The American University in Cairo

School of Sciences and Engineering

An Adaptive Hybrid Genetic-Annealing Approach for Solving the MAP Problem on Belief Networks

A Thesis Submitted to

The Department of Computer Science

In partial fulfillment of the requirements for

The degree of Master of Science

By

Manar Ibrahim Hosny

B.Sc. in Computer Science

Under the supervision of

Dr. Ashraf Abdelbar

May 2000

TABLE OF CONTENTS

LIST OF TABLES

Chapter 1

1 Introduction

1.1 Algorithm and Complexity Analysis

In computer science, researchers are often faced with the problem of comparing two algorithms in terms of their efficiency and in terms of speed and resource consumption. The field of algorithm analysis helps scientists to perform this task by providing an estimate of the number of operations performed by the algorithm, irrespective of the particular implementation or input used.

Exact analysis of algorithm complexity is usually hard to achieve. As a result, approximation is usually the alternative approach used. The *O* notation is usually used to provide an upper bound of the complexity of an algorithm. We say that an algorithm is of *O(n)* (Order *n*), where *n* is the size of the problem, if the total number of steps carried out by the algorithm T(*n*) is at most a constant times *n*, with the possible exception of a few small values of *n*.

T(n) is O(f(n)) if $T(n) \leq kf(n)$ for some k, for all $n>n_0$

Where k is some constant and n_0 is a small possible value of n

The *O* notation is a measure of asymptotic analysis. Using it, we can be certain that as *n* approaches infinity, an algorithm of *O(n)* is better than an algorithm of $O(n^2)$.

In addition to analyzing the efficiency of a particular algorithm, we sometimes need to know whether there exist better algorithms for solving a particular problem. The field of complexity analysis analyzes problems rather than algorithms.

Two major classes of problems can be identified:

1. Problems that can be solved in polynomial time.
2. Problems that cannot be solved in polynomial time, irrespective of the type of the algorithm used.

The first class of problems is called *P*, polynomial time problems. It contains problem with running times like $O(n)$, $O(\log n)$ and $O(n^{1000})$. They are relatively easy problems.

Another important class of problems is *NP*, non-deterministic polynomial time problems. This class includes problem for which there exists an algorithm that can guess a solution and verify whether the guessed solution is correct or not in polynomial time. If we have an unbounded number of processors that each can be used to guess and verify a solution to this problem in parallel, the problem can be solved in polynomial time.

One of the big open questions in computer science is whether the class *P* is equivalent to the class *NP*. Most scientists believe that they are not equivalent. This, however, has never been proven.

Researchers also distinguish a sub class of *NP*, called the *NP-complete* class. In a sense, this class include the hardest problems in computer science, and is characterized by the fact that either all problems that are *NP-complete* are in *P*, or not are in *P*. Many *NP-complete* problems require arrangement of discrete objects, like the traveling salesman problem (TSP), and the job shop scheduling problem. These problems belong to combinatorial optimization problems.

An optimization problem for which the associated decision problem is *NP-complete* is called an *NP-hard* problem. For example, if the problem is a cost

minimization problem, such that it is required to find a solution with the minimum possible cost, the associated decision problem would be formulated as: “ is there a solution to the problem whose cost is ≤ *B*, where *B* is a positive real number?”

Solving combinatorial optimization problems has been a challenge for many researchers in computer science. Exact methods used to solve regular problems cannot be used to solve combinatorial optimization problems given current resources. The natural alternative would be to use approximate methods that give good rather than optimal solution to the problem in a reasonable amount of time.

1.2 Heuristic Search Methods

Many heuristic search methods have been designed and used in solving combinatorial and NP-complete problems (Haralick & Elliot, 1980; Pearl, 1984; Stone & Stone, 1986; Glover, 1989). Genetic algorithms and simulated annealing are two successful methods in this area.

Genetic algorithms, referred to thereafter by GAs, are search methods based on the principles of natural selection and survival of the fittest. The algorithm operates on a population of individuals representing solutions to the required problem. Each generation, a new set of solutions is generated using bits and pieces of good solutions in the previous generation, using an operator called crossover. Occasionally new parts are tried to allow for better exploration of the search space. This is performed using the GA mutation operator.

GAs have been developed by John Holland in the 1970s (Holland, 1975). The main idea was the attempt to develop a system that simulates nature in its robustness and adaptation; a system that operates consistently well on a variety of problems and can

survive many different environments. These algorithms are computationally simple yet powerful in their ability to improve and obtain good solutions. They are also less sensitive than other search methods to assumptions made about the search space like continuity, existence of derivatives, number of local optima...etc.

Simulated annealing (SA) is another well-known heuristic search method that has been used successfully in solving many combinatorial optimization problems (Chams et al., 1987; Connolly, 1988; Wright, 1989). The term is adopted from the annealing of solids where we try to minimize the energy of the system using slow cooling until the atoms reach a stable state. The slow cooling technique allows atoms of the metal to line themselves up and to form a regular crystalline structure that has high density and low energy. The initial temperature and the rate at which the temperature is reduced are called the annealing schedule.

In solving a combinatorial optimization problem we start with a certain feasible solution to the problem. We then try to optimize this solution using a method analogous to the annealing of solids. A neighbor of this solution is generated using an appropriate method, and the cost (or the fitness) of the new solution is calculated. If the new solution is better than the current solution in terms of reducing cost (or increasing fitness) the new solution is accepted. However, if the new solution is not better than the current solution, it is accepted with a certain probability, which decreases exponentially with the badness of the move. Thus, the procedure is less likely to get stuck in a local optimum since bad moves still have a chance of being accepted.

GAs are capable of wide exploration of the search space, since they operate on a population of individuals and combine good solution using a recombination operator. SA

on the other hand operates by producing several small moves on one solution, and thus is capable of fine tuning a good solution obtained in search of a better solution.

Combining GAs and SA is an attractive area of research, since the hybridization has the potential of achieving the benefits of both techniques. SA in this context can be used as part of the genetic engine to improve the solutions obtained by a GA.

1.3 Goal and Motivation

In the current research we try to augment a standard GA with SA which acts as a directed or intelligent mutation operator. Unlike previous research, the SA used in this technique is adaptive in the sense that its parameters evolve and optimize themselves according to the requirements of the search process. Using adaptation is intended to make simulated annealing parameter adjustment an easy and automatic task. In addition, adaptation should help guide the search towards optimal solutions, and improve the quality of the search.

The algorithm is tested on an important problem in the field of inference and acting under uncertainty, which is the MAP (maximum a-posteriori) assignment problem, also known as the Most Probable Explanation problem (MPE) on Bayesian Belief networks. This problem is *NP-hard* and it has many applications in the fields of medical diagnosis, computer vision and natural language understanding.

1.4 Organization

The rest of this thesis is organized as follows:

Chapters 2, 3, and 4 provide a review of background information about the main topics used in the research. These are Genetic Algorithms, Simulated Annealing, and Bayesian Belief Networks, respectively.

Chapter 5 is a survey of the most important previous research in the literature. Chapter 6 explains research goal and motivation, and compares the technique with other hybridization techniques. Chapter 7 provides implementation details, and chapter 8 is a discussion of the results obtained from experimentation. Finally, chapter 9 concludes with a direction for future research. Appendix A is a listing of the source code, and Appendix B contains the topology of some BBN networks used in testing.

Chapter 2

2 Genetic Algorithms

2.1 Overview

Genetic algorithms are intelligent search methods that have been used successfully in solving many difficult problems such as combinatorial optimization problems. The principles of GAs were founded by John Holland (1975). The theme of a GA is to simulate the processes of biological evolution, natural selection and survival of the fittest in biological organisms. In nature individuals compete for the resources of the environment, they also compete in selecting mates for reproduction. Individuals who are better or fitter in terms of their genetic traits survive to breed and produce offspring. Their offspring carries their parents' basic genetic material, which lead to their survival and breeding. Over many generations this favorable genetic material propagates to an increasing number of individuals. The combination of good characteristics from different ancestors can some times produce "super fit" offspring who out perform their parents. In this way species evolve to become more and more suited to their environment.

GAs operate in exactly the same manner. They work on a population of individuals representing possible solutions to a given problem. Each individual is usually represented by a string of bits analogous to chromosomes and genes, i.e. the parameters of the problem are the genes and are joined together in a solution chromosome. A fitness value is assigned to each individual in order to judge its ability to survive and breed. The highly fit individuals are given a chance to breed by being selected for reproduction, i.e. the selection process usually favors the more fit individuals. Good individuals may be

selected several times in a generation, poor ones may not be selected at all. By favoring the “most fit” individuals, favorable characteristics spread throughout the population over several generations, and the most promising areas of the search space are explored. Finally, the population should converge to an optimal or near optimal solution. Convergence means that the population evolves toward increasing uniformity, and the average fitness of the population will be very close to the highest fitness.

During the reproduction phase of a GA, two individuals breed by combining their genes in an operation called crossover. Not all selected pairs undergo crossover; A random choice is applied where the likelihood of crossover is some given probability. If crossover is not performed, offspring are produced simply by duplicating their parents. Crossover allows the basic genetic material of the parents to pass to their children who form the next generation. Another operation that is performed by GAs is mutation. Mutation is applied to each child generated from crossover. With a certain small probability each gene may be altered. Thus, Crossover allows a rapid exploration of the search space by producing large jumps, While mutation allows a small amount of random search and helps ensure that no point in the search space have a zero probability of being explored.

In summary, the basic principles of GAs are: **Coding** or representing the problem as a set of parameters (genes), assigning a **Fitness** or objective value indicating the utility or the goodness of the chromosome, and finally **Reproduction** by performing crossover and mutation over selected chromosomes.

The basic outline of a GA is as follows:

1. 1.Initialize and encode a random population of solutions called chromosomes.
2. Decode and evaluate the fitness or the objective of each chromosome.
3. Create a new generation by stochastically selecting some chromosomes from the current population as parents that will breed and produce new offspring. The selection criterion depends on the fitness of the selected parents.
4. Apply crossover between the selected parents to produce new children.
5. Apply Mutation with some small probability to some genes of the newly produced offspring, or to some selected members of the population.
6. Repeat steps 2-5 as needed until a certain stopping criterion is achieved.

Figure 2. 1 : Outline of a GA

Example of crossover (one-point crossover):

Parent1: **1100111|000**

Parent2: 0101010|111

Child1: **1100111**|111

Child2: 0101010|**000**

Example of Mutation

Parent: 11**0**0111000

Child: 11**1**0111000

2.2 Basic Terminology

In biological systems, a chromosome represents the genetic material of an individual. The chromosome is composed of genes carrying hereditary features. The total genetic package is called the genotype, and the organism formed by the interaction with the total genetic package is called the phenotype.

In artificial systems a chromosome is a string or a structure. The structure decodes to a solution or a point in the space. The genes of the structure can take their values from a set of different values called the alleles. The position of the gene in the structure is called its locus.

2.3 Crossover Types

2.3.1 One-Point Crossover

The traditional GA uses one-point crossover, where the two mating chromosomes are each cut once at corresponding points, and the sections after the cuts are exchanged. However, many different crossover algorithms have been devised, often involving more than one cut point.

2.3.2 Two-point crossover

In this technique, two cut points are chosen randomly in the parent chromosome. The section between the selected cut points is exchanged between the two children.

Example:

Parent1: **1100|111|000**

Parent2: 0101|010|111

Child1: **1100**|010|**000**

Child2: 0101|**111**|111

One-point crossover can be seen as a special case of a two-point crossover with one of the cut points fixed at the start of a string.

2.3.3 Uniform crossover

In this technique a random mask of bits is created. Each gene in the offspring is created by copying the corresponding gene from one of the parents. The parent is selected according to the value of the corresponding bit in the mask.

Example:

Mask: 0010101010

Parent1: **1100111000**

Parent2: 0101010111

Child1: **11**0**0**0**1**0**0**1**0**

Child2: 01**0**1**11**1**0**1

2.3.4 Partially Matched Crossover(PMX):

This technique is useful in order-based problems, such as the traveling salesman problem, where gene values are fixed and the fitness depends on the order in which they appear. In PMX it is not the genes that are crossed, but the order in which they appear. Offspring have genes that inherit order information from each parent. This avoids the problem of generating offspring that violate the problem constraints, such as having duplicate cities in a chromosome that represents a solution to the TSP problem.

Example:

Parent1: 9 8 4 |5 6 7 | 1 3 2 10

Parent2: 8 7 1 |2 3 10| 9 5 4 6

Child1 : 9 8 4 |2 3 10| 1 6 5 7

Child2 : 8 10 1|5 6 7 | 9 2 4 3

In this example cities 5,6,7 exchange their positions with 2, 3 and 10 respectively.

2.4 Why use GAs in solving problems?

There are a number of successful search techniques that have been proposed for use in search and optimization problems. Random search, hill climbing and Iterated search are examples of such techniques. Although these techniques usually perform well on functions with only one peak (Unimodal functions), they perform poorly on functions with many peaks (Multimodal), because they can be easily trapped in a local optimum and never locate a global optimum solution. In addition, all these techniques operate by modifying one single solution, and thus are poor in exploring the search space.

GAs, on the other hand, represent an intelligent method for solving problems, because they operate on a random population of solutions and allocate trials to promising areas of the search space. In fact, they usually succeed in solving problems that many other techniques have failed to solve. These are problems that require an adaptive algorithm as opposed to a fixed one, and in this case the best use is made of one of the great features of GAs which is their **Robustness** or their ability to perform consistently well on a broad range of problem types. GAs are not easily affected by the change of the input or the presence of noise as the conventional AI systems, depth-first, breadth-first...etc. They are also appropriate for searching large or multi-dimensional spaces.

Another advantage of GAs is that they do not depend heavily on information available from the problem at hand, they are remarkably easy to connect to existing

simulations and models, and they have a clean interface that just requires the ability to propose a solution and then evaluate it.

GAs are also easy to hybridize to generate knowledge augmented GAs. This could be done in cases where the best answer requires applying some problem-specific techniques of search that can be combined with the general features of GAs. Using the operations of selection of the fittest, mutation, and crossover, GAs quickly reach extremely fit individuals (not always the most fit), but who are fit enough to solve problems of very large magnitude.

2.5 How GAs work?

The Schema Theorem and the Building Block Hypothesis

In order to understand how a GA works its important to shift our attention from strings to what actually a string represents, and how it is similar to other strings in the population at certain string positions. According to Holland (1975), a schema is a similarity template describing a subset of strings with similarities at certain string positions. Considering that a string is represented by the alphabet *{0,1,#}*, where a *#* means don't care, a schema matches a string if at every location in the string a *1* matches a *1* in the string, *0* matches a *0*, and a # matches either. For example, the schema *11#10#* matches the strings *{110100,111100,110101,111101}*.

