does not help too much in assessing what solutions to generate; it follows that vast exploration is required. However, in the size of the search space we are dealing with it is hard to undertake a comprehensive search effort. So exploring a variety of carefully chosen solutions should be a good way to find longer snakes. Since we have balanced the exploitation and exploration effort, it now becomes important for ants not to not duplicate their search from generation to generation. This is done by having a high trail evaporation to make way for newer solutions. Since it uses parameters from both the ACS and MMAS we expect that the parameters for MMAS_SIB will have to be fine tuned for each dimension. MMAS_SIB can mirror the searching capability of the ACS as well as the MMAS with minor changes in its parameters. This should give it the advantage of performing well on all dimensions. The performance results of all these algorithms are presented in the following Chapter.

Chapter 5

Results

This section will present the test results of all three algorithms: ACS, MMAS and MMAS_SIB. The ACS was the first ACO to be implemented and experiments were conducted with the two most important parameters, namely the exploitation parameter (q_0) and the trail evaporation parameter (α). This gave us a sense of which parameters affected each dimension and provided insight about the best performing combination of parameters for each dimension. The knowledge from experiments with the ACS was then used to design experiments for the MMAS. We observed that α has to be set very high for the MMAS (unlike the ACS which has a low α value). Hence, we test values for α ranging from [0.1 to 0.5] in the ACS and [0.5 to 0.8] for the MMAS (Note: MMAS does not use q_0). Depending upon which values of α and q_0 had performed the best for the MMAS and the ACS respectively, combinations of these were tried for each dimension in MMAS_SIB . We present parameter tuning results for the MMAS_SIB. All three algorithms performed better with the seeding scheme discussed in Chapter 4.6 and hence the testing results for dimension higher than 7 presented in this chapter are from the seeded versions of the algorithms. The chapter concludes with comparative results from all three algorithms in dimensions 5-9 and some observations about timing results.

5.1 Parameters

All the test results in this section have been obtained by using some standard parameters in each dimension. As initial tests showed much better performance without heuristic information, the heuristic information parameter (β) is always set to 0. The results for all the trials have been averaged over 25 runs of the algorithm. Dimensions 5 and 6 were run for a 100 generations while dimensions 7-9 were run for 500 generations. Since initial testing suggested that there was no incremental benefit to having more than 50 ants and that 25 ants also produced comparable results in much less time, 25 ants were used for all the problem sizes.

5.2 ACS Results

Table 5.1: Performance results for the ACS with (1) q_0 value from 0.5 to 0.9, with α constant at 0.1 and (2) α values from 0.1 to 0.5, with q_0 constant at 0.5. * denotes the length of the worst snake found, ** denotes the average and *** denotes the best length found. The best results in each of these categories is bolded for all dimensions, except dimension 5.

d	q_0=0.5	q_0=0.7	q_0=0.9	α=0.1	α=0.3	α=0.5
5	13*	13	13	13	13	12
	13**	13	13	13	13	12
	13***	13	13	13	13	13
6	**24**	23	22	**24**	23	22
	24	**24.38**	24.19	24	24	23
	26	**26**	25	**26**	25	25
7	**43**	**43**	40	**43**	**43**	41
	44.26	44.53	42.46	44.26	**44.76**	44
	46	46	45	46	**47**	46
8	**90**	89	81	**90**	89	87
	94.23	93.54	87.23	**94.23**	**94.23**	94.03
	97	**97**	93	**97**	**97**	**97**
9	159	**162**	152	159	160	158
	166.23	**168.7**	164.26	166.23	167.03	167.13
	173	175	169	173	**180**	175

Table 5.1 shows the results for two sets of experiments. The first set of experiments involved changing the exploitation parameter (q_0) while the second set of experiments consisted of changing the trail evaporation (α). Seeding with a longest snake from the previous dimension was used in dimensions 8 and 9. As α is typically very low in the ACS the value α

= 0.1 [Dorigo and Gambardella, 1997] was chosen as a baseline for experiments with varying q_0. As Table 5.1 shows, the most successful overall value of q_0 was 0.5, closely followed by 0.7. There was a marked drop in performance for $\alpha = 0.9$. This was consistent with our expectation that the over exploitation of solutions in the ACS can cause stagnation when combined with a high q_0 value. It can be prevented by using lower values of q_0 (by lower we refer to values that are lower than those typically used in the ACS for the TSP. [Dorigo and Gambardella, 1997] use $q_0 = 0.9$). This effect worsens with increase in problem size. In Table 5.1 consider the differences between the lengths of best snakes found using varying q_0 (consider only best values from column 2). The difference between the best length and the length found using $q_0 = 0.9$ is 1 in dimension 6 and 7, 4 in dimension 8, and 6 in dimension 9.

The next phase of experiments, which was aimed at understanding the effect of changing α, used the optimized value of $q_0 = 0.5$ as a constant for all dimensions. The lower values of α (0.1 and 0.3) combined with $q_0 = 0.5$ work the best. There is such a deterioration in performance for $\alpha = 0.5$ that the algorithm failed to even find the longest snake in dimension 5. The exploitation of the best solutions is hampered with high trail evaporation. Since the evaporation is a fraction of the pheromone present on an edge the edges with higher pheromone are affected more by it. Hence the trail of best solutions is lost quicker with high evaporation. Of special interest is the fact that different values of α work for different dimensions. $\alpha = 0.1$ works the best for dimensions 6 and 8, while $\alpha = 0.3$ gave the best results for dimension 7. This dimension specific information is used to advantage when fine tuning MMAS_SIB to suit every dimension. Since the ACS is designed for rapid exploitation of promising solutions, it gained significantly in performance from the seeding scheme that was used. The ACS found the longest snakes in dimensions 5-7, matched the record in dimension 8 and came very close to the record in dimension 9.