The length of the schema is the distance between the outer most non # symbols. The length of the schema in the previous example is 5. The order of the schema is the number of non # symbols it contains, the previous schema is of order 4.

A string of length l is a member of 2^l different schemata, because each position may take its actual value or *#*. A population of size n will have a number of schemata

ranging from 2^l to $n2^l$ (where $n2^l$ corresponds to the case that each string in the population represents a unique schema). Holland (1975) showed that the optimum way to explore the search space is to give opportunities of reproduction for individuals in proportion to their fitness with respect to the rest of the population. And since it is assumed that an individuals high fitness is due to the fact that it contains good schemata, in this way good schemata receive an exponentially increasing number of trials in successive generations. By passing some of the good schemata to next generations, the likelihood of finding better solutions exists. This theorem is called the **Schema Theorem** (Holland, 1975).

Holland also showed that since each individual belongs to many different schemata, a GA while operating on individual strings, actually processes a large number of schemata. This number is of order n^3, where n is the population size. This property of GA is called **Implicit Parallelism** and one of the good explanations of the power of GAs.

The effect of the different genetic operators on a particular schema can also be examined. As mentioned above, reproduction favors highly fit schemata by giving them an increasing number of samples. Crossover may disrupt the schema if it was cut by the crossover operator. Schemata of long length are more likely to be disrupted by crossover, while schemata of short defining length are less likely to be destroyed due to crossover. For example, the schema 1####0 is more likely to be cut than the schema #10##. Mutation with its usual small rate does not disrupt the schema very frequently.

The building block hypothesis puts these observations in formal terms by stating that: ***highly fit, short defining length schemata (called building blocks) are propagated generation to generation by giving exponentially increasing samples to the observed***

best ones.(Goldberg, 1989; Holland, 1975). The bits of good building blocks work well together and tend to lead to improved performance when incorporated into an individual.

To encourage the formation of building blocks, a good coding scheme should try to place related genes close to each other and also attempt to avoid interaction between genes. Interaction between genes (known also as Epistasis) means that the contribution of a gene to the fitness depends on the presence of other genes in a chromosome.

Unfortunately, meeting the two recommendations of the building block hypothesis is not usually easy. Multimodal functions, which are the interest of GAs, always have parameters that interact together. A good coding scheme should try to minimize the interaction between genes.

2.6 Advanced Issues in GAs

2.6.1 Genetic Drift and Premature Convergence

A good search method should combine both exploration and exploitation. Exploration is the ability to locate and investigate new areas in the search space, while exploitation is the ability to make use of previous knowledge of visited points. For example, random search is good at exploration, while hill climbing is good at exploitation. Holland (1975) showed that if the following simplifying assumptions are made, a GA will combine both exploration and exploitation. First, the population size is infinite. Second, the fitness function accurately reflects the utility of the solution. Third, the genes in a chromosome do not interact significantly.

Of course, assumption one can never be met in practice. The effect of a limited population size is the accumulation of stochastic errors, which leads to the problem of genetic drift. This problem occurs when a gene becomes predominant and its spreads to

the whole population from one generation to the next. Once a gene has converged in this way, crossover will not be able to change its value, and as generations go by all genes will converge in the same way. The rate of genetic drift can be reduced by increasing the rate of mutation. However, a very high mutation rate makes the search more or less random.

Another closely related problem is premature convergence. This happens when the genes of a highly fit (but not optimal) individual rapidly dominate the population, causing it to converge to a local maximum. When this happens, crossover will produce identical copies of individuals and no further improvement can be achieved unless mutation was very lucky in locating some new promising search areas. This problem is highly related to selecting the more fit individuals for reproduction. To overcome premature convergence the selection strategy could be modified.

In principle, individuals are selected from the population for reproduction form what is called a "mating pool". Highly fit individuals could be copied into the mating pool more than once, while lower fit individuals may not receive any copies in the mating pool. The mating pool is usually the same size as the original population. Pairs of individuals are then selected from the mating pool at random and combined to form new offspring. The process is repeated until the mating pool is exhausted. The following strategies can be used to select parents for reproduction and the choice among them is highly application dependent:

1- Uniform Selection:

This selection method picks randomly from the population. Any individual has a probability of selection that is equal to 1 divided by the population size.

2- Rank Selection:

In this selection method, individuals are sorted according to their objective function values, and each individual is assigned a number of offspring that is a function of its rank in the population

3- Roulette Wheel Selection:

This selection method picks an individual based on the magnitude of the fitness score relative to the rest of the population. The higher the score, the more likely an individual will be selected. The probability of the individual being chosen is equal to the fitness of the individual divided by the sum of the fitnesses of each individual in the population.

4- Tournament Selection:

This method uses the roulette wheel method to select two individuals then picks the one with the higher score. The tournament selection typically chooses higher valued individuals more often than the roulette wheel selection.

5- Deterministic Remainder Sampling Selection.

This selection scheme uses a two-staged selection procedure. In the first stage, each individual's expected representation is calculated. A temporary population is created using the individuals with the highest expected numbers. Any remaining positions are filled by first sorting the original individuals according to the decimal part of their expected representation, then selecting those highest in the list. The second stage of selection is uniform random selection from the temporary population.

6- Stochastic Remainder Sampling Selection.

This method uses a two-staged selection procedure. In the first stage, each individual's expected representation is calculated. A temporary population is filled using the individuals with the highest expected numbers, Any fractional expected representations are used to give the individual more likelihood of filling the space. For example, an individual with an expected number of 1.4 will have one position and a 40 percent chance of having a second position. The second stage selection is uniform random selection from the temporary population.

Besides choosing an appropriate selection strategy, a GA should choose an appropriate replacement strategy. Replacement refers to the method by which new offspring are inserted in the population. To keep the population size fixes, the new offspring generated in one generation should replace other individuals in the previous generation. The replacement strategy may also affect convergence towards an optimal or a sub-optimal solution.

In traditional GAs the whole population is replaced by the generated offspring in each generation. This type of GAs is called the Simple Genetic Algorithm (Goldberg, 1989). Other types of GAs prefer a steady state replacement, where the population of parents and children overlap. In each generation only a few individuals from the original population are replaced. The percentage of population that is replaced each generation is called the generation gap. This replacement scheme is more like the living organisms in which parents and children coexist in each generation, and competition is allowed between them. Steady state replacement requires a strategy for choosing some unlucky individuals to be replaced. For example, a newly generated offspring may replace a

parent, a random individual, or the worst individual in the population. Although steady state GA is relatively computationally expensive, since some statistics of the population have to be calculated after each mating, it has the advantage of making new offspring available once they are generated, which allows more areas of the search space to be explored immediately.

2.6.2 Epistasis and Deception

Epistasis is the interaction between different genes in the chromosome. It is the extent to which the contribution of fitness of one gene depends on the values of other genes. If a small change is made in one gene, the fitness of the chromosome will also change. This resultant change may vary according to the values of other genes.

The level of interaction between genes may be mild or profound depending on the extent to which the chromosome fitness, resulting from a small change in one gene, varies according to the values of other genes. The hardest case is when a particular change in a gene produces a change in fitness that varies in both sign and magnitude depending on the values of other genes. Epistasis usually refers to this hard case.

If the interaction between genes is mild, the problem can be generally solved using various simple techniques like hill climbing. However, GAs can outperform other simple techniques in solving problems with significant epistsis. Unfortunately, however, the building block hypothesis mentioned above requires that a successful GA should be designed with a minimum interaction between genes. This suggests that a GA will not be effective on precisely those cases in which it is mostly needed.

Another very related problem is deception, which is a special case of epistasis. A problem is referred to as deceptive if the average fitness of schemata, which are not

contained in the global optimum, is greater than the average fitness of those which are. A problem is referred to as fully deceptive if all low order schemata containing a sub-optimal solution are better than other competing schemata (Beasley et al., 1993).

In a deceptive problem, schemata, which are not contained in the global optimum, increase more rapidly than those which are. As a result, the GA will be mislead away from the global optimum instead of towards it.

The problem of epistasis may be tackled by changing the coding scheme, such that the interaction between the genes is minimized, and by using appropriately designed crossover and mutation operators. In some problems the effort to do that is not trivial. Traditional GA theory, based on the schema theory, relies on low epistasis. If genes in a chromosome have high epistasis, a new theory may have to be developed, and new algorithms developed to cope with this (Beasley et al., 1993).

2.6.3 Operators Probabilities

As mentioned above, the basic operators of GAs are Crossover and Mutation. Crossover is the main source leading to a thorough search of the search space, because of its ability to produce large jumps and generate new solutions very rapidly. Mutation, however, is also very critical because it is the only way of restoring diversity and avoiding genetic drift and premature convergence. As the population converges mutation usually becomes more productive than crossover.

The chosen probability for both operators is very important in leading the GA towards the optimum solution. The optimum mutation probability is more critical, however, than that of crossover. Usually crossover is performed with a much higher

probability than mutation. The task of choosing appropriate probabilities is application dependent, and it is best achieved with trial and error.

Some researches tried to develop dynamic operator probabilities, where the optimal value for each operator probability may change during the run. Davis (1985) tried linear variations in crossover and mutation probability, with crossover decreasing during the run and mutation increasing. Booker (1985) uses a dynamically variable crossover rate depending on the spread of fitnesses. When the population converges, the crossover rate is reduced to give more opportunity for mutation to find new variations.

Another adaptive technique developed by Davis (1989,1991) depends on giving credit to each operator if produces a chromosome better than any other in the population. During the course of a run, operator probabilities vary in an adaptive problem dependent way. An operator that consistently looses weight is probably less effective than other operators. This technique has the advantage of alleviating the problem of choosing operator probabilities. Its drawback is that credit may be sometimes given to operators that simply locate local optima, rather than helping to find the global optimum(Beasley et al., 1993).

Other researches like Ackley (1987) tried varying the mutation probability by decreasing it exponentially during a run. No clear analysis is given to explain why this approach should lead to an improvement. One possible explanation is that mutation probability is analogous to the temperature of SA, which must be reduced during the run to aid convergence. Increasing Mutation probability near convergence may introduce a large degree of diversity that could lead the GA away from convergence.

2.6.4 Niche and Speciation

In natural systems, a niche may be viewed as the organism's role in the environment and a species is a class of organisms with common characteristics. Inducing a niche like and speciation behavior in a GA can help its search towards the global optimum.

In a GA, niches are analogous to maxima in the fitness function. A multimodal function have several peaks, and it might be desirable to locate all peaks. Unfortunately, a traditional GA will not do that because eventually all the population will converge on a single peak, this is due the genetic drift problem introduced above. The basic techniques used to overcome this problem and to encourage a niche like behavior in a GA are:

1. Maintaining Diversity
2. Sharing the Payoff associated with a niche.

Maintaining diversity can be achieved by several techniques. For example, a technique called pre-selection (Grefenstette, 1987) a newly produced offspring replaces its parent only if its fitness is higher than the fitness of its parent. This helps maintaining diversity because individuals replace others that are similar to them.

Another technique called crowding (Dejong, 1975) diversity is maintained by allowing an offspring to replace the most similar individual from a set of randomly chosen individuals in the population, using hamming distance as a similarity measure, i.e. an individual replaces another individual in the same niche. A multiple sub-population approach with migration has also been used to simulate niching and speciation (Grefenstette, 1987). Goldberg and Richardson (Grefenstette, 1987) describe the advantage of sharing, several individuals which occupy the same niche are made to share

the fitness payoff among them. Once a niche has reached its full capacity it no longer seem rewarding in comparison with other unfilled niches.

Another important technique to encourage speciation and reduce the formation of lethals is restricted mating. A lethal is an unfit child that is produced by two highly fit individuals. Nature avoids the formation of lethals by preventing mating between different species. Restricted mating only allows individuals from the same niche (similar to each other in their phenotypes) to mate.

2.7 Applications of Genetic Algorithms

GAs have been used successfully in a wide range of problems. Some of these problems have been used in practice and others still remain in the research area. The most important applications are:

Numerical Function Optimization: GA techniques were found to outperform others on difficult, discontinuous, multimodal and noisy optimization problems.

Image Processing: This includes the task of aligning two images of the same area taken at different times, such as x-ray or satellite images. Another application is producing pictures of criminal suspects where the witness operates as the objective function for each picture.

Combinatorial Optimization Problems: which include tasks that require arrangements of discrete objects. Such as the traveling salesman problem, bin packing, job shop scheduling or time tabling.

Design Tasks: which can be a mixture of combinatorial and function optimizations. For example, designing bridge structures, optimal routing in multiprocessor systems, construction of neural networks, and many other design tasks.

Machine Learning: the classical example is classifier systems developed by (Holland et al., 1989) for learning a set of "if.. then" rules to deal with a particular situation. This has been applied to game playing and maze solving as well as political and economic modeling.

2.8 Knowledge Based and Hybrid Techniques.

The above discussion shows that GAs have a great potential. They are not limited to solving one class of problems, but many different classes. Moreover, their potential increases if they were tailored and modified to suit the particular application under consideration. Knowledge-based techniques try to combine problem specific information with genetic algorithms. Although this approach will make the GA less robust, because it will be more problem specific, it may improve performance significantly. For example, chromosome representation is not restricted to bit strings. A chromosome can be represented by a vector, a graph, a string of characters, or even a complete object. This latest technique has been used for minimum cost routing and wavelength allocation in a network of nodes. The chromosome in this case was represented as a C++ object, composed not only of nodes, links and ordered sequences of paths, but also objects representing the network adjacency matrix, connection matrix, and the traffic requirements. Network objects are themselves the structure undergoing adaptation (Sinclair, 1989).

Problem specific knowledge can be also incorporated into the crossover or mutation operators. For example, a crossover operator may be modified in a way that prevents the formation of poor or invalid chromosomes. This will reduce the amount of time wasted during the evaluation of such poor chromosomes. Problem specific

knowledge can also be used for heuristic initialization in which the initial population contains a set of reasonably good points instead of a completely random set of points.

Hybrid techniques have been also used in many applications to improve the performance of genetic algorithms. For example, a GA can be combined with a local search technique such as simulated annealing or hill climbing techniques, in which a GA can be used to find the hills, and a local search technique is used to climb the hills and improve the solution obtained. A particular advantage of this technique is that a GA can spend excessive time refining an acceptable solution, while a simulated annealing search for example has an adjustable parameter – the cooling rate- that can be fine tuned to minimize the time to a reasonable bound. Combining both techniques offers the advantages of both.

2.9 Parallel Genetic Algorithms

Parallel processing is the current trend of computer science, and genetic algorithms are attractive for application on parallel machines. As mentioned above, GAs are not guaranteed to find an optimal solution; however as the size of the population increase, the chances of finding more efficient solution become better. Unfortunately, this increases computation time and cost of the solution.

However, since GAs operate on a population of individuals, this makes them parallel in nature. Parallel genetic algorithms (PGAs) can be used to process a large number of individuals in parallel and produce better solutions in less time, because they better explore the search space. PGAs maintain multiple separate sub-populations that are allowed to evolve independently in parallel, and this allows each sub-population to

explore different parts of the search space, each maintaining its own individuals and controlling how mixing occurs with other sub-populations.

PGAs also help to overcome the problem of **premature convergence**, in which a sub-optimal solution dominates the population and leaves no chance for improvement. The children chromosomes produced thereafter will be very similar to each other and to their parents, thus causing the crossover operation to be largely ineffective.

Another advantage of parallel GAs is to help find solutions for multi-objective functions, i.e. each sub-population can emphasize a different objective.

There are different ways to parallelize GAs, but they can be divided into three main categories: **Global Parallelization**, **Coarse Grained Parallel GAs**, and the **Fine Grained Parallel GAs**.

2.9.1 Global parellization (Micro-Grain GAs)

This technique is characterized by having a single population while the evaluation of individuals is performed in parallel. A speedup that is proportional to the number of processors is expected, although probably sub-linear. This technique is most suited for applications in which the fitness function is very costly.

Each processor is assigned a subset of the population, and it is responsible for evaluating the fitness of each of its members. The best case is when each processor is assigned only one individual, making evaluation time equivalent to the time needed to evaluate the most costly individual. There is no communication between the processors, because fitness evaluation is an independent process.

There are two models to be considered here:

2.9.1.1 The Shared Memory model

In this model each processor can read individuals assigned to it from the shared memory, and write the results back in the shared memory without conflict. However, synchronization between generations is needed, such that each processor must wait for all others to finish evaluating individuals belonging to one generation, before processing the next generation.

2.9.1.2 The Distributed Memory Model

In this model there is a "master" processor that is responsible for storing the population, and sending the individuals to the "slave" processors that will evaluate their fitness. Then, the master collects the results and applies the genetic operators (mutation and crossover) to produce the next generation.

Global parallelization can be used to perform genetic operators (selection, mutation and crossover) in parallel as well. However, operators that require global statistics such as population average fitness are not suitable in this case, because they will cause serious performance bottleneck. Also, communication between processors and message passing, in case of distributed memory, is a serious drawback that might degrade performance.

2.9.2 Coarse Grained Parallel GAs

The grain size in parallelism refers to the ratio of the time spent in computation and the time spent in communication. When this ratio is high the processing is called coarse grained, and is suitable for implementation on MIMD (Multiple Instruction Multiple Data) machines.

In this model the population is divided into several sub-population, each is assigned to one processor and is allowed to evolve independently from the others. Occasionally some individuals migrate between sub-populations in order to introduce new genetic material that will allow a better exploration of the search space and avoid premature convergence.

The parameters that affect this model can be categorized along three dimensions: migration method, connection scheme, and processor node homogeneity.

2.9.2.1 Migration Method

Migration is controlled by two parameters: **migration rate**, which determines how many individuals are migrated between sub-populations, and **migration interval**, which determines when migration occurs.

As mentioned above, migration is important to allow for introducing new genetic material in a sub-population (also called a deme). This is based on the theory of **"Punctuated Equilibria"**, which states that new spices are likely to form quickly in relatively small isolated populations after some change in the environment occurs.

Previous research has shown that there is a critical migration rate below which the performance of the algorithm might degrade as a result of the isolations of demes, and above which the partitioned population behaves as a one large population.

There are basically three methods of migration:

Isolated Island GAs: in which there is no migration between sub-populations.

Synchronous Island GAs: in which all sub-populations evolve at the same rate, after which migration can occur, for example, migration occurs after a certain number of

generations. Of course, different machine speeds and different loads can cause some processors to stay idle while others are still processing their sub-populations.

Asynchronous Island GAs: in which migration occurs irrespective of the state of evolution across all of the system's sub-population. This kind of migration is more suitable for the different loads and diverse machine architectures as well as the large number of processors found in the parallel machines of today. However, in this approach there is a possibility that a relatively high-fitness individual from a fast-evolution node is inserted in low-fitness population on a slow-evolution node. This might cause the genetic material of this individual to dominate the sub-population, possibly resulting in premature convergence.

Some variations of these migration methods exist, such as sending a copy of the best individual found in each deme to all its neighbors after every generation to ensure good mixing. Another migration method performs migration after the sub-population has converged. And a third one uses a master processor to which all processors executing the GAs periodically send their best individuals. Then the master processor chooses the fittest individuals among those it received, and broadcasts them to all the nodes.

2.9.2.2 Connection Schemes

The connectivity of the processing nodes is also very important in the performance of parallel GAs, because it determines how fast (or how slow) a good solution disseminates to other demes. If the topology has high connectivity or short diameter or both, good solutions will spread very fast, and may quickly dominate the population, possibly causing premature convergence. On the other hand, if the connectivity is low or the diameter of the network is large, good solutions will spread

slowly, allowing for better exploration of the search space. There are two main connection schemes:

Static Connection Scheme: the network topology is static and does not change over time. There are various topologies, such as a rings, meshes, n-cubes, etc., but the topology determines which nodes are allowed to exchange individuals.

Dynamic Connection Scheme: here the topology of the network is mutable during run time. This allows for changing the migration scheme based on the current state of the evolutionary process. For example, individuals can migrate only to other sub-populations that are similar (or dissimilar) to them in terms of hamming distance, which can make the migration process more effective.

There are also some variations of the migration schemes, such as sending individuals to random destinations, rather than destinations that are determined by the topology or the similarity between individuals.

2.9.2.3 Node Homogeneity

Node homogeneity is a measure of how similar the GA processes are on different processing nodes. We can distinguish the following categories:

Homogeneous Island GAs: in which all nodes use the parameters (population size, crossover rate, mutation rate, migration interval, etc.)

Heterogeneous Island GAs: sub-populations evolve with different parameters, genetic operators, objective functions and encoding methods. Of course, interchanging individuals between sub-populations in this case will be more difficult, but these problems can be addressed.

2.9.3 Fine Grained Parallel GAs

In this model, the population is divided into a large number of very small demes, the ideal case is to have one individual for every processing element. This model calls for massively parallel computers, and is suitable for implementation on SIMD (Single Instruction Multiple Data) machines.

Here also, selection and mating occur within a single sub-population, but the demes overlap providing a way to disseminate good solutions across the entire population; i.e., selection and crossover are performed between a processor and its neighbors. For example, a processor may borrow some individuals from the neighboring processors, perform genetic operations on them, and discard the worst ones.

The danger of a sub-optimal state being reached in this model is greater than any other model for two reasons. First, there is a greater degree of migration between sub-populations. Secondly, the number of chromosomes in each sub-population is less than any other PGA model. In order to address these problems, the size of the network and the degree of overlap must be controlled.

It is common to place the individuals of a fine-grained PGA in a 2-D grid, because in many parallel computers, the processing elements are connected using this topology. However, most of these computers have a global router that can send messages to any processor in the network (at a higher cost) and other topologies can be simulated on top of the grid. Some research show that the performance of the fine grained PGA is affected by the topology of the interconnection network, and it seems that the topology with a medium diameter gives good results.

2.9.4 Hybrid Algorithms

A few researchers have tried to combine two of the methods to parallelize GAs. For example, we can use global parallelization on each of the demes of a coarse grained GA. Migration occurs between demes as in the coarse grained algorithm, but the evaluation of individuals is handled in parallel.

2.9.5 Underlying Problems

Although parallel GAs promise a lot in terms of both speed up and quality of solutions, there is a number of underlying problems that are not fully addressed, two of these problems are population size and migration

Population Size: this is related to the deming issue. That is, to how many demes should the population be divided? And what is the size of each deme?

Migration: after determining the number and the size of each deme, we need to establish the way they are going to communicate (migration). The parameters affecting migration are migration interval and migration rate. Migration interval is related to when individuals should be migrated. Intuitively, migration should occur after the expected number of building blocks in each individual is relatively high, i.e. when it covers a reasonable part of the search space. Migration before that is a waste of resources. Migration rate is the number of individuals that must migrate. If the migrated individuals are rich in their building blocks, then it is enough to migrate just a few.

Topology: which is the best way to connect demes. In a topology with a long diameter, good solutions will take longer to reach all demes. On the other hand, in a topology with a small diameter, good solutions will spread very fast and possibly dominate the population.

Chapter 3

3 Simulated Annealing

3.1 Overview

Simulated annealing is a well-known heuristic search method that has been used successfully in solving many combinatorial optimization problems (Chams et al., 1987; Connolly, 1988; Wright, 1989). It is a hill-climbing algorithm with the added ability to escape from local optima in the search space. However, although it yields excellent solutions it is very slow. It is mostly used for solving problems that are not very well understood.

The term simulated annealing is adopted from the annealing of solids where we try to minimize the energy of the system using slow cooling until the atoms reach a stable state. The slow cooling technique allows atoms of the metal to line themselves up and to form a regular crystalline structure that has high density and low energy. The initial temperature and the rate at which the temperature is reduced are called the annealing schedule.

In solving a combinatorial optimization problem using SA, we start with a certain feasible solution to the problem. We then try to optimize this solution using a method analogous to the annealing of solids. A neighbor of this solution is generated using an appropriate method, and the cost (or the fitness) of the new solution is calculated. If the new solution is better than the current solution in terms of reducing cost (or increasing fitness) the new solution is accepted. However, if the new solution is not better than the current solution, the new solution is accepted with a certain probability which is usually

set to exp(- Δ/T) Where Δ is the change in cost between the old and the new solution and T is the current temperature, i.e. the probability decreases exponentially with the badness of the move. Thus the procedure is less likely to get stuck in a local optimum since bad moves still have a chance of being accepted.

The annealing temperature is first chosen to be high so that the probability of acceptance will also be high, and almost all new solutions are accepted. The temperature is then gradually reduced so that the probability of acceptance will be very low and the algorithm works more or less like hill climbing, i.e. high temperatures allow better exploration of the search space, while lower temperatures allow fine tuning of a good solution. The process is repeated until the temperature approaches zero or no further improvement can be achieved. Which is analogous to the atoms of the solid reaching a crystallized state.

3.2 Theoretical Foundation

The theoretical foundation of SA was lead by Kirkpatrick et al. (1983). The theory is based on statistical mechanics of physical systems.

Consider a physical system with many degrees of freedom that can reside in any one of a large number of possible states. The probability of occurrence of state *i* is $\mathbf{p_i}$, such that:

$$p_i >= 0 \qquad (1)$$

$$\sum_i p_i = 1 \qquad (2)$$

Let E_i denote the energy of the system when it is in state i. A fundamental result in statistical mechanics tells us that when the system is in thermal equilibrium with its surrounding environment, state i occurs with probability defined by

$$P_i = 1/Z \exp(-E_i/K_bT) \qquad (3)$$

Where T is the absolute temperature in Kelvins, K_b is Boltzmann's constant, and Z is a constant that is independent of all states. One degree Kelvin corresponds to –273 degrees Celsius, and $K_b=1.38 \times 10^{-23}$ jouls/kelvin.

Imposing the condition for the normalization of probabilities defined by equation (2) on equation (3) gives us the normalization constant Z

$$Z= \sum_i \exp(-E_i/K_bT) \qquad (4)$$

Z is called the partition function and the factor $\exp(-E_i/K_bT)$ is called the Boltzmann factor. The distribution given by equation *(3)* is called The *Boltzmann distribution.* Two important properties of the Boltzmann distribution are:

1. States of low energy have a higher probability of occurrence than states of high energy.
2. As the temperature T is reduced, the probability is concentrated on a smaller subset of low energy states.

For example, growing a single crystal from a melt, require that the atoms reach a stable low energy state at a low temperature. However, achieving a perfect crystal requires lowering the temperature slowly enough, and spending a long time at temperatures in the vicinity of the freezing point. If cooling proceeds two fast, the atoms will be allowed to get out of equilibrium and the resulting crystal will have many defects.

Kirkpatrick makes the connection between statistical mechanics and combinatorial optimization by stating that: ”Finding the low-temperature state of a system when a prescription for calculating its energy is given is an optimization problem not unlike those encountered in combinatorial optimization...” (Kirkpatrik et al., 1983)

Kirkpatrick et al. (1983) applied the Metropolis algorithm from statistical mechanics to combinatorial optimization problems. The Metropolis algorithm introduced in (Metropolis et al., 1953) provides an efficient simulation of atoms in equilibrium at a given temperature. The algorithm provides a generalization of iterative improvement where controlled uphill moves (moves that do not lower the energy of the system) are probabilistically accepted in the search for obtaining better solutions and escaping local optima.

In each step of the Metropolis algorithm an atom is given a small random displacement. If the displacement results in a decrease in the system energy, the displacement is accepted and used as a starting point for the next step. If on the other hand the energy of the system is not lowered, the new displacement is accepted with a certain probability $exp(-\Delta E/k_bT)$ where ΔE is the change in energy resulting from the displacement, T is the current temperature, and k_b is a constant called a Boltzmann constant . Depending on the value returned by this probability either the new displacement is accepted or the old state is retained. For any given T, a sufficient number of iterations always lead to equilibrium, at which point the temporal distribution of accepted states is stationary (this distribution is called the Boltzman distribution).

The SA algorithm has also been shown to possess a formal proof of convergence using the theory of Markov chains (Eglese, 1990). A sequence of random variables

$X_1,X_2,...,X_n,X_{n+1}$ forms a Markov chain if the probability that the system is in state X_{n+1} at time $n+1$ depends exclusively on the probability that the system is in state X_n at time n. We may think of a Markov chain as a model consisting of a number of states linked together on a pair-wise basis by possible transitions. If the transition probabilities are fixed and do not change with time, the Markov chain is said to be homogeneous in time.

In case of a system with finite possible sates K the transition probabilities constitute a K *–by-* K matrix, where p_{ij} represents the probability of transition from state i to state j.

$$P = \begin{bmatrix} p_{11} & p_{12} & \cdots & p_{1k} \\ p_{21} & p_{22} & \cdots & p_{2k} \\ \vdots & \vdots & \vdots & \vdots \\ p_{k1} & p_{k2} & \cdots & p_{kk} \end{bmatrix}$$

In the above system, the transition from one state to another takes place in some fixed number of steps.

In an SA algorithm, if the temperature parameter T is kept constant, the probability of moving from any state i to any other state j is independent from the iteration number. Representing these probabilities in a transition matrix, it may be shown that it is possible to get from any state i to any other state j in a finite number of moves, which corresponds to a homogeneous Markov chain. This Markov chain has a unique stationary distribution, which does not depend on the initial state. This distribution corresponds to the Boltzmann distribution in statistical mechanics.