5.3 MMAS Results

The MMAS does not use an exploitation parameter (q_0), therefore the evaporation parameter α and trail limits become the most important part of its functioning. Experiments were conducted with α while the trail limit function was held at a constant. The trail limit function refers to the relationship between the τ_{max} and τ_{min} values for the MMAS. The initialization of the τ_{max} and τ_{min} values which correspond to the upper and lower trail limit respectively depends on the problem. The tighter the bound, (i.e. the lesser the difference between the τ_{max} and τ_{min}) the lesser the worst case time estimate on finding the optimal solution [Stützle and Dorigo, 2002]. The tradeoff is that the tighter the bound the lesser is the number of distinct paths explored thereby reducing the chances of finding a very good solution. We chose the ratio $\tau_{min} = \tau_{max}/3n$ (which is inspired by [Stützle and Dorigo, 2002]), where n is the dimension. Experiments were conducted to ascertain the best α values for the MMAS in dimensions 5-9 and the results are presented in Table 5.2. Seeding was used in dimensions higher than 7.

Even though there is very little differentiation in the results, α values of 0.7 and 0.8 seemed to work better than 0.5, especially in dimensions 7 and 8. The higher values of $\alpha = 0.7$ and $\alpha = 0.8$ yielding better results is expected, because the MMAS is designed to have better performance with high trail evaporation (reasons for which are discussed in Chapter 4.8.2). The MMAS showed no change in worst performance for the different values of α. In fact there was no significant fluctuation in the average and best results either. MMAS found the longest snake in dimension 6 for every set of parameters, unlike the ACS. The overall results for the MMAS could not match the performance of the ACS in longest snakes found. However, it performed more consistently than the ACS. This is due to the fact that MMAS explores the search space more thoroughly than the ACS. Unfortunately, due to the immense size of the search space most of the snakes found are not the longest in that dimension. The MMAS withstood changes in parameters better than the ACS which showed immediate changes in performance. This quality of the MMAS combined with the

Table 5.2: Performance results for the MMAS with varying trail evaporation(α).* denotes the length of the worst snake found ** denotes the average and *** denotes the best length found. The best results in the average and best category are bolded for all dimensions, except dimension 5.

D	α=0.5	α=0.7	α=0.8
5	13*	13	13
	13**	13	13
	13***	13	13
6	24	24	24
	25.15	24.88	24.88
	26	**26**	**26**
7	41	41	41
	42.3	**42.57**	42.34
	43	44	**45**
8	89	89	89
	92	91.53	**92.15**
	94	**95**	**95**
9	155	155	**157**
	159.93	159.23	**160.73**
	167	**169**	165

good results found using ACS should give MMAS_SIB the robustness and improved search capability to tackle all dimensions.

5.4 MMAS_SIB Results

Since MMAS_SIB has been introduced in this paper there was no prior literature to guide the parameter tuning for MMAS_SIB. Hence the knowledge about parameters gained from testing done with the ACS and the MMAS was instrumental in developing MMAS_SIB for each dimension. The best parameters from both algorithms were used and different combinations of these were tried for dimensions 7-10. We illustrate this process using dimensions

7 and 9 as an example since we have not been able to find very good results for either of these using the ACS or the MMAS. We do not show similar results for dimension 8 because the ACS found a snake of length 97 (which is the current lower bound for the dimension). Table 5.3 shows the results for experiments to fine tune α and q_0 for MMAS_SIB in dimension 7. Since we expect the MMAS_SIB to work well on exploitation parameter values from ACS, q_0 values from 0.5 to 0.9 are used. Similarly α values from 0.5 to 0.8 which are characteristic of the MMAS are used. Seeding is used in dimensions higher than 7.

Table 5.3: Performance results for MMAS_SIB in dimension 7, with values of q_0 from 0.5 to 0.9 and α from 0.5 to 0.8. * denotes the length of the worst snake found, ** denotes the average and *** denotes the best length found. The best results in each of these categories is bolded.

	α=0.5	α=0.7	α=0.8
q_0=0.5	42*	41	**43**
	43.58**	43.3	**43.92**
	46* **	**46**	45
q_0=0.7	42	**43**	**43**
	43.96	**44.92**	44.38
	46	**47**	46
q_0=0.9	42	**44**	43
	44.8	**45**	44.73
	47	**47**	**47**

q_0 values of 0.7 and 0.9 combined with the α value of 0.7 proved to be the best performing parameters in dimension 7. The q_0 value of 0.7 and the α value of 0.7 were expected to do well as they had in the ACS and the MMAS respectively. An unexpected result of the interactions between the various parameters was the emergence of the α value of 0.9 as a good parameter. The MMAS_SIB for dimension 7 combined the length results of the ACS with the consistency of the MMAS. This is evident from the fact that the best lengths found for almost all the parameter combinations are 46 and 47. Even though the MMAS_SIB displayed the characteristics of both the ACS and the MMAS, it did not better their performance. A strategy of seeding the longest snake from dimension 6 had to be used, in order to find the longest snake in the dimension. This yielded the longest snake of length 50 for all the parameter combinations shown in Table 5.3.