The Limit as $T \rightarrow 0$ of this stationary distribution is a uniform distribution over the set of optimal solutions, i.e. *the SA algorithm converges asymptotically to the set of globally optimum solutions***.** This convergence property is a very important and

encouraging result. The question, however, is how many iterations are sufficient to guarantee convergence?

In an attempt to solve this question we can try to describe the SA algorithm as a sequence of Markov chains of finite length, using decreasing values of the temperature *T*. This can be considered as a single non-homogenous Markov chain, as the transition probabilities are now not independent of the number of iterations, which violates the homogeneity condition of a Markov chain. This model of SA does not require the stationary distribution to be reached at any non-zero temperature.

Using this assumption, Hajek provides a necessary and sufficient condition for convergence (Hajek, 1988). He showed that if $T(k)=c/\log(1+k)$, where *k* is the number of iterations, the condition for convergence is that the constant *c* be greater than or equal to the depth of the deepest local minimum which is not a global minimum. This temperature function represents very slow cooling. It has also been shown in (Mitra et al., 1986) that attempting to approximate the uniform distribution on the set of optimal solutions, typically leads to a number of iterations, which is larger than the size of the solution space, and so results in exponential running time for most problems.

3.3 Implementing SA

Implementing SA requires many choices that must be considered. These choices include both general SA parameters and problem specific decisions. The choice of the general parameters of the SA operator is critical to the performance of the algorithm. Following (Eglese, 1990) These parameters are:

1. The value of the initial temperature *T*.

2. A temperature function *T(t)* that determines how the temperature will change with time.
3. The number of iterations *N(t)* to be carried at each temperature
4. A stopping criterion to terminate the algorithm.

The initial temperature *T(0)* is generally chosen high enough so that almost any move is accepted regardless of its fitness. This choice is adopted from the physical analogy and corresponds to heating up a substance until all particles are randomly arranged in a liquid.

The temperature function is usually a proportional temperature function $T(t+1)=\alpha\, T(t)$ where α is a constant close to *1*. Typical values of α used lie between *0.8* and *0.99*. Using such value provides very small decrements of temperature values, which corresponds to a very slow cooling of the substance until the temperature approaches zero.

The number of iterations carried out at each temperature value should be large enough to bring the system to a stable state analogous to thermal equilibrium in the physical analogy. Some applications may choose *N(t)* to be constant for each temperature. The stopping criterion of the algorithm is usually the stagnation of the system when no change in the result can be obtained for a specified number of iterations or temperature changes.

The choice of a cooling schedule is a subject of controversy among researchers. Some choose *N(t)* to be large enough at each value of *T* such that the system approaches "a stationary distribution " at that value of *T*. However, as mentioned above, the result achieved by Hajek in (Hajek,1988) indicates that if the cooling process is performed

sufficiently slowly, there is no need to attain equilibrium at each temperature. There is a trade-off between a large reduction of temperature values and a small number of iterations at each temperature. Some researches even suggest only one iteration per temperature value (Lundy & Mees, 1986), while providing very slow reduction of temperature values. Others suggest that the majority of iterations should be conducted at suitably fixed temperature (Connolly, 1988).

Whatever cooling schedule is chosen, it is important not to spend to long at high temperatures, where most neighborhood moves are accepted. It is also important not to spend too long at the end of the algorithm where most moves are rejected. These two situations can waste much processing time. At the end of the algorithm it is worth checking that at least a local optimum has been obtained (Eglese, 1990).

Implementing SA also requires a set of problem specific decisions. These include: defining the set of feasible solutions to the problem, define a clear objective function, generating an initial solution, and defining a neighborhood operator that generates moves using current solution.

The topology of the neighborhood structure is critical to the performance of SA algorithm. This is clearly indicated by the result obtained by Hajek (1988) as mentioned above, which states that the rate of cooling required for asymptotic convergence depends on the depth of the deepest of the local minima, i.e. the topology of the neighborhood structure. In general, a smooth topology with shallow local optima is favored over a bumpy topology with many deep local minima.

A neighborhood function is easy to implement for discrete problems. Implementing a neighborhood function for continuous problems is more challenging.

Continuous SA should choose a point on the unit hyper-sphere at random about the point representing the current solution. This selection gives the search direction. The algorithm would then choose a random length to step in that direction.

Constrained problems also raise some difficulties. A choice must be made between restricting the solution space to solutions that conform to the constraints, or allowing solutions that break the constraints at the expense of a suitably defined penalty function. The generic simulated annealing algorithm is described in Fig 3.1.

```
1-      Start with some state S.
2-      T= T0
3-      Repeat{
4-          While (not at equilibrium){
5-              Perturb S to get a new state Sn
6-              ΔE = E(Sn)-E(S)
7-              if ΔE < 0
8-             replace S with Sn
9-      Else with probability e -ΔE/T
10-             replace S with Sn
11-     }
12- T = c × T   // c<1
13- } until frozen
```

Figure 3. 1 The SA Algorithm

3.4 Modifications to SA

A major problem in applying the SA algorithm is the trade-off between the quality of the solution and a very long computational time imposed by the requirement of slow cooling to achieve convergence.

In an attempt to solve this problem some modifications to the basic SA algorithm have been suggested. These modifications are easy to implement and have provided an improvement of the quality of the solution and/or processing time (Eglese, 1990).

One attempt was to **store the best solution found so far**. Since the SA algorithm accepts solutions probabilistically, it may accept solutions that are worse than the current solution. A good solution found during the run may be discarded because it was not lucky during the acceptance attempt. Storing the best solution found so far prevents the SA algorithm from returning a solution that is worse than the best solution ever found. In addition, Glover and Greenberg (1989) argue that with this modification there is less need for the SA algorithm to rely on a strong stabilizing effect over time. Connolly (1989) support this idea by showing that using this modification it is possible to find a suitable fixed temperature and to carry out all remaining iterations at that temperature. In the final phase, a descent algorithm can be employed to find the local optimum containing the best solution encountered in earlier phases.

Another modification is **sampling the neighborhood without replacement**. At the end of SA run, the temperature drops to very small values making the probability of accepting a new move very low. If only a few moves improve the current situation, the algorithm will waste a lot of time trying to locate these moves. The modification tries to generate neighborhood moves in such a way that all possible moves in the neighborhood

of a solution are attempted before repeating a move, unless a new solution is accepted. For example, the neighborhood may me searched in a sequential manner as in (Connolly, 1988) or in random manner as in (Jhonson et al., 1987).

Some researches also modify the basic SA algorithm using **problem specific information.** For example, Grover (1986) showed that significant speedups can be obtained by calculating the change in objective value Δ using approximate rather than exact methods. These approximations were found to yield solution quality compatible with exact methods, provided the error is kept less than a particular function of the temperature T.

Tovey (1988) suggested an adaptive method for approximately calculating Δ. The approximate method is not used every iteration, but it is applied with a certain probability which is updated as the algorithm proceeds, so that the resulting algorithm will simulate the results of the basic SA approach.

Tovey (1988) also suggests a neighborhood operator that is able to identify promising areas in the neighborhood, and gives a greater probability to generate moves that fall in the promising areas. It is not clear, however, whether this technique performs significantly better than sampling the neighborhood without replacement.

In an attempt to reduce the amount of time spent in the latter part of the SA algorithm when the majority of the moves are rejected, Green and Supowit (1986) proposed a rejection-less method. In this scheme a probability distribution is constructed over the set of all moves to show the relative probability of a move being accepted if it is chosen. A move is then generated at random according to this distribution and accepted automatically. Applying this algorithm to different problem types requires finding an

efficient way to update the probability distribution, and this depends on problem specific information.

3.5 Hybrid Techniques

There are two basic approaches to combine SA with other techniques

1. Using another technique to generate a good initial solution that SA can improve.
2. Using SA to generate a good solution that can be used by another search technique.

An example of the first technique is found in (Chams et.al.1987) in solving the graph coloring problem, and in (Jhonson et.al., 1987) in solving the graph partitioning problem. A good starting solution obtained using another search method was found to improve both the quality of the solution as well as the processing time. When using this technique the initial temperature chosen should be reduced, in order not loose the benefits gained by having a good starting solution. They also show that using problem specific information to generate a good starting solution is preferable to using general heuristics. The second approach is exemplified by using SA as a way of obtaining a good initial solution for a branch and bound, an integer programming, or an evolutionary algorithm.

3.6 Parallel SA

Although parallelizing SA is not an easy task because the algorithm is inherently sequential, several approaches have been developed to implement parallel versions of the algorithm with the aim to speed up the search process.

According to Eglese (1990), two main strategies of paralleliziation are used: single-trial parallelism and multiple-trial parallelism. In the first strategy, the calculations

to evaluate a single trial are divided among a number of processors. The implementation of this strategy and the speed up that can be achieved is problem dependent, since it depends on the how the serial portion is processed in parallel.

In the second strategy, several SA trials are evaluated in parallel. Three variations of this strategy exist:

1. **The division algorithm:** in which the number of iterations at each temperature is divided among the processors. After a change in temperature, each processor may start from the final solution obtained by that processor at the previous temperature. The best solution found among all processors is then taken to be the final solution returned by the algorithm. Another variant of this approach is to transfer the best solution found among all processors after each stage of temperature change.
2. **The clustering algorithm:** In this algorithm, two or more processors are used to generate one sequence of accepted moves. Processors evaluate possible moves independently until one is accepted. This move is then communicated to all processors in the cluster, which abort their current calculations and resume with the accepted solution. The algorithm should be more efficient towards the end of a run than at the beginning. This is due to the reduced number of accepted solutions at the later phase of the algorithm.
3. **The error algorithm:** In which several processors are used to investigate potential neighborhood moves in parallel. Any accepted move updates the current solution. The name of the algorithm is derived from the fact that that some calculations made by a processor of the change in objective value for a potential move may be calculated wrongly if another processor has just accepted a move of which the

processor is unaware. The performance of the algorithm is thus dependent on the neighborhood structure and the amount of error resulting from two moves being accepted simultaneously.

3.7 SA Performance

Assessing the performance of SA requires considerable testing and comparison with other search methods. In general the same remarks can be made concerning the performance of SA if compared with other techniques.

First, a comparison of SA with an iterative descent algorithm that starts from several random starting positions indicates that SA can give significantly better results in the same amount of computing time. The result, however, is not general because it is highly dependent on the problem type and the neighborhood structure. For example, a problem with a search space that has one global optimum and no local optima is easily solved using a descent algorithm faster than SA.

Second, in comparison with problem specific techniques, Jhonson et al. (1987,1988) found that SA outperformed some classical graph partitioning algorithms, in both quality and speed for certain types of random graphs. The same result was obtained for solving the traveling salesman problem. For graph coloring problems, SA was able to find very good solutions but with increased processing time. Again, this result cannot be generalized since it is obvious that SA as a general algorithm may not be able to compete with some techniques designed specifically for certain problems.

Finally, a comparison of SA with other heuristic methods is still inconclusive. For example, a comparison with Tabu search for graph coloring problem done by Hertz and de Werra (1987) indicated that Tabu search was superior to SA. However, in another

research done by Bland and Dawson (1989), SA was found to obtain better results for the layout optimization problem than Tabu search. More research is still needed in this area to find the best way of implementing these algorithms.

3.8 Advantages of SA

SA has several attractive features, especially in difficult optimization problems in which a good solution with a reasonable computational effort and processing time is preferred to an optimal solution with much greater programming or computing effort. The basic advantages of SA are:

1- It is very easy to implement, since it just requires a method for generating a move in the neighborhood of the current solution, and an appropriate annealing schedule.

2- It can be applied to a wide range of problem types. For example, any combinatorial optimization problem can be tackled using SA if an appropriate neighborhood structure has been devised.

3- High quality solutions can be obtained using SA if a good neighborhood structure and a good annealing schedule have been chosen. This, however, may be at the expense of a long processing time.

Chapter 4

4 Uncertainty and Belief Networks

4.1 Knowledge Based Agents and First Order Logic

The most important task of any AI (Artificial Intelligent) system is to build agents that act rationally. An agent is anything that can be viewed as perceiving its environment through sensors and effectors. A rational or logical action is an action that causes the agent to be most successful, i.e. an action that maximizes the performance measure of the agent. A rational action is highly dependent on the agents knowledge of the world and the environment in which its operates, i.e. its knowledge base.

One technique used in AI to represent the knowledge base of the agent uses first order logic, which is a general purpose representation language that is based on an ontological commitment to the existence of objects and properties or relations in the world. Some of these relations are functions in which there is only one value for a given input. The following are examples of objects, properties, relations and functions:

Objects: people, houses, numbers, theories, colors, wars, countries …

Relations: brother of , bigger than, inside, part of, has color ….

Properties: red, round, bogus, prime …

Functions: father of, best friend, one more than …

First order logic can express any thing that can be programmed. It has sentences, which represent facts, terms which represent objects. It also has constant symbols, variables, and function symbols that are used to build terms, and quantifiers and predicate

symbols that are used to build sentences. Knowledge representation using first order logic is the most studied and best understood knowledge representation scheme used in AI (Russel and Norvig, 1995).

4.2 Acting Under Uncertainty

One problem with first order logic, and thus with the logical agent approach, is that the agent always never have access to the whole truth about its environment. There are always some questions that the agent cannot answer, and there are always some incorrectness or incompleteness in the agents understanding of his environment. The agent must therefore act under uncertainty. First order logic cannot correctly and completely represent all facts about the domain, because there are too many conditions to be explicitly enumerated, or because some of the conditions are unknown.

For Example, trying to use first order logic to cope with a domain like medical diagnosis will fail for several reasons:

1. **Difficulty:** It is too much work to list the complete set of antecedents or consequents needed to represent a complicated rule, and too hard to use the enormous rules that result.
2. **Incomplete theoretical knowledge:** Medical science has no complete theory for the domain.
3. **Incomplete practical knowledge:** Even if all theoretical knowledge is available, we may not be certain about a particular patient's case because all the necessary tests have or cannot be done.

Knowledge in the medical domain, as well as many other domains such as low, business, design, automobile repair, and many others, can best be represented by only a

degree of belief as opposed to a fact. Dealing with a degree of belief is done using probability theory, which assigns a numerical degree of belief between 0 and 1 to sentences. A probability of 0 corresponds to a definite belief that the sentence is false. A probability of 1 corresponds to a definite belief that the sentence is true. Probabilities between 0 and 1 correspond to intermediate degrees of belief in the truth of a sentence.

When deciding upon actions, an agent assigns probabilities to certain propositions. The agent assigns probabilities to propositions depending on the percepts that it has received from its environment. In uncertain reasoning this is called the evidence. As the agent receives new percepts, its probability assessments are updated to reflect the new evidence. Before the evidence is obtained we talk about prior or unconditional probability. After the evidence is obtained we talk about posterior or conditional probability. In most cases, the agent will have collected some evidence from its percepts, and is interested in determining the conditional probabilities of the outcomes given the evidence it has.

4.2.1 The joint probability distribution

One important notion in dealing with probabilities is the joint probability distribution, which completely specifies an agent's probability assignments to all propositions in the domain. A probabilistic model of a domain consists of a set of random variables that can take on particular values with certain probabilities. Let the variables be $X_1, X_2, \ldots X_n$. An atomic event is an assignment of particular values to all the variables, i.e. a complete specification of the state of the domain.

$P(X_i)$ is a one dimensional vector of probabilities for the possible values of the variable Xi. The joint probability distribution $P(X_1, X_2, \ldots X_n)$ assigns probabilities to all

possible atomic events. Thus, the joint probability is an *n*-dimensional table with an entry for every possible state that gives the value of the probability of the occurrence of this state.

The joint probability distribution can answer any question about the domain, but as the number of variables increases, the joint probability distribution increases exponentially. In addition, the task of assigning probabilities to variables is usually difficult and may be infeasible unless a sufficient amount of statistical information is available.

4.2.2 Baye's Rule

One important rule that allows unknown probabilities to be calculated from known stable ones is Bay's rule. Its general form is:

$P(B/A) = P(A/B)P(B) / P(A)$

Its application in a field like medical diagnosis is very useful. In many cases statistical data provides information about some prior or unconditional probabilities, as well as some conditional probabilities on cause and effect relationships (disease and symptoms), which helps in calculating the values of other unknown probabilities.

4.3 Probabilistic Reasoning Systems

In most cases there is no enough information in the environment to prove that any given action will work. In this case logical reasoning will not be of much help in achieving rational decisions. The agent should be able to use probabilistic reasoning to achieve the decisions that will maximize its success.

As mentioned above, the joint probability distribution can answer any problems about the domain. The problem, however, is that it is not practical to use in cases that

have a large number of variables. It is also a very difficult task to specify probabilities of all atomic events.

Using Bay's rule simplifies the computations required to answer specific questions about conditional probability values, because it incorporates conditional independence relationships among variables. One tool used to capture uncertain knowledge and represent dependence between variables in an efficient way consistent with Bay's rule is Bayesian Belief Networks (BBNs).

4.3.1 BBNs

Bayesian Belief networks (BBN) are used to represent and reason about complex systems under uncertainty and combine the advantages of an intuitive visual representation with a sound mathematical basis in Bayesian Probability. They have been applied to many scientific applications such as medical diagnosis, diagnosis of mechanical failures and computer language understanding.

BBNs are DAG (Directed Acyclic Graphs) graphs that represent the probabilistic dependence and the conditional independence among variables. Each variable is represented as a node and each variable can assume a number of discrete values with a certain conditional probability, given the values of its parents.

According to Russell and Norvig (1995), the following properties hold for a BBN.

1. A set of random variables makes up the nodes of the network.
2. A set of directed links connect pairs of nodes. A link between two nodes indicates that there is a direct influence from the first to the second, the first is the parent and the second is the child.

3. Each node has a conditional probability table that quantifies the effects that the parents have on the node.
4. The graph has no directed cycles.

Pearl (1988) provides a more formal definition. Let $D = (V,E)$ be a directed acyclic graph whose nodes are identified with the random variables $v_1, v_2, \ldots v_n$ from the set of random variables V. D is said to be a *minimal independency map of P* (where P is the probability distribution over V), if and only if every $v \in V$ is conditionally independent, given its parents $\prod (v)$, of all its non-descendents; in other words,

Probability of v given its parents and non-descendents = probability of v given its parents only.

"A Bayesian belief network of P is a DAG (V,E), such that (V,E) is a minimal independency map of P, augmented with a set of conditional probability distributions $\{P_v : v \in V\}$ *where each* P_v *is a local probability distribution which specifies the probability of each possible instantiation of v given every possible instantiation of its parents."* Pearl (1988)

However, if we assume only binary valued random variables (boolean variables), it is enough to specify for each variable the probability of TRUE, since the sum of both probabilities must be 1.

An instantiation or a full assignment A of a binary-valued belief network, is an assignment that assigns a truth value to each member of V. A partial assignment e is an assignment to a subset of the nodes.

Based on the assumption that the belief network is an independency map of the network variables, the joint probability of any given full assignment can be computed as the product of the probabilities of every variable given its parents, i.e.

$$P(v_1,v_2,\ldots v_n) = \prod_{i=1}^{n} (P (v_i \mid \Pi (v_i))$$

The following example follows (Russell & Norvig, 1995).

You have a burglar alarm that goes off when it detects a possible burglary, and also occasionally when an earthquake happens. Two neighbors John and Mary promise to call you at work when they here the alarm. John may sometimes confuse telephone ringing with the alarm, and Mary may some time miss the sound of the alarm because she likes to listen to loud music. This information can be represented in the following BBN graph.

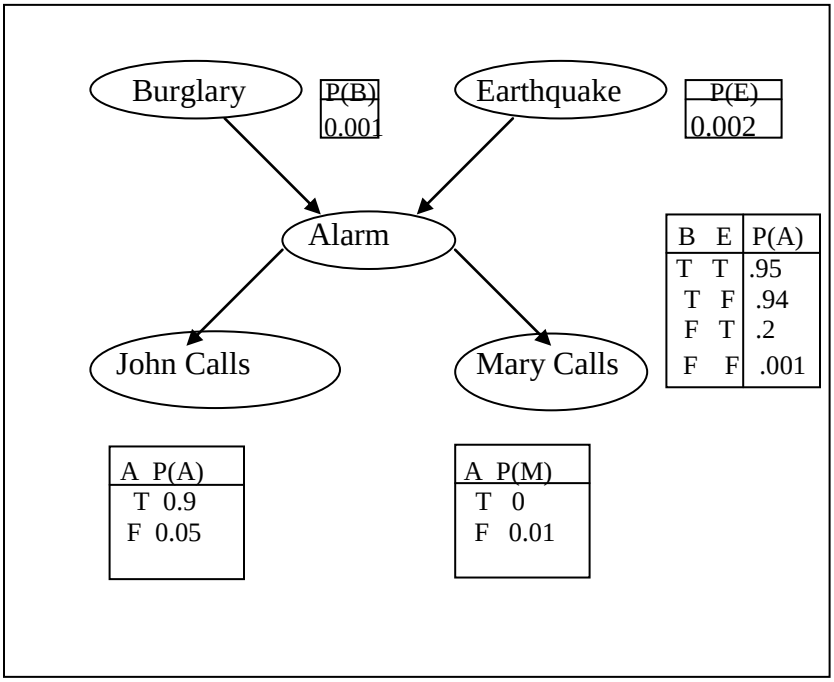


Figure 4. 1 Belief Network Example

4.3.2 Inference in BBNs

A BBN is a natural way to represent conditional independence information. The links between nodes represent the qualitative aspects of the domain, and the conditional probability tables represent the quantitative aspects. It is a complete representation for the joint probability distribution for the domain, but it is often exponentially smaller in size.

Inference in BBNs means computing the probability distribution of a set of query variables, given a set of evidence variables, i.e. P(Query/Evidence). For example, we can use the previous BBN graph to infer the probability of burglary knowing that both John and Mary had called. Any variable in the network can serve as either a query or evidence.

BBNs are capable of several types of inferences. They can be used for diagnostic inference, which allows calculating the probability of a cause given the effect, such as P(Burglary/JohnCalls). They can also be used in causal inference which allows calculating the probability of an effect given the cause, such as P(JohnCalls/Burglary). They are also capable of making inter-causal inference between causes of a common effect, such as P(Burglary/Alarm ^ Earthquake), and finally it is capable of mixed inference where two or more types of the above are combined.

Besides answering query variables, BBNs can help in making decisions based upon the probabilities of the variables in the network. For example, a decision about a certain treatment method can be made based on a diagnostic network, which identifies probabilities of certain related diseases and symptoms. They can also be used to decide what additional evidence should be observed in order to reach more useful inferences from the network. In addition, they can aid in sensitivity analysis in order to identify what

variables of the domain are more important than others in calculating query probabilities, and thus should be more accurate. Finally, they also help in explaining the results of probabilistic inference to the user.

Complexity of inference in belief networks depends on the network structure. In general, computation time for inference in singly connected networks, in which there is at most one directed path between any two nodes in the network, is linear in the size of the network.

A Multiply connected network, on the other hand, is a graph in which two nodes are connected by more than one path. This could happen when there are two or more possible causes for some variable, and the causes share a common ancestor, i.e. when one variable influences another through more than one causal relationship.

In general, exact inference in multiply connected networks is *NP-hard*. This is easy to prove by observing that that a general belief network can represent any propositional logic problem (if all probabilities are 1 or 0), and propositional logic problems are known to be NP-complete.

4.3.3 The MAP Problem

Explanation or finding causes for observed facts (evidence), is frequently encountered in the field of artificial intelligence. For example, natural language understanding may be seen as finding the facts that would explain the existence of the given text. In medical diagnosis if certain symptoms were observed, we would like to find the disease or diseases that explain the observed symptoms. In computer vision or image understanding, we would like to find the objects that explain the given image.

Cost based abduction attempts to find the best explanation for a set of facts by finding the minimal cost proof for the facts. The costs are computed by summing the costs of the assumptions necessary for the proof plus the cost of the rules. Charniak (1994) proved the equivalence of the cost minimization problem to the Bayesian Belief Network MAP (Maximum a-posteriori) assignment solution of the system. also known as the Most Problem Explanation problem (MPE). As the name indicates, solving this problem helps us to find the most probable or logical explanation for a set of observed evidence, by finding appropriate values for variables in the network.

More Formally, in the MAP problem we are given a bayesian belief network B and a partial assignment e of B, which represents a set of evidence for which we seek an explanation. It is required to find the instantiation A with the maximum probability $P(A \mid e)$. This is equivalent to maximizing $P(A)$ under the constraint e. If the evidence set is empty, then we want to find an instantiation with a maximum unconditional probability.

In the MAP problem we try to find an assignment that maximizes the joint probability. In case that we have a certain evidence set then some variables have certain fixed assignment, and the instantiation that we choose must respect this assignment. For example, in the previous BBN graph one instantiation could be:

B = T, E = F, A = T, J = T, M = F

And the joint probability would be:

P(B=T) × P(E=F) ×P(A=T/B=T&E=F) ×P(J=T/A=T) ×P(M=F/A=T) =

0.001×0.998×.94×0.9×0.3= 2.5329 e-4.

According to the above discussion of multiply connected BBNs, the MAP problem is *NP-hard*. Moreover, approximating it with a bounded degree of accuracy have

been proven to be *NP-hard* (Abdelbar & Hedetniemi, 1998) The complexity of the problem increases with the number of variables in the system, the number of states per variable and the number of undirected cycles in the network. Exact inference in such complex cases may not be feasible, and approximate methods become the natural alternative.

4.3.3.1 Applying Genetic Algorithms to the MAP Problem

GAs is one technique that can be used to provide approximate solutions to combinatorial optimization problems. The first attempt to use GAs to solve the MAP problem on BBNs seems to be the work done by Rojas-Guzman & Kramer (1993). The aim of the research was to examine the potential of using GAs in obtaining near optimal solutions for large and multiply connected BBNs.

The genetic algorithm presented in this work uses non-binary alphabet to represent individual solutions. Each chromosome is represented as a graph instead of a string. This is because it is desirable to encode node neighborhood in the chromosome. Each node in the graph represents a gene that corresponds to one variable in the belief network. Each gene can take a number of discrete values that a variable can assume in the network.

The fitness function of each chromosome is the absolute probability of each possible solution. The fitness is a product with one factor for each node, each factor is either a prior probability (for root nodes), or a conditional probability (for internal and leaf nodes).

Crossover is achieved when new individuals are created by combining the chromosomes of their parents. Parents are selected among the best found in the previous

generation. Since individuals are represented as graphs, a cluster, which is a subset of nodes is interchanged. A cluster is defined by choosing a random node as the center of the cluster, and then choosing all nodes that fall less than *N* links away from the center, where *N* is a user defined constant.

Mutation is performed by changing the value of one gene, which represents a value that a certain node can assume. However, variables that are assigned known values at instantiation time, do not change their value by mutation, to guarantee that all individuals retain legal and meaningful representation. These variables represent the evidence set for which we seek an explanation.

To test the performance of the algorithm four networks of different sizes and connectivity were used. The first network BBN1 is a 13-node singly connected network. The second network is a 20-node multiply connected network with 5 undirected cycles. The third network BBN3 is a simplification of BBN2 with only one cycle. The fourth network has no links among variables but has the same search space size as BBN3.

The optimal solution for the first 3 networks was found using exhaustive search of all possible combinations of values. The optimal solution of the fourth network was calculated as the largest prior probability of each node.

The results obtained by the running the GA on each network 135 times were promising. For the most difficult network (BBN2) the optimal solution was obtained 30% of the runs. A solution among the 10 best was obtained 60% of the time, and a solution among the best 50 was obtained in 100% of the runs. Each GA run took less than one minute, as opposed to 70hrs for obtaining the optimal solution by exhaustive search. For

the simple network BBN4 with no arcs, the optimal solution was found in 100% of the runs.

The effect of the network structure on the results was also interesting. According to the authors, theoretically, the GA should not be affected greatly by the number of cycles in the network. The more profound effect is expected to come from the degree of connectivity of the network. This is not surprising if one recalls that the most difficult problems for GAs are the ones that have large variable (gene) interaction, called Epistasis in GA terminology. This expectation was supported by the results obtained. It was clear that the extreme case of 0 arcs (BBN4) was easy to solve and a greedy algorithm would be probably more efficient than a GA. Zero epistasis would occur in a network without links, while high epistasis with each node directly connected with all other nodes. Fortunately, a BBN is seldom fully connected, and gene interaction is limited to immediate neighbors. This characteristic supports the notion of small compact blocks that helps to guide the search towards the optimal solution, and thus making a GA approach attractive over a greedy algorithm.

Theoretically, it appears that gene location in the string may have an effect on the results. It may be desirable to locate neighboring genes in the network close to each other in the chromosome representation. Experiments are required, however, to test the effect of this representation.

According to the authors, further research is required to determine whether the proposed approach will prove practically useful to build decision support tools to diagnose and manage complex systems. Future research should concentrate on coupling the GA with another local search technique that can be used to refine the near optimal

solutions obtained by the GA. In addition, the approach should be tested on larger networks with different degrees of connectivity, and should be compared with other approximate methods.

Chapter 5

5 Literature Review

5.1 A Comparison between GA and SA

Simulated annealing and genetic algorithms are two very similar techniques. The concept of both is borrowed from nature. They both work well on a variety of problems and require little problem specific information. They both require some criterion of determining the fitness or the cost of a solution, and they both create random solutions in the search space, and move from one solution to another probabilistically.