Table 5.4: Performance results for MMAS_SIB in dimension 9, with values of q_0 from 0.5 to 0.9 and α from 0.5 to 0.8. * denotes the length of the worst snake found, ** denotes the average and *** denotes the best length found. The best results in each of these categories. is bolded.

	α=0.5	α=0.7	α=0.8
q_0=0.5	159*	161	**162**
	167.81**	166.182	167.27
	177***	174	173
q_0=0.7	**164**	160	162
	172.9	170.27	168
	179	178	175
q_0=0.9	166	164	162
	167	169.4	171.09
	169	174	**184**

Similar experiments were conducted in dimension 9, based on the best performing parameters for the ACS and the MMAS. α values of 0.7 and 0.8 were expected to do well, when combined with a q_0 value of 0.7. Table 5.4 shows that the q_0 value of 0.9 combined with an α value of 0.8 found the longest snake in dimension 9. The same occurrence was observed in dimension 7, where the q_0 value of 0.9 had the best performance when combined with good α parameters from the MMAS. In the testing done with the ACS we had estimated that the lower α values would do better due to the size of the search space. When combined with the exploration effort introduced by the trail limits from the MMAS, the ACS needs a higher q_0 value to perform well. A snake of length 184 is found with an α value of 0.8 and a q_0 value of 0.9, which is better than the longest snake found using the ACS. However, the shortest snake found for this combination of parameters is 162 which is not characteristic of the consistency of the MMAS. The complete opposite of this is observed for the q_0 value of

0.9 combined with the α value of 0.5. The worst, average and best snake found are within 3 edges of each other, but the overall performance is sort of lackluster. We can say that in MMAS_SIB for dimension 9 qualities of either the ACS or the MMAS dominate depending upon which parameter combinations are considered. To gain better control over the performance of MMAS_SIB it is necessary to understand how q_0 and α interact when implemented with trail limits. As for the goal of obtaining a better performance by combining the ACS and the MMAS, it was certainly achieved in dimension 9.

5.5 Comparative Results

Table 5.5: Overall best time and length results for dimensions 5-9. Times are denoted in milliseconds.

d	Best length	Best time	Algorithm
5	13	30	ACS
	13	30	MMAS
	13	30	MMAS_SIB
6	26	790	ACS
	26	1640	MMAS
	26	590	MMAS_SIB
7	50	16940	ACS
	50	13390	MMAS
	50	14010	MMAS_SIB
8	97	64290	ACS
	95	75270	MMAS
	97	83340	MMAS_SIB
9	180	225980	ACS
	169	209220	MMAS
	184	667300	MMAS_SIB

Table 5.5 presents the overall best results for all three algorithms in dimensions 5-9. Seeding was used in dimensions greater than 6. Dimension 5 was run for 10 generations, 6

for 100 and 7-9 for 500 generations each. The best times shown in the table are run times for the specific trial in which the best length was found. It might be worth mentioning that increase in times as dimension goes up, can be attributed to the size of the search space increasing as well as the number of generations. The parameters used in each of the algorithms were the best performing parameters for each dimension. We found the longest snakes in dimensions 5-7 and matched the world record in dimension 8. In dimension 9 the result obtained was 184, while the current lower bound rests at 186. The results for dimension 8 were found in 64290 milliseconds (which is approximately 64.2 seconds), while the results in dimension 9 took 667300 milliseconds (which is approximately 11 minutes). MMAS_SIB performed as well (dimension 5-8) or better (dimension 9) than the ACS and the MMAS, as shown in Table 5.5.

The performance of ACO on the SIB problem was very encouraging. The overall results clearly indicate that a search strategy for one dimension cannot be generalized for a higher dimension. Different qualities of the search algorithm might be needed in different dimensions to achieve successful results. After initial testing it was decided that there were some parameters of ACO that would not affect performance very much and hence were not tested. Different trail limit functions for the MMAS could be tried to see if improvements in length are found. Successful α values from the ACS might be tried for the MMAS_SIB, instead of using only the successful α values from the MMAS. Even if the results from the current versions of the algorithms are not bettered, these trials might shed some more light on the interplay between the various parameters of ACO, especially those in the newly built MMAS_SIB.

Chapter 6

Conclusion and Future Work

In this thesis we have presented an approach to the SIB problem using Ant Colony Optimization. As part of our research we explored two existing ACO algorithms: the ACS and the MMAS. We also proposed a new algorithm, MMAS_SIB, which combines the best features of the two. Our results show that MMAS_SIB is an effective searching algorithm across all dimensions of the SIB problem. The performance of ACO for the SIB problem was very encouraging. The longest known snakes in dimension 5-8 were matched. In dimension 9 the MMAS_SIB found a snake of length 184 which was two edges shy of the world record of 186. The MMAS_SIB performs very well on the time criteria, but leaves room for improvement with consistency and predictability.

The ACO framework offered certain advantages over population based evolutionary methods, when it came to path finding problems like the SIB problem. This helped the algorithm achieve the following computational efficiencies:

1. Preserving the validity of the solution: In case of the SIB problem the complete solution to the problem is a sequence of nodes, which together form the snake. The validity and length of the snake depends upon the specific node order. Genetic algorithms (which have been a popular approach to the SIB problem) work on the assumption that preserving high quality node sequences will lead to a better solution. However this is hard to do after the crossover and mutation operators in the Genetic Algorithm (GA) have been applied. Both, crossover and mutation, make chnages to the node sequence, with the hope of finding a better solution. Changing a node within a snake might violate the validity of the snake. Specialized operators like XOR mutation or the

enhanced edge recombination crossover used by [Bitterman, 2004] can be used to try and maintain good node sequences. Another approach is to avoid crossover altogether [Casella, 2005]. ACO provides a better alternative. The sequence of promising nodes is retained as the strength of pheromone on their respective edges instead of actual node sequences in the solution, thereby circumventing the issue altogether. This also means that ACO always produces only valid solutions.