Both techniques are not guaranteed to give optimal results. SA possesses a formal proof of convergence. The convergence of a SA algorithm can be controlled using the annealing schedule to produce a sufficient number of iterations that will lead to slow cooling and finally to asymptotic convergence. GAs, on the other hand, do not possess such proof of convergence. Their convergence cannot be easily controlled since they are heavily dependent on random operators, and they work on a large number of individuals.

In both algorithms good solutions may be discarded. In SA, this happens when a new structure is accepted and the old one is discarded even in the case that the old one is better. In GAs, this happens because a GA always accepts new solutions resulting from genetic operations. This characteristic may cause disruption where good solutions are discarded or damaged preventing optimal performance.

Another important difference between SA and GA is the ease with which each method can be parallelized. GAs are easy to parallelize because they operate on a

population of individuals that can be evaluated and processed in parallel. SA, on the other hand, works on a single solution at a time. It moves from one solution to the next in a sequential manner; thus it is not easy to parallelize. In addition, the population of individuals gives GAs useful redundant information about what it has learned from previous searches. Critical components of past good solutions can be captured and combined together via crossover to give better solutions and better exploration of the search space. On the other hand, SA does not posses such memory of the past since it operates on only one solution at a time and exploration is limited to the immediate neighborhood.

Besides explicit parallelism mentioned above, GAs also possess an important feature called implicit parallelism. In short, this means that while a GA operates on an individual encoded by a string, this string actually represent a group of individuals who share with the string its schemata, or its basic features. Both Implicit and explicit parallelism help to achieve super linear speedups when GAs are parallelized. (Chen et al., 1998; Goldberg & Mahfoud, 1993)

5.2 Previous Work

Combining GAs and SA is an attractive area of research. GAs are naturally capable of exploring wide areas of the search space, since they operate on a large population of solutions in parallel. The crossover operator allows large jumps in the solution space by combining two solutions. The mutation operator on the other hand performs one small move on one individual. As a result, GAs are not capable of fine tuning a good solution, because they have no method of performing several small moves on a solution. SA, on the other hand, possesses this characteristic. SA is capable of fine

tuning a good solution because it produces a sequence of small moves that usually result in an improvement of the current solution.

Thus, a combination of GAs and SA may allow the benefits of both to be realized especially in solving difficult problems such as combinatorial optimization problems. In fact, there are several attempts in the literature to hybridize GAs and SA. In general, research in this area can be classified into four main categories.

The first category tries to use simulated annealing to improve the quality of the solutions obtained by genetic algorithms. This category tries to carry over to the basic genetic algorithm the fine tuning features of SA, thus a good solution obtained by GAs can be further refined to obtain better and closer to the optimal solution. examples of such research are the Boltzmann Darwin Strategy (Boseniuk & Ebling, 1988), the SAGA algorithm (Brown et al., 1989), the UFO algorithm (Abdelbar & Hedetniemi, 1997), and SA as a genetic operator (Abdelbar & Attia, 1999).

The second category tries to carry over to simulated annealing the population oriented feature specific to GAs. Some research belonging to this class strives to carry over to GAs the asymptotic convergence properties of SA, by having a population of solutions which have a distribution that is provably near Boltzmann (Goldberg, 1990; Mahfoud & Goldberg, 1993). Other research is more concerned with finding the optimal annealing parameters in a population oriented fashion. In (Lin et al., 1991) The population serves as parallel Markov chain, while in (Cho & Choi, 1998) the population is a just collection of solutions, each has its own annealing temperature that is adjusted according to its rank in the population.

The third category tries to augment the basic simulated annealing algorithm with the GA recombination or crossover operator. The main idea is that when SA reaches stagnation, a large jump in the solution space is introduced using the GA recombination operator. This will help the algorithm to perform a better exploration of the search space, which is a characteristic of the GA not usually enjoyed by regular SA. This category includes the work by (Koakutsu et al., 1996)

The fourth category includes research that tries to introduce the SA acceptance probability, based on the annealing schedule, to the genetic algorithm operators. The main idea is convert the usual replacement strategy of GAs, in which offspring replace parents irrespective of their fitness, to a SA controlled replacement strategy. Doing so, superior children have a higher chance of replacing their parents, while inferior children still have a non-zero chance of replacing their parents. This category includes the work by (Esbensen, 1992) , (Adler, 1993) and (Chen et al., 1998).

Following, a summery of the research done in each category is provided.

5.2.1 Using SA to Improve Solutions obtained by GAs

5.2.1.1 Boltzmann-Darwin strategy (Boseniuk & Ebling, 1988)

This research solves the traveling salesman problem using a mixed Boltzmann-Darwin strategy; i.e. it combines SA with GA while introducing the idea of life cycles.

The general scheme depends on having N tours each one can be represented in the population by one or more copies of itself. Each tour can change its internal structure by Boltzmann type mutations with temperature T^k, where k is the current stage of the tour. Each tour has a lifetime consisting of at least two stages, childhood and maturity. The child hood stage starts at a high temperature, and the maturity stage is reached when

the temperature reaches zero, i.e. the life cycle of a tour is similar to a short annealing procedure. When a tour reaches maturity it is able to reproduce itself by making an identical copy of itself that replaces of the worst tour in the system.

Experimental results showed that the mixed strategy yields, in special cases, better results than pure simulated annealing. In addition, the algorithm is well suited for parallel implementation.

5.2.1.2 The SAGA algorithm (Brown et al., 1989)

The research introduced here tries to solve the Quadratic Assignment Problem using a hybrid SA-GA approach. The QAP is an *NP-hard* problem, that is usually represented by two *N*N* matrices *C* & *D*. The matrix *C* is called the structure or the cost matrix, with entries c_{ij}. The matrix *D* is called the distance matrix with entries d_{ij}. We want to find a permutation *P* of the *N* indices, such that

$$\sum_{i=1}^{N} \sum_{j=1}^{N} c_{ij}\ d_{ij} \quad \text{is minimized}$$

The TSP is a special case of the QAP in which the desired structure is a cyclic permutation of the cities. The QAP is a good candidate for the application of the adaptive concepts employed by both genetic algorithms and simulated annealing. The idea was that after identifying the promising regions of the search space by the GA, SA is invoked to optimize members of the final population. The hybrid Simulated Annealing – Genetic Algorithm approach is called (SAGA). This technique is useful for improving the quality of the solution obtained by GAs, using a local search method such as SA, which will perform a fine tune of the solution. In other words, each offspring is required to "mature" before it is allowed to reproduce.

The following are the steps performed by the SAGA algorithm

1. Initialize the parameters of the GA.
2. Generate an initial population of solutions for the GA.
3. Use the GA to produce k good solutions.
4. For each of the k solutions, do the following
    a. initialize the parameters of SA
    b. improve the "good" solutions using SA, and return to the GA population

5. Repeat steps 3 and 4 as needed.

The algorithm was better improved by applying step 4 in parallel for the k solutions.

In SAGA the selection operation is a greedy operator that selects the first parent from the best *s* structures where *s* is a user defined constant, while the second parent is selected at random

Crossover is similar to partially matched crossover, in which a portion of the structure of one parent is copied directly to the offspring, and the rest is copied from the second parent resolving conflicts.

Ex: P1: 1 2 3 4 5 6 7 8 P2: 6 4 8 3 7 2 1 5

Step 1: O: - - | - - - | - - -

Step2: O: - - | 3 4 5 | - - -

Step3: O: 6 - | 3 4 5 | 2 1 -

Step4: O: 6 8 | 3 4 5 | 2 1 7

To find an appropriate annealing schedule for the SA operator, the following heuristic was applied.

1- An average change in cost (Δ) was calculated by applying random pair wise interchange of the terms of the solution, and computing the corresponding change in cost. This method is one of the most commonly used heuristics for solving the QAP. The mean absolute deviation (MAD) of 100 pair wise interchanges was calculated.

2- To calculate the initial temperature T_0, a certain initial probability of acceptance β was assumed. T_0 was then set to $T_0 = \beta \times MAD$.

3- The decrement constant α used to decrement the temperature value was set to $\alpha = [T^* / T_0]^{1/\eta}$ where T_0 is the initial temperature, T^* is the final temperature and η is the number of temperatures in the schedule.

The algorithm was tested with two standard problem sets found in the literature. Results indicated that SAGA was superior in solution quality to steepest descent pair-wise interchange method, which is the most important heuristic search method to solve the QAP. However, the run time of the SAGA algorithm was not as good as its competitor for small problems. This situation is reversed for large problems. Based on these results, SAGA has excellent potential for solving large-scale QAP applications.

According to the authors, future research should be directed to solving the real problems with the algorithm. In addition, more tests should be done to improve the quality of the SA and GA parameters used in the algorithm.

5.2.1.3 The UFO algorithm (Abdelbar & Hedetniemi, 1997)

In this paper a hybrid of GA and SA was used to solve the MAP problem on BBN's. The algorithm is similar to the SAGA algorithm, but it is more "loosely-coupled", since not all newly produced offspring from the genetic algorithm undergo simulated annealing.

The algorithm was implemented on a multiprocessor system where one processor -called the kernel- was used to run a regular genetic algorithm. The other processors are called satellite processors, and each one is used to apply simulated annealing on a chosen individual from the population. The idea is that periodically some individuals undergo a process of genetic improvement for whatever reason before being returned to the population, as if it was abducted by UFOs who improve its traits and then return it to the population.

The difference between this algorithm and the SAGA technique is that in the SAGA algorithm every new offspring resulting from a crossover operator undergoes a SA process. In the UFO algorithm whenever an SA processor finishes its current search, it requests another individual from the satellite kernel processor.

The algorithm was implemented on PVM a public domain SW package that allows a network of heterogeneous Unix machines to be used as a single large parallel Computer.

The results indicate that the hybrid method performs better that either GA or SA alone, even disregarding the effect of parallelism. The problem in the algorithm is that SA has a much higher cost in terms of processing time, when compared with the regular random mutation operator. A single application of the annealing operator can take as

much time as one or two hundred generations of applying simple random mutation and crossover operators. Thus, the algorithm is particularly suited for parallel processing. Since a satellite processor does not need to communicate with the master processor while it is performing the annealing procedure, the hybrid algorithm is more suited for application on loosely coupled, distributed multi-computers with high communication cost.

5.2.1.4 Simulated Annealing as a Genetic Operator (Abdelbar & Attia, 1999)

The focus of this research was on the potential for obtaining better solutions by supplementing a genetic algorithm with a simulated annealing operator as a type of intelligent or directed mutation operator, as distinguished from the random or undirected regular mutation operator.

The regular mutation operator is very cheap computationally. It introduces diversity in the population and allows slow evolution to be achieved in a large number of generations. Simulated annealing as a genetic operator allows occasional modification of some individuals in the population in an intelligent and directed manner. Thus, it can be seen as modeling a situation where selected members of the population undergo "fast-track" evolution for whatever reason, before being admitted to the population.

The problem that was solved using this technique is the MAP problem on BBNs (Maximum A Posteriori Assignment Problem). The implementation used the popular MIT class library GALIB. The genetic algorithm used was a steady state GA with a population replacement factor of 10%. The chromosome representation was a binary string representing a candidate truth assignment to the underlying BBN.

The crossover operator was a standard one-point crossover operator with probability P_{cross} of being performed, and the mutation operator selected a random bit to flip with probability P_{mut}. Selection for reproduction was based on uniform random distribution. The objective function is the joint probability of the underlying BBN, given the truth assignment and the conditional probability table of each node in the network.

Simulated annealing was implemented as a special case of mutation. A call to mutation may cause simulated annealing to be performed on the individual with a certain predefined probability P_{SA}. The solution resulting from SA is inserted in the population as the result of mutation. If SA is not performed, regular mutation is performed with probability $1\text{-}P_{SA}$. The SA operator used a fixed initial temperature *T0*, and the temperature was reduced by a certain factor *f*. The neighborhood operator used to find neighboring states for annealing was the regular mutation operator of the GA.

The algorithm was tested on 30, 40 and 50-node networks with random probabilities uniformly distributed between [0,1]. First, A regular GA was run to convergence 200 times, with different randomly generated values of P_{cross} and P_{mut}. The best 10 parameters pairs were used to test the Genetic algorithm augmented with SA. For each pair, 200 P_{SA} values were tested. Results indicate that for every one of the genetic parameter pairs, simulated annealing produced an improvement in the quality of the solution returned. However, the effect of mutation seems to be positively related to the ratio of P_{cross} to P_{mut}. When this ratio was highest, the effect of simulated annealing was more profound. In addition, the performance of the algorithm was better for the 50-node network than for the other smaller networks.

5.2.2 Population Oriented Simulated Annealing

5.2.2.1 Boltzman Tournament selection for Genetic algorithms (Goldberg, 1990)

In this research a Boltzmann tournament selection procedure is derived and implemented to give stable distributions within a population of structures that are provably near Boltzmann. This is basically a means to carry over Boltzmann distributions and cooling schedules to GAs, thereby guaranteeing asymptotic convergence to a population oriented structure as well.

The algorithm depends on selecting an individual from the current population. Two other individuals are then selected to compete with the first individual. The selected individuals are chosen according to the differences between their objective functions and the objective function of the first individual. A tournament like competition in two stages is held between the three individuals. This competition uses logistic acceptance probability $exp(-Ei/T)/ (exp(-Ei/T)+exp(-Ej/T))$, where i & j are the two competing individuals. The winner of the tournament finds its way to the new population, and the process is repeated until the new population is full.

The algorithm is strait forward except that in the first stage of selection between two individuals an anti-acceptance probability is used, which favors poorer individuals. The objective of this is to help the population achieve a Boltzmann distribution stably. If the better individual is always favored, as in the usual acceptance probability, the best individual will ultimately fill the population with copies of itself, in a manner similar to other GA selection scheme. This is particularly true in case of repeatedly comparing an individual to a copy of itself. Using the anti-acceptance probability helps the population

equivalent of generating a neighbor uniformly at random. Choosing competitors that have different function values also helps to achieve this objective.

One advantage of the above technique is that it is easily implemented on parallel hardware, unlike the regular SA procedure which is difficult to parallelize.

5.2.2.2 Parallel Recombinative Simulated Annealing (Mahfoud & Goldberg, 1993)

This research provides a parallel version of simulated annealing that strives to retain the desirable asymptotic convergence properties of SA, while adding the population approach and the regular GA crossover and mutation operators. It is a parallel version of the research summarized in the previous section.

In this algorithm several copies of SA run in parallel with mutation as the neighborhood operator and crossover combining individual solutions. Alternative solutions in PRSA, unlike Boltzmann tournament selection, do not come purely from the current population, but from applying both crossover and mutation.