2. Complexity of evaluating solution quality: The fitness evaluation of an individual in most population based evolutionary methods is based on finding the longest snake within a node sequence of a certain length [Potter et al., 1994; Bitterman, 2004; Casella, 2005]. This type of fitness evaluation can be very computationally expensive and have time complexity to the order of $O(n^2)$, where n is the length of the sequence to be evaluated. In the ACO, each ant maintains the cost of the path from the first node to the present node. When a new node is added, the cost of this new node is added to the total path cost [1] without recalculating the cost so far. This gives the evaluation function a time complexity of $O(1)$.

3. Extending snakes: In population based searches for the SIB problem there might be the need to use a local search procedure to make sure that the longest snake found in a sequence is not extendable past the last node or before the first node. In short, there is no inherent guarantee of finding maximal snakes, i.e. snakes that cannot be extended any further in either direction. Single point searches (searches which have one candidate solution and try to improve it) for the SIB problem [Brown, 2005] are better suited to finding maximal snakes, because they continually improve the solution, until no further improvement is possible. However, single point searches are incapable of generating and evaluating several solutions simultaneously. ACO combines the advantages of both

[1]Path cost is a measure of evaluating a solution. It can be any function depending upon the problem representation and implementation.

these types of searches by developing several solutions in parallel and continuing the search until each solution can not be extended any further.

4. Scaling up for problem size: To scale up ACO to bigger instances of a problem does not require the number of ants to be increased drastically, but only the number of iterations. In fact most implementations use 10-25 [Dorigo and Gamberdella, 1997; Stützle and Hoos, 2000] ants as compared to several thousand individuals used in other population based searches ([Casella, 2005] used 10,000 individuals). This means that run times are not multiplied on the scale of individuals times iterations, but stay proportional to the number of iterations. Other implementation like the memory adaptive iterated local search [Brown, 2005] depends on the memory available to implement them. The memory requirements of ACO are very reasonable and seldom exceed system resources.

So, in theory ACO seems to enjoy some of the best features of both, population based and single point searches for the SIB problem.

In the literature, ACO has been applied to problems that have been represented as least cost, shortest path, minimization problems [Birattari, 2002]. Our problem representation makes the longest path the most optimal. To successfully adapt ACO to a maximization problem was an interesting challenge. The reason that ACO lends itself to minimization problems is due to the nature of ant foraging. The shorter paths gain an advantage because it takes less time for ants to travel on them, resulting in more trips and an increased amount of pheromone. In a program time is not continuous but is measured in discrete intervals that are decided by the flow of the program. This means that while in nature ants are constantly making trips back and forth, bringing food to their nest, in ACO all ants complete only one trip in each program cycle.[2] After the completion of each program cycle, the effect of more trips being made on the shorter path is simulated by depositing more pheromone on the

[2]By program cycle we refer to the process of each ant sequentially building a complete solution and then resetting their path to start building a new solution. This corresponds to real ants finding a food source and bringing food back to the nest.

shorter path. This led us to believe that extending ACO to a maximization problem should yield fruitful results. We tested this hypothesis successfully through our work.

The most important lesson learnt about ACO through this research, is that it is very important to understand the shape of the search space when designing an ACO tailored to a specific problem. The search space for the SIB problem has not been very well defined by work done so far. However, from the way the search effort had to be customized for every dimension in the SIB problem, it is apparent that the size and shape of the search space is different in every dimension. Another disadvantage ACO faces in the SIB domain is that heuristic information for look ahead (look ahead means deciding how promising a solution component is before the entire solution is constructed) is not easy to acquire. Due to these factors a detailed understanding of the SIB problem would definitely help the performance of ACO.

Future work could include improvements in the ACO for the SIB, by studying how combining the parameters from the ACS and MMAS (like we did with MMAS_SIB) affects the properties of the algorithm. Detailed analyses of the MMAS_SIB can be conducted using criteria like pheromone levels, edge desirability [Dorigo and Gambardella, 1997] and the number of distinct solutions explored. We conclude that the potential of this new algorithm can be utilized better only after developing a deeper understanding of it.

References

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

L.E Adelson, R. Alter, and T.B. Curtz, “Long snakes and a characterization of maximal snakes on the d-cube”, *Proceedings, 4th SouthEastern Conference on Combinatorics, Graph Theory and Computing, Congr*, 8:111-124, 1973.

R. Beckers, J. L Deneubourg and S. Goss, “Trails and U-Turns in the Selection of a Path by the Ant Lasius Niger”, *J. Theor. Biol.*, 159:397-415, 1992.

M. Birattari, G. Di Caro, and M. Dorigo, “Toward the formal foundation of Ant Programming”, *Third International workshop on Ant Algorithms, ANTS 2002*, Brussels, Belgium, 188-201, 2002.

E. Bonabeau., M. Dorigo, and T. Theraulaz, *From Natural to Artificial Swarm Intelligence*, Oxford University Press, New York, 1999.

D. Bitterman, *“New Lower Bounds for the Snake-In-The-Box Problem: A Prolog Genetic Algorithm and Heuristic Search Approach”*, Master’s thesis, University of Georgia, 2004.

B. Bullnheimer, R. F. Hartl and C. Strauss, “An improved ant system algorithm for the vehicle routing problem”, *Annals of Operations Research*, 89:319-328, 1999.

W. Brown, *“An Iterated Local Search with Adaptive Memory applied to the Snake in the Box problem”*, Master’s thesis, University of Georgia, 2005.

D. Casella and W. D. Potter, “New Lower Bounds for the Snake-In-The-Box Problem: Using Evolutionary Techniques to Hunt for Snakes”, *The 18th International FLAIRS Conference*, Clearwater Beach, Florida, 264-269, 2005.

A. Colorni, M. Dorigo, V. Maniezzo, and M. Trubian, “Ant System for job-shop scheduling”, *JORBEL-Belgian Journal of Operations Research, Statistics and Computer Science*, 34(1):39-53, 1994.

D. Merkle, M. Middendorf, H. Schmeck, “Ant colony optimization for resource-constrained project scheduling”, *IEEE Transactions on Evolutionary Computation*, 6(4):333-346, 2002

J. L. Deneubourg, S. Aron, S. Gross, and J. M. Pasteels, “The Self-Organizing Exploratory Pattern of the Argentine Ant”, *J. Insect Behavior* , 3:159-168, 1990.

M. Dorigo, V. Maniezzo, and A. Colorni, “The ant system: optimization by a colony of cooperating agents”, *IEEE Transactions on Systems, Man, and Cybernetics*, Part B, 26(2): 29-41, 1996.

M. Dorigo and L. M. Gamberdella, “Ant Colony System: A Cooperative Learning Approach to the Travelling Salesman Problem”, *IEEE Transactions on Evolutionary Computation*, 1(1):53-66, 1997

M. Dorigo, G. Di Caro, and L. M. Gambardella, “Ant Algorithms for Discrete Optimization”, *Artificial Life*, 5(3): 137-172, 1999.

M. Dorigo and G. Di Caro, “The Ant Colony Optimization meta-heuristic”, D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, McGraw Hill, London, UK, 11-32, 1999.

M. Dorigo and T. Stützle, *“Ant Colony Optimization”*, The MIT Press Cambridge, Massachusetts, 2004.

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

W. J. Gutjahr. "A graph-based ant system and its convergence", *Future Generation Computer Systems*, 16(8):873-888, 2000.

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, page numbers-2000.

V. Klee, "A method for constructing circuit codes", *J. Assoc. Comput. Mach.*, 14:520-538, 1967.

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.

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", *7th International Conference On Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, Austin TX, 421-426, 1994.

D. S. Rajan, and A. M. Shende, "Maximal and reversible snakes in hypercubes", *24th Annual Austalasian Conference on Combinatorial Mathematics and Combinatorial Computing*, Northern Territory University, Darwin, Australia, http://citeseer.csail.mit.edu/rajan99m
1999.

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

T. Stützle and H. Hoos, “Improvements on the ant system, introducing the MAX-MIN ant system”, *Proceedings of ICANNGA97 - Third International Conference on Artificial Neural Networks and Genetic Algorithms*, 245-249, 1997.

T. Stützle and H. H. Hoos, “MIN-MAX Ant System.” *Future Generation Computer Systems*, 16(8):889-914, 2000.

T. Stützle and M. Dorigo, “A Short Convergence Proof for a Class of ACO Algorithms”, *IEEE Transactions on Evolutionary Computation*, 6(4):358-365, 2002.

D. Tuohy, Personal Communication, June 2005.

Appendix

Program flow chart and code for ACOs

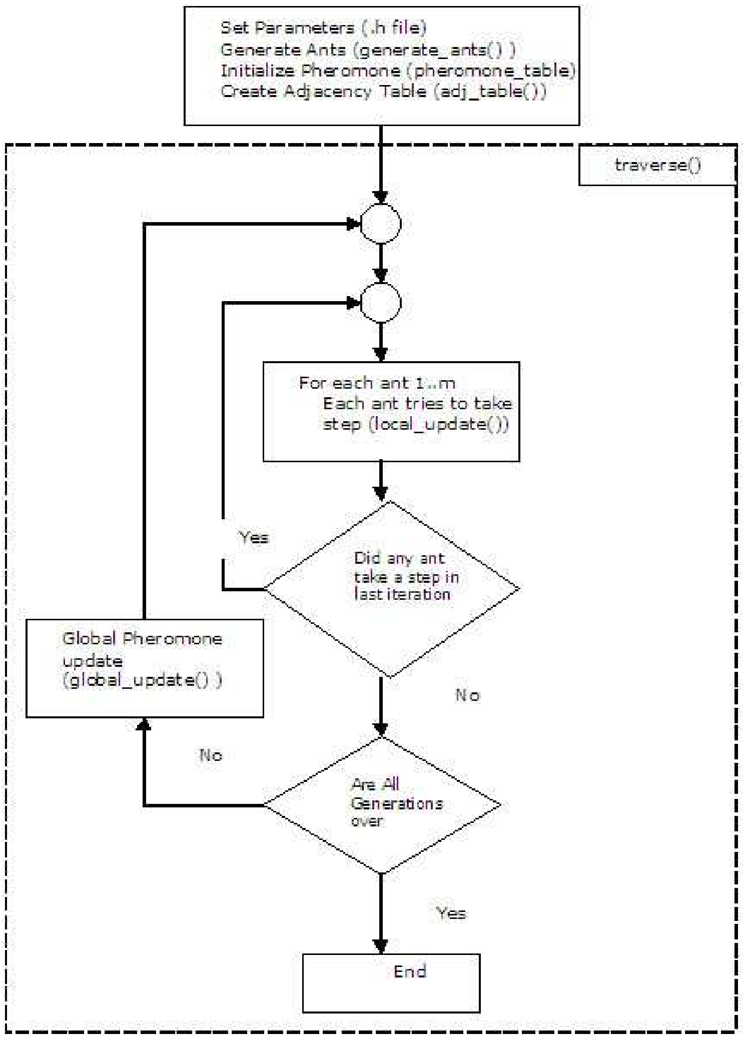
Set Parameters (.h file)
Generate Ants (generate_ants())
Initialize Pheromone (pheromone_table)
Create Adjacency Table (adj_table())
traverse()
For each ant 1..m
Each ant tries to take step (local_update())
Yes
Did any ant take a step in last iteration
No
Global Pheromone update (global_update())
No
Are All Generations over
Yes
End