In PRSA, cooling is synchronized across processors. Thus a Boltzmann distribution is approached on every processor. The combined distribution also approaches Boltzmann. Crossover reconciles independent solutions, and together with mutation play the role of an extended population-level neighborhood operator between independent SA. Slow cooling and diversity maintaining operators help PRSA to avoid genetic drift and premature convergence.

The following algorithm is run on every processor:

1- Set initial temperature to a sufficiently high value.

2- Initialize a random population.

3- Repeatedly generate each new population from the current population as follows:

Do n/2 times (n is the size of the population)

a. Generate two children using crossover.

b. Mutate every newly generated child.

c. Hold one or two Boltzmann trials between children and parents.

d. Overwrite the parents with the trial winners

e. Periodically lower T.

A Boltzmann trial refers to using the logistic probability $1/(1+exp(\Delta E/T))$ to select between an old and a new solution. One possible way to perform competition between two parents and two children is to allow both parents to compete as one unit against the two children, this is called double acceptance/rejection. A second approach called single acceptance/rejection allows a parent to compete against the child that inherited its right end.

The research proves the asymptotic convergence property for two versions of the algorithm. The two versions differ in the selection and replacement criteria, and in the way a competition is carried out between parents and children. The proof depends on proving the two conditions that suffice to guarantee asymptotic convergence

1- The ability to move from any state to the optimal solution in a finite number of steps.

2- The probability at any given temperature for generating state y from state x, is the same as the probability of generating state x from y.

Applying PGSA on two deceptive test problems, showed that the algorithm consistently converged to the global optimum at all population sizes. In addition, both serial speedup and combined speedup (from implicit and explicit parallelism) can be polynomial.

5.2.2.3 The Annealing Genetic Approach (Lin et al., 1991)

The main purpose of this research is to design an efficient annealing schedule for simulated annealing. Genetic algorithm's parameters were applied to find good annealing parameters.

In designing a simulated annealing operator, the following factors must be carefully analyzed and optimized:

1. The optimum initial temperature value.
2. The length of the Markov chain at each temperature, which implies the number of iterations that should be carried out at each temperature.
3. Detecting the equilibrium condition of the system at each temperature.
4. Finding methods to help the system escape local optima.
5. Finding a suitable temperature reduction factor.
6. Detecting that the system has reached stagnation, and no further improvement is possible.

In order to find a good initial temperature, the annealing genetic approach performed preprocessing calculations of the over all algorithm. The following steps were conducted:

1. Create an initial random population P_0
2. Apply regular genetic operators on the population once to produce an intermediate population.
3. Take the worst individual in the intermediate population and apply a Markov chain on it. If the resulting individual is better, insert it into a new population called P1, and continue to generate other individuals using the same Markov chain. If the new state is

worse than the initial state, stop this Markov chain, select another individual from the intermediate population and repeat the same process.

4. Stop when the new population P_1 has been completely generated.

In order to calculate the initial temperature value, the researches assumed an initial acceptance probability of *0.6*. The change in cost *(ΔC)* was taken to be the difference between the highest and lowest cost of the final population. The initial temperature can thus be calculated from the formula: *Paccpet = exp (-ΔC/T)* as

$T = -\Delta C / \ln 0.6 \approx 2\Delta C$. The initial temperature was then taken to be

$$T_0 = \Delta C / (pop\_size/2).$$

The second important parameter is the length of the Markov chain. From the above procedure it is clear that the length of the Markov chain is bounded by the population size, because the chain was generated from multiple states of the population.

For the new population to achieve a quasi-equilibrium state at each temperature, regular genetic operators (crossover and mutation) were applied on the population to create a new population.

In order for the system to reach convergence, the average cost of the population at each generation should not exceed that of the population in the previous generation. The condition for achieving convergence was determined by keeping the cost of the best so far solution at each generation. When 80% of a new population in a certain generation have their costs equal to the best solution found so far, convergence is declared and the genetic annealing algorithm stops.

The value of the best decrement factor was calculated adaptively. When the difference in cost between two successive generations is high, α was taken to be 0.5,

which represents fast annealing. When the difference in cost is low, α was set to 0.95, which represents slower annealing.

The annealing genetic approach was tested on the traveling salesman problem. The results indicated that the algorithm could actually find results closer to the best known solution than SA alone. The authors also proved that their algorithm is of $O(n^2)$. Their proof, however, is problem specific and difficult to be applied to different problems. Although the experimental results did not show a great enhancement in the solution quality when compared to the best known solution, the algorithm actually reduced the amount of time needed to find an approximate solution.

5.2.2.4 NPOSA A New Population Oriented SA (Cho & Choi, 1998)

This is a new algorithm that introduces the evolutionary concept to SA. The idea is to have a population of solutions each has its own local temperature. The local temperature is used as usual to improve the solution using SA. The temperature is also adjusted according to the individual's rank in the population. If an individual finds that its cost is high compared to others in the population, it raises its temperature to give itself a chance to improve through uphill moves. Otherwise it drops its temperature to avoid uphill moves and allow more fine search, i.e. higher temperature is assigned to an inferior individual with a high cost and vice versa.

The advantage here is that the temperature need not be defined explicitly by the user, since it is adjusted implicitly using the individuals rank. This also has the advantage of making the algorithm less sensitive to the initial temperature value as is the case in regular SA in which the cooling schedule parameters have to be carefully determined, usually by trial and error, because they highly affect the final solution.

The algorithm was applied to the 100-city TSP problem and compared with the regular SA results on the same problem. It was found that NPOSA obtains better solutions closer to the optimum than SA.

5.2.3 Augmenting SA with GAs Recombination Operator

5.2.3.1 Genetic Simulated Annealing GSA (Koakutsu et al., 1996)

This technique is an SA-oriented hybrid approach which tries to incorporate the GA based crossover operator into SA in order to produce large jumps in the solution space and enlarge the search region. GSA generates the seeds of SA sequentially, i.e. the starting solution of a SA local search depends on the best so far solutions of all previous SA local searches. This sequential approach seems to generate better child solutions than a its parallel counterpart in which the seeds of SA local search are generated in parallel, and the order of applying each SA local search is independent.

Initially a population is created at random and then three operations are repeatedly applied on the population. SA-based local search, GA based crossover operation, and population update.

While performing SA search, GSA keeps the best so far solution found. At the end of each SA based local search GSA replaces the current solution with the best so far solution.

When the system reaches a frozen state, a jump in the search space is introduced by performing GA based crossover on two randomly chosen individuals from the population. The new solution produced by the crossover operator undergoes a full annealing process before being inserted into the population. At the end of the SA search,

the new solution replaces the worst solution in the population. The process repeats for a predefined CPU time.

The algorithm was applied to the non-slicing floor plan design problem which is defined as "given a set of arbitrary shaped and fixed sized modules and connection information, find a minimum area placement with the shortest wire length". By comparing the results with SA, it was found that that GSA improved the average chip area by 12.4% and the average wire length by 2.95%.

5.2.4 Introducing SA Acceptance Probability to GA operators

5.2.4.1 Parallel Genetic Simulated Annealing (Chen et al., 1998)

This technique is a parallel version of GSA. It tries to combine the parallel features of genetic algorithm with the selection criterion and the convergence property of SA. It is a massively parallel algorithm suited for implementation on MIMD (Multiple Instruction Multiple Data) machines.

The algorithm works by setting a temperature value for each processor, and then having each PE (Processing Element) create a random solution. Then, for a certain number of iterations the following is performed: this processor receives a new individual from another processor. The second processor is determined using two random values (direction and distance) created by the first processor at random. Mutation is performed on the resident individual and crossover is performed between the resident individual and the new one to create two children.

The SA part now comes to work. Selection is performed between three individuals, the resident and the two children, in a tournament fashion. The selection criterion works in a manner similar to SA, in which a new candidate is accepted if the

cost increase is less than or equal to the current temperature. The winner of the tournament replaces the resident individual, and finally the temperature is reduced.

The basic contribution of this method is exploiting the parallel nature of GA by having each individual reside on a different processor. At the same time no parallelization of any serial or problem specific portion of GA or SA is required. The calculations required for the objective function, mutation, crossover, and the selection criterion are replicated on each processor. Thus, communication and synchronization between processors is minimized.

Another new idea introduced here is the random temperature approach as opposed to the fixed temperature approach. For each PE, the initial temperature is computed in a function, which samples the effect of operators in the domain to calculate an expected change in cost. Initial temperature value is calculated such that the probability of accepting an uphill move is between 0.5 and 1. The α parameter is calculated from a randomly generated final temperature, the initial temperature, and the maximum number of iterations required for SA.

The PGSA algorithm was tested on two problems: the traveling salesman problem (TSP), and the error correcting code design problem (ECC).

The results obtained indicated that the random approach to create initial temperature values for each PE separately produces results similar or better than the uniform approach regardless of the population size. The advantage in the random case is that the progress is apparent immediately after the algorithm starts, unlike the uniform temperature case in which little progress is achieved during the initial period due to the high temperature at the beginning of the run. Another advantage is that using this

approach, there is hardly a need for fine tuning the SA parameters. Obtaining the optimal parameters is a critical and very tedious task in traditional SA, because these parameters highly affect the results obtained by the algorithm.

In addition, the random temperature approach allows the algorithm to keep a diversity of annealing schedules. At PEs where the initial temperature, the final temperature and α are high, inferior solutions are accepted at a greater probability. This helps to maintain diversity in the population, which is critical to the performance of the algorithm. On the other hand, PEs with lower temperatures and α accept inferior solutions with a much lower probability. Thus good solutions are protected from disruption. Keeping a balance between diversity and disruption is one of the key features that lead to the success of the algorithm.

The second striking result was that the performance of the algorithm scales up linearly with the increase in processing elements. In TSP domain, the constant of proportionality approaches one for finding a solution within 2% of the optimal. This means that we can reduce the execution time by half if we double the number of processing elements.

Another important finding was that the performance was improved with increased population size. This result appears to be unique to the PGSA technique, because in standard GA systems the performance usually degrades if the population size exceeds a few hundred. A traditional GA uses roulette wheel selection criterion for selecting parents. This approaches allows above average solutions to replicate themselves in the new generation, and low fitness solutions gradually die during evolution. Diversity may be lost very quickly, leading to fast and premature convergence. Increasing population

size does not help in solving the problem. On the other hand, PGSA technique helps to maintain diversity by using SA-type annealing schedule in place of selection, when a choice must be made between parents and their offspring.

5.2.4.2 Adaptive SAGA (Esbensen & Mazumder, 1994)

In this research a mixture of GA and SA is introduced and applied to the Macro cell placement problem. The idea was to bring the fine tuning feature of SA to the genetic algorithm after stagnation, thus combining the benefits of both algorithms. By nature, in a GA the cost of the solution improves rapidly in the initial phase. In the later phase of the process, improvements become very slow, and most run time is wasted trying to achieve very small improvements. SA, on the other hand, does not converge as fast as the GA in the initial phase, but it is usually capable of obtaining improvements faster in the later phase. Combining both algorithms attempts to gain the benefits of both.

The basic GA has been modified in two ways

1- the mutation performed on an individual is accepted with a certain probability as in SA. This probability is determined by the temperature of the individual and each individual has its own annealing schedule.
2- Initially the algorithm behaves as a regular GA. As time goes on ,however, the GA is switched to SA as a result of stagnation

The algorithm keeps track of the best individual ever found. It performs regular crossover on two individuals chosen from the population according to their fitness values. Mutation, as said previously, is SA controlled. A mutation that increases the individual fitness is always accepted, while a mutation that decreases fitness is accepted with a

probability proportional to the change in fitness. The temperature of the individual is then modified according to its own annealing schedule.

A step towards SA is taken whenever no improvement has been accomplished for a number of generations. This is achieved by gradually reducing population size and increasing the probability of mutation, i.e. more SA controlled mutations will be performed on a smaller number of individuals. Finally the algorithm will behave as a pure SA when the population size reaches one.

The algorithm was tried on the macro cell placement problem. The problem can be defined as follows: Given

- A set of rectangular cells, each with a number of terminals at fixed positions along the edges of the cell.
- A net list specifying the interconnections of all terminals, and
- An approximate horizontal length W of the chip under construction.

It is required to find

- The position of each cell.
- The orientation and possible reflection(s) of each cell.
- A rectangle B defining the shape of the chip. The objective is to find B, such that B has the minimum possible area, satisfying the following constraints:

1- No pair of cells overlap each other

2- The rectangle B encloses all cells and has approximate horizontal length W.

3- The area within B, which is not occupied by cells, is sufficiently large to contain all routing needed to implement the inter-connections.

The genotype was encoded using a binary tree in which the i^{th} node corresponds to cell i. Two types of edges were used, top edges and right edges. All edges are directed and oriented away from the root. Crossover and mutation operators were specifically designed for the problem such that the resulting offspring should not violate the problem constraints.

The algorithm was tested on Apte, Xerox and Hp benchmarks from the 1992 MCNC International Workshop on Placement and Routing described in (Esbensen, 1992). It was found that the combined approach actually performs better than a pure GA, in terms of solution quality, on the first two test cases. For the Hp benchmark, no significant improvement was obtained. A comparison with other systems used to solve the same problem also indicated that the SAGA technique was superior.

5.2.4.3 Simulated Annealing Mutation and Recombination (Adler, 1993)

This technique augments the mutation and recombination operators of GAs with a SA-like acceptance probability scheme. The basic idea is to use the SA stochastic acceptance function internally to limit adverse moves.

The simulated annealing mutation operator (SAM) performs regular mutation on a candidate solution. The SA part is introduced by applying an acceptance trial with a temperature schedule between the parent and the child produced by mutation.

The simulated annealing recombination operator (SAR) works in a similar manner. The acceptance trial is held between the two parents, and then between the winner and each of the two children resulting from recombination. The winning two individuals from this acceptance trial is inserted in the population.

The SAM and SAR operator do not affect the GA convergence, since the schemata theorem does not assume any specific behavior on the genetic operators. The hybrid technique can also be seen as introducing the population notion to the regular SA operator. In addition, the convergence properties of SA are also not affected by the hybrid algorithm, since each individual in any generation is the successor of another member in the previous generation, which maintains the Markov chain model within GA domain.

The algorithm was tested on the problem of training a feed forward neural network. The objective function was the mean squared error of running the network forward with the given weights. The performance of the hybrid technique outperformed pure GA by an order of magnitude.

5.3 A Note on Adaptive Simulated Annealing

The researches summarized above include many attempts to adapt the basic simulated annealing algorithm. The main objective is to optimize the SA parameters in an adaptive manner, without the need for the manual adjustment of operators through trial and error. For example, the work done by Esbensen & Mazumder (1994) uses a SA controlled mutation, in which each individual has its own annealing schedule. The initial temperature is calculated for each individual using a probability of acceptance that changes adaptively during the course of evolution depending on the number of mutations performed on the individual.