Figure A.1: Program flow chart for ACO algorithms

```
/*****************************************************
**This is a sample .h file for all three algorithms.
**Variations for each algorithm are marked as comments for each parameter
****************************************************/


#include<iostream>
using namespace std;
#include "ulist.h"
#include<math.h>

const double ALPHA = 0.7;          //[0,1]importance of pheromone
const double BETA = 0.0;                //[0,1]importance of heuristic
const int DIMENTION = 10;
const int NUM_ANTS = 25;
const double GLOBAL_STEP = 0.5;         //Exploitation  probability
                       //GLOBAL_STEP is always 0 for MMAS, MMAS_SIB
const int MAX_GEN = 1000;
class swarm
{
    class Ant           //ant class
    {
        public:
        UList path;             //Holds ant path
        int length;
        int ant_id;             //you initialize with this
        int generation;         //increase this with each generation
        double fitness;         //fitness=available nodes/length
        int overlap;            //This is a count of overlapped nodes
        Ant(int num)
        {
            ant_id = num;
            generation = 1;             //when you create generation is 1
            path.Insert(0);
            length=1;
            overlap=0;
            double nodes = static_cast<double>(pow(2.0,DIMENTION));
            fitness=(double)(DIMENTION+1);          //No of available nodes
            node=new int[(int)(nodes)];

        }

    };
int **x;                            //adjacency table
int canon[NUM_ANTS];                                //canonical bit for each ant
public:
 double PHER_MAX;
 double PHER_MIN;

   Ant *MAX_SNAKE;                      //Iteration best snake found
   Ant *GLOBAL_MAX;                     //longest snake found
   UList MAX_SNAKES_HIST[NUM_MAX_SNAKES];
   double **pher_table;                 //Pheromone table
   double **initial_pher;               //Initial pheromone at each node
   int paths[NUM_ANTS][400];            // table that holds paths of all ants
                      //Adjust size for every dimension
   time_t seed;
   Ant *baby[NUM_ANTS];
   int flip(int a,int b);
   void adj_table();
   void pheromone_table();              //initializes pheromone table
   void generate_ants();                //generates NUM_ANTS with first node 0
   int next(int id);                    //takes ant_id and returns next node
   void traverse();                     //control method for whole program
   int valid(int node, int id);        //returns validity of node
   void kill(int id);              //resets all ant paths
   int anyAntAlive();                   //does any ant has feasible nodes left
   int scheck(UList& id);
   void take_step(int id, int next_node);       //adds node to snake
   void transition(UList& id);                  // transition sequence of snake
   int iter_best();                             //Returns iteration best snake
   void global_update(int gen);                // Global update rule
   void local_update(int id, int node);     //Local update rule
   void reset_fit(int id);
   double heuristic(int id, int next_node);
};

/*************************************************************************
**This creates the adjacency table for hypercube of dimension:DIMENTION.
**This method is common to all algorithms.
*************************************************************************/

void swarm::adj_table()
{
   seed = time(NULL);
   cout<<"seed is: "<<seed<<endl;
   srand(seed);
   int nodes = static_cast<int>(pow(2.0,DIMENTION));
   cout<< "power result is "<<nodes<<endl;
   int a,i,j,b,c;
   x =   new (int*)[nodes];
   for(i=0;i < nodes;i++)
    {
        x[i] = new (int)[DIMENTION+1];
        x[i][0] = i;

        for(j=1;j< DIMENTION+1;j++)
        {
            a = flip(i,j);  //generate neighbors
            x[i][j]= a;       //put them in columns of row i
        }
    }
}


/**********************************************
** Takes dimention and intializes all places in
** pheromone table to a random pher value
** In ACS use the length of longest snake
*********************************************/
void swarm::pheromone_table()
{
    double random=10000.0;
    int i,j;
    int nodes = static_cast<int>(pow(2.0,DIMENTION));
    pher_table = new (double*)[nodes];
    initial_pher = new (double*)[nodes];

    for(i=0;i<nodes;i++)    //Need to have only adj nodes
    {
    pher_table[i] = new (double)[nodes];
    initial_pher[i] = new (double)[nodes];
    for(j=0;j<nodes;j++)
    {
/** UNCOMMENT THIS FOR ACS, COMMENTED FOR MMAS and MMAS_SIB
        if(DIMENTION>9||DIMENTION<6)
                random= (double)(rand()%(DIMENTION+1));
        if(DIMENTION==6)
            random=26;
        if(DIMENTION==7)
            random=50;
        if(DIMENTION==8)
            random=97;
        if(DIMENTION==9)
            random=186;
*/
        pher_table[i][j]=random;
        initial_pher[i][j]=random;
    }
    }
}
```

```
/***************************************************
**generates ants needed for population based on NUM_ANTS
*****************************************************/

void swarm::generate_ants()
{   int i,j,n;
    int nodes = static_cast<int>(pow(2.0,DIMENTION));
    for(i=0;i < NUM_ANTS;i++)
    {
    baby[i] = new Ant(i);
     for(j=0;j<nodes;j++)
        {

            baby[i]->node[j]=0;
        }

        for(j=1;j<DIMENTION+1;j++)
        {
            n=x[0][j];
            baby[i]->node[n]=1;
        }
        baby[i]->node[0]=2;
    }
    MAX_SNAKE=new Ant(10);
    GLOBAL_MAX=new Ant(10);

//The following loop seeds all paths with the given array
int koc12[]=

{0, 1, 3, 7, 15, 31, 63, 55, 51, 49, 48, 56, 60, 28, 20, 22, 18, 26, 10, 42,
46, 38, 36, 37, 45, 41, 105, 104, 72, 88, 80, 81, 83, 91, 123, 122, 126,
118, 116, 117, 125, 93, 77, 69, 68, 70, 66, 98, 99, 103, 111};


 int kl=98;
  for(int b=0;b<NUM_ANTS;b++)
    {
        for(int d=0;d<kl;d++)
        {
        paths[b][d]=koc12[d];
       take_step(b,koc12[d]);

        }
     baby[b]->length=kl;
        for(int f=kl;f<400;f++)
        {
```

```
        paths[b][f]=0;
        }

    }
/* UNCOMMENT THIS PART IF NOT USING A SEED


  for(int b=0;b<NUM_ANTS;b++)
    {
    for(int c=0;c<400;c++)
    {
        paths[b][c]=0;
    }

    }
*/
}




/***********************************************************
**Each ant takes one step, adds node to path, and keeps going until it
**has no more nodes. This is repeated until all generations are over.
*********************************************************/

void swarm::traverse()
{
    Ant *l;
    UList list;
    int  next_node,length1=0;
    int last_max_gen=0;
    int step_counter=0;
    int iter_max=0;
    int i=0;
    int j=0;
    l= baby[0];
    while(l->generation < MAX_GEN+1 )
        {

        for(i=0;i<NUM_ANTS;i++ )
        {
            l= baby[i];
        next_node = next(i);        //find next node, return 0 if no node left

            if(next_node !=0)
            {
            step_counter++;     //total number of valid steps taken
            take_step(i,next_node);   // inserts next node in ants path


        }//if next_node not 0
        else
        {
            list.Insert(l->ant_id);
            if(list.length()==NUM_ANTS)
            {
            for(int gb=0;gb<NUM_ANTS;gb++)
            {
            l=baby[gb];
            if(l->length > GLOBAL_MAX->path.length())
                {
             //replace all stats of global best snake
                        l->path.CopyList(GLOBAL_MAX->path);
                        GLOBAL_MAX->ant_id = l->ant_id;
                        GLOBAL_MAX->overlap= l->overlap;
                        GLOBAL_MAX->fitness = l->fitness;
                        GLOBAL_MAX->length = l->length;
                        GLOBAL_MAX->generation=l->generation;
                        last_max_gen=l->generation;

            /***************CHANGE MIN AND MAX WHEN GLOBAL BEST FOUND*******/
            PHER_MAX=GLOBAL_MAX->length;
            PHER_MIN=PHER_MAX/(double)(DIMENTION*2);
                       /**COMMENT LAST TWO LINES FOR ACS **/
            }
          }

            iter_max=iter_best();   //finds the best snake in last gen
            //replace all stats of iteration best snake
            baby[iter_max]->path.CopyList(MAX_SNAKE->path);
                        MAX_SNAKE->ant_id =baby[iter_max]->ant_id;
                        MAX_SNAKE->length = baby[iter_max]->length;
                        MAX_SNAKE->overlap= baby[iter_max]->overlap;
                        MAX_SNAKE->fitness = baby[iter_max]->fitness;
                        global_update(l->generation);


                for( j=0;j<NUM_ANTS;j++)
                {
                    kill(j);   //reset all ants stats
                }
                    list.DeleteList();
```

```
                break;
        }
        }

       }//for iterate through all ants

      if(i>=NUM_ANTS-1)
      {
          i=0;
      }

    }//while true
}



//***************************************************/
//Valid returns 0 if node is valid ie: not visited and
//not adjacent to something which has been visited for not valid
//***************************************************/
int swarm::valid(int node, int id)
{
    int isValid = 1;
    int code;
    int i,neigh,length,previous;
    length=baby[id]->path.length();
    previous=paths[id][length-1];   //the node which was last added to the path
                    //becasue it will always be adj to this one
    if(baby[id]->path.Find(node)==1)
    {
      isValid = 0;
      code = 0;
    }
    else
    {
        for(i=1;i<DIMENTION+1;i++)
        {
       neigh= x[node][i];
       if (baby[id]->path.Find(neigh)==1 && neigh != previous)
           {
          isValid = 0;
              code =  1;
              break;
           }
        }
    }
    return isValid;
```

```
}


/****************************************************
**returns next node, with highest pheromone uses GLOBAL_STEP in ACS
** But not in MMAS and MMAS_SIB
*****************************************************/
int swarm::next(int id)
{
    double random=0.0;
    double node_pher=0.0;
    double cumulative=0.0;
    double r=0.0;
    double r1=0.0;

    while(r==0.0)
    {
    r=(double)rand();
    }
    while(r1==0.0)
    {
    r1=(double)rand();
    }

    if (r1 > r )
      random = r/r1;
    else
      random = r1/r;
    double max_pher = 0.0;
    double heur, total_pher = 0.0;
    int i,node;
    int  count=0;
    int  counter=0;
    int max_node=0;
    int size=0;
    int line=0;
    int l = baby[id]->path.length();
    int overlap=baby[id]->overlap;
    int row = paths[id][l-1];   //find the last visited node
    double fit=baby[id]->fitness;
    double table[DIMENTION][4]={0};
        for(i=1;i<DIMENTION+1;i++)
        {
        node= x[row][i];    //get the neighbor node from adjacency table

        if (valid(node,id)==1)  //returns validity of node
        {
```

```
   table[line][0]=node;         //store it as feasible node
   table[line][3]=i;            //store bit position flipped to get node
   size++;              //keep track of array size
   heur=heuristic(id,node);
   node_pher = pher_table[row][node]*(pow(1,1/fit));
   table[line][1]=node_pher;
   total_pher = total_pher+node_pher;
   line++;
}//if valid
}//for DIMENTION
if(size!=0 && total_pher==0)
{
    random = rand() % size;
    int rint= (int)(random);
    max_node = static_cast<int>(table[rint][0]);
}
// IF EXPLOITATION, GLOBAL_STEP=0 in MMAS,MMAS_SIB
if(random < GLOBAL_STEP && size!=0)
{
while(counter<size)
{
    if(table[counter][1] > max_pher)        //find if max is more
    {
    max_pher = table[counter][1];
    max_node =  static_cast<int>(table[counter][0]);
 }//if
counter++;
}//while
}//if random<global EXPLORATION
else
{

if(size!=0 && total_pher!=0)
{
while(count<size)               //end of array
{
        cumulative=cumulative+table[count][1];
       table[count][2]=cumulative/total_pher;
    count++;
}
r=r1=0.0;       //set these back to 0.0
while(r==0.0)
        {
        r=(double)rand();
        }
    while(r1==0.0)
        {
                r1=(double)rand();
                }
        if (r1 > r )
            random = r/r1;
            else
            random = r1/r;
        for(int k=0;k<size;k++)
        {
                if(random<table[k][2])
                {
            max_node= static_cast<int>(table[k][0]);
            break;
                }
        }
        }//if size not 0
    }//else exploration
    return max_node;
}



/********************************************************
**This is a local pheromone update which takes one node and ant_id.
**This function has to be called after the next node is added.
**This function is used only in the ACS.
*******************************************************/

void swarm::local_update(int id, int node)
{
    double pher, change;
    int *array=baby[id]->path.returnArray();
    int length=baby[id]->path.length();
    pher=pher_table[array[length-2]][array[length-1]];
    //change is set to intial pher
    change= initial_pher[array[length-2]][array[length-1]];
    pher_table[array[length-2]][array[length-1]]=
              ((1-ALPHA)*pher)+(ALPHA*PHER_MAX);
    if(pher_table[array[length-2]][array[length-1]]<0)
    pher_table[array[length-2]][array[length-1]]=0;

}



/********************************************************
**Global  update includes adding pheromone to the iteration best
**and evaporation for other edges
```

```
**The following method is for the MMAS and MMAS_SIB
**For the ACS always use only the global best snake to deposit pheromone
***************************************************/
void swarm::global_update(int gen)
{
    int j,i,node,next,length;
    int *l;
    cout<<"***********GLOABL UPDATE*************gen is"<<gen<<endl;
    int nodes = static_cast<int>(pow(2.0,DIMENTION));
    for(i=0;i<nodes;i++)
    {
        for(j=0;j<nodes;j++)
        {
        pher_table[i][j]=(1-ALPHA)*pher_table[i][j];
        if(pher_table[i][j]<PHER_MIN)
        pher_table[i][j]=PHER_MIN;
        if(pher_table[i][j]>PHER_MAX)
                pher_table[i][j]=PHER_MAX;
        }
    }
        l=MAX_SNAKE->path.returnArray();
        length = MAX_SNAKE->path.length();

        if( gen>=25 && gen<75 && gen%5==0)
        {
            l=GLOBAL_MAX->path.returnArray();
            length = GLOBAL_MAX->path.length();


        }
         if(gen>=75 && gen<125 && gen%3==0)
        {
            l=GLOBAL_MAX->path.returnArray();
            length = GLOBAL_MAX->path.length();


        }
         if(gen>=125 && gen<250 && gen%2==0)
        {
            l=GLOBAL_MAX->path.returnArray();
            length = GLOBAL_MAX->path.length();


        }
        else
        {
            if(gen>=250)
```

```
                {
                l=GLOBAL_MAX->path.returnArray();
                length = GLOBAL_MAX->path.length();
               updated GB_SNAKE"<<endl;
                }
            }
            for(i=0;i<length-1;i++)
            {
                node=l[i];
                next=l[i+1];
                pher_table[node][next]= pher_table[node][next]+ (0.1*length);

                if(pher_table[node][next]>PHER_MAX)
                    pher_table[node][next]=PHER_MAX;
            }
    }
```