The work by Cho et al. (1998) also uses a different annealing schedule for each individual. The temperature value is adjusted for each individual depending on its rank in the population. This will give poorer individuals more chance of improving through simulated annealing.

The PGSA approach (Chen et al., 1996) uses different annealing schedules for each processing element. In addition, the initial temperature value is created using a randomly generated probability of acceptance. The reduction factor α is also calculated adaptively using a randomly generated final temperature, the initial temperature, and the desired number of iterations. As explained above, varying the annealing schedule between processors and using adaptively generated parameters allow the balance between disruption and diversity to be maintained.

Finally, the annealing genetic approach (Lin et al., 1991) also utilizes a heuristic method to calculate the initial temperature value, using a certain initial probability of acceptance, an expected change in cost, and the current population size. In addition, the reduction factor α is set adaptively such that fast annealing is performed when only a quick and dirty solution is required, while slow annealing is performed when convergence is approached and more fine search is needed.

Adaptation in simulated annealing seems to be an attractive area of research. The reason is that SA suffers from a major drawback, which is its sensitivity to the annealing parameters. Finding optimal annealing parameters is by no means an easy or strait forward task. Therefore, adaptive simulated annealing, in which the annealing parameters are optimized during processing depending on the current situation, seems to be the natural alternative.

Chapter 6

6 Adaptive Simulated Annealing as a Genetic Operator

6.1 Motivation

The technique presented in this research can be described as a hybrid genetic-annealing technique, in which simulated annealing acts as a directed or intelligent mutation operator. Although the feasibility of this approach has been established in a previous work (Abdelbar & Attia, 1999), there were still many open questions that need further investigation.

The theory behind the hybrid GA-SA technique, in which SA plays the role of an intelligent or directed mutation operator, is based on the idea that SA in this context represents a situation in which some individuals in the population undergo "fast track" evolution for whatever reason. This can also be thought of as helping some newly generated offspring to reach maturity before being inserted in the population. Theoretically, this should allow better solutions to be obtained as a result of directed mutation. The presence of superior children in the population should help the evolution of genetic algorithms, through the regular mutation and crossover operators, towards optimal or closer to the optimal solutions.

The results obtained in the previous thesis work (Abdelbar & Attia, 1999) support this theoretical observation. It was actually found that the hybrid technique yields better results than GA alone when the technique was tested on large size belief networks. It was found, however, that the best results were obtained when the annealing parameters were specifically tailored for the network under consideration. This observation may limit the

use of the technique, since manual adjustment of parameters is by no means easy or strait forward, although it is very critical for the final results obtained.

Designing a simulated annealing algorithm, in which the parameters adapt themselves according to the current situation in the search process, seems to be an attractive alternative that will remedy the defects of the regular static simulated annealing algorithm.

Traditionally, if we were to use GAs to optimize the parameters of any algorithm, we would think of a Meta-genetic algorithm, or a genetic algorithm within another genetic algorithm. The primary GA will have each individual consisting of the parameters that should be optimized. To determine the fitness of each set of parameters (each individual), we would run another secondary complete GA using the parameters of this particular individual. The best result obtained by the secondary GA would be returned to the primary GA as the fitness of that particular set of parameters. This process is repeated for all individuals in the primary genetic algorithm. Of course, applying such technique is very difficult and time consuming.

The concept of GAs and SA are both borrowed from nature. Nature again may provide the solution to our problem. In nature the genotype of an individual may contain genes that do not directly affect the individuals phenotypic characteristics or its performance in life. Nevertheless, these genes affect the offspring produced by that individual. Some genes transmitted from the parent to the child may result in inferior offspring that have some phenotypic defect. Other genes may result in superior children if they were carried over from a parent to a child. Defective offspring do not usually live long enough to transmit the defective genes from one generation to another. These genes

thus have a short life span. On the other hand, superior offspring continue to live and reproduce, and their genetic material spreads and propagates from one generation to the next.

This situation is exactly what we are trying to mimic in this research. The analogy in this case is that the SA parameters play the role of hidden genes that do not directly affect the solution in terms of its fitness. Their role comes to play when offspring of this individual are produced using these parameters. Good parameters result in superior offspring that continue to survive from one generation to another. Bad parameters produce inferior offspring that quickly die out together with their bad genes, parameters in our case.

6.2 Thesis Contribution

The discussion in chapter 5 (section 5.3) shows that several attempts in the literature, concerning hybrid GA-SA techniques, tried to introduce adaptation in the SA operator. These include (Lin et al., 1991; Esbensen & Mazumder 1994; Chen et al. 1996 and Cho et al. 1998). The adaptation of parameters in all these cases depended on a heuristic adjustment of parameters using randomization, or by checking the existence of certain conditions in the search process, such as approaching convergence, number of mutations, rank of individual...etc. Non of these attempts introduced self-adaptation of parameters that evolve and learn from experience without any external guidance or supervision.

Thus, the first contribution of this thesis is that the SA operator will be self - adaptive, and its parameters will evolve and change according to the current situation and the requirements of the search process. This should eliminate the problem of having to

find fixed global annealing parameters that should be general enough to give good results for all individuals in the population, yet specific enough to suit each and every stage of the search. As mentioned above, this task is very difficult and may be infeasible in many practical applications. For example, in some cases, especially at the beginning of a run, we may just need a quick and dirty solution, which means having fast annealing schedule with high reduction parameter α. In other cases, especially when we approach convergence, we might need much slower cooling, in order to achieve better fine tune of the good solutions obtained.

The role of adaptation in this research is not restricted to finding optimal parameters easily, but it also offers a great contribution to the quality of the solutions obtained. In a sense, the optimization of parameters as well as optimizing the solution to the problem will go hand in hand. From the analogy with nature, we can see that as the parameters evolve with time, they can directly affect the quality of the solutions produced. Bad solutions resulting from bad parameters will not live long enough to degrade performance, while good parameters will produce good results that will continue to live and produce other superior solutions.

The second important contribution of this thesis is that the genetic algorithm used is augmented with problem specific knowledge. The problem selected to test the algorithm is the MAP problem on BBNs, described in chapter 4. Utilizing problem specific information is achieved by having a crossover operator that is aware of the structure of the BBN graph. This crossover operator is similar to the one used in (Rojas Guzman & Kramer, 1993). In this technique a random node is selected as a center of the cluster. All nodes that fall within *N* links away from the center node, where *N* is a user-

defined parameter, are selected. This subset of nodes is then interchanged between the two children produced from crossover (see chapter 7 for details).

The advantage of the cluster based crossover operator over the regular one point crossover used in (Abdelbar & Attia 1999), is that the cluster based crossover allows nodes that are directly affected by each other, because they are directly connected, to be transferred together. This should have a better effect on the resulting fitness, because nodes are transferred according to their location in the graph, and not according to their location in the chromosome representation.

Finally, the previous thesis work (Abdelbar & Attia 1999) tested the algorithm on randomly generated large size belief networks, with no particular structure. In this thesis we try to test the performance of the algorithm on graphs that have specific structures. We aim to test the features of the network that directly affect the GA performance. In the work by (Rojas Guzman & Kramer, 1993) it was thought that connectivity has a more profound effect than the number of cycles in the network. The truth of this observation has not yet been established. In this work we try to provide an answer to this open question by testing several networks with different structures. Specifically, we concentrate on networks with a large number of undirected cycles, large connectivity, and large number of nodes.

6.3 A Comparison between Adaptive GASA and other Techniques

In this section a comparison between the new technique introducing adaptive SA as a genetic operator, with the most famous techniques in the literature. The new adaptive technique is referred to as ADP-GASA.

Table 6. 1 A Comparison Between ADP-GASA and SAGA

SAGA (Brown et al., 1989)	ADP-GASA
1- The entire SA algorithm is called by the GA to improve current solution. 2- The SA algorithm is called for every newly created child, i.e. the hybridization is tightly coupled. 3- The annealing parameters are fixed for all individuals, and do not change during evolution. 4- Regular crossover and mutation operators performed by the GA.	1- The entire SA algorithm is called by the GA to improve current solution. 2- The SA algorithm is called for some newly created children, i.e. the hybridization is loosely coupled. 3- The annealing parameters are unique for each individual, and evolve with time. 4- Cluster-based crossover operator performed by the GA. Mutation is diverted to SA with some probability attached to each individual.

Table 6. 2 A Comparison Between ADP-GASA and Adaptive SAGA

Adaptive SAGA (Esbensen & Mazumder, 1994)	ADP-GASA
1- Mutation is altered by applying SA acceptance probability to newly generated offspring.	1- Mutation is altered by applying SA with a certain probability to some generated offspring.
2- Each individual has unique annealing parameters. The annealing parameters change according to the number of mutations performed on the individual.	2- Each individual has unique annealing parameters. The annealing parameters are adaptive and evolve with time.
3- The algorithm starts with a certain population size, and gradually decreases this population each time the GA goes through a stagnation period.	3- The population size is fixed and does not change over time.
4- The algorithm starts with pure GA and gradually shifts to pure SA.	4- The amount of SA performed each generation is adaptive, and changes according to the requirement of the search process.

Table 6. 3 A Comparison Between ADP-GASA and GSA

GSA (Koakutsu et al., 1996)	ADP-GASA
1- The entire SA algorithm is called by the GA to improve current solution. 2- The annealing parameters are global among all individuals in any generation. 3- Regular crossover operator performed by the GA. The mutation operator is removed and replaced with SA. The SA operator is called for every child resulting from crossover.	1- The entire SA algorithm is called by the GA to improve current solution. 2- Each individual has unique annealing parameters. The annealing parameters are adaptive and evolve with time. 3- Regular crossover operator performed by the GA. Mutation is diverted to SA with some probability attached to each individual.

Table 6. 4 A Comparison Between ADP-GASA and PGSA

PGSA (Chen et al., 1998)	ADP-GASA
1- A massively parallel hybrid SA/GA technique with one individual residing on each PE (Processing Element)	1- Purely sequential algorithm where the whole population resides on one processor.
2- The SA acceptance probability is applied to the selection operator. Selection is performed, after regular mutation and crossover, between three individuals: the resident one, the visiting one and the two newly created children. The winner becomes the resident	2- SA is a special case of mutation. Selection is regular GA selection method.
3- Each PE has a different initial temperature, final temperature and cooling factor.	3- Each individual has a different initial temperature, final temperature and cooling factor.
4- The initial temperature is calculated from random initial probability of acceptance between 10^{-10} and 1.0.	4- The initial temperature is calculated from a large change in fitness and a small initial probability like 0.01.

5- Final temperature is set with random final acceptance probability between 0 and 10^{-10}. 6- The cooling factor is calculated from the initial and final temperature of each PE.	5- No final temperature, the SA algorithm stops when stagnation is reached. 6- The cooling factor is taken from a predefined range in the initialization of the algorithm.

Table 6. 5 A Comparison Between ADP-GASA and PGSA- cont'd

Table 6. 6 A Comparison Between ADP-GASA and SAM/SAR

SAM/SAR (Adler,1993)	ADP-GASA
1- The crossover and mutation operators use the SA acceptance probability, based on a global temperature to select between parents and children.	1- Crossover and mutation work like regular GA crossover operators. Children replace the worst individuals in the population without checking any probability. Mutation is diverted to SA with a certain probability attached to the individual.
2- All individuals have the same initial temperature, and the same annealing reduction factor.	2- Each individual has its own initial temperature, its own reduction factor, and its own probability of performing SA.
3- The parameters of SA are fixed, and do not change during processing.	3- The parameters of SA are adaptive, and evolve with time.

Table 6. 7 A Comparison Between ADP-GASA and NPOSA

NPOSA (Cho et al., 1998)	ADP-GASA
1- A population oriented SA. No GA operators are used.	1- A hybrid GA/SA approach. Regular GA mutation and crossover are used. Mutation is diverted to SA with a certain probability.
2- Each individual has its own temperature.	2- Each individual has its own initial temperature, its own reduction factor, and its own probability of performing SA.
3- No reduction of temperature values. Temperature changes according to the rank of the individual in the population. The temperature is increased if the individual's rank is low. The temperature is decreased if the individual's rank is high.	3- The parameters of SA are adaptive, and evolve with time.
4- A new solution replaces the original one with a probability of acceptance that depends on the current temperature.	4- Full annealing process is performed on the selected individual, until stagnation is reached.

Chapter 7

7 Algorithm Design and Implementation

7.1 GAlib overview

The implementation of the technique described in the previous chapter uses GAlib, which is a C++ library of genetic algorithm objects produced by MIT laboratory of genetic algorithms (Wall, 1999). With GAlib we can add evolutionary algorithm optimization to almost any program using any data representation and standard or custom selection, crossover, mutation, scaling, and termination methods.

An evolutionary program developed with GAlib will work primarily with two classes:

1- The genome class: each genome instance represents a single solution to the problem under consideration.

2- The genetic algorithm class: which defines how the evolution should take place.

A problem solved using genetic algorithms with the help of GAlib must be represent a single solution in a single data structure, this is called a genome in GAlib. The genetic algorithm will create a population of solutions based on a sample genome that is provided. The genetic algorithm will then evolve the population to obtain the best solution.

7.1.1 Genome Types

GAlib provides four genome types: binary string genome, list genome, array genome and tree genome. The user can also create genomes with multiple dimensions of these types. For example, by creating a 2-d array genome, or a 3-d binary string genome.

After defining the type of the genome, the objective function of the genome should be defined. This is completely up to the user, and is not created with the help of the library. The objective function should return a positive real value that is used to evaluate the genome.

In addition to the objective function, the genome has three primary operators: initialization, mutation and crossover. These operators are defined differently for each type of genome. The user has the option to use built in operators, or to write his own operators. A genome may also have a comparator that is used to compare two genomes together.

7.1.2 Genetic Algorithms Types

The library also provides different types of genetic algorithms: the simple GA, the steady state GA, the incremental GA, and the deme GA. These types differ in the way that they create individuals and replace old individuals in the course of an evolution.

The simple GA, defined by Goldberg (1989), uses non overlapping populations and optional elitism. Each generation of the algorithm creates an entirely new population of individuals. If elitism is used, the best individual(s) is carried from one generation to the next.

The steady-state GA uses overlapping populations. In this technique the new population is added to the old one, and the worst individuals are then destroyed. The degree of overlap, which is the percentage of population that should be replaced each generation, is user defined.

In the incremental GA, only one or two children are created each generation. The user can specify what individuals should be replaced in each generation, such as replacing parents, replacing worst or random individuals in the population.

The last type, the deme GA, evolves multiple populations in parallel using a steady state GA. Each generation, the algorithm migrates some of the individuals from each population to one of the other populations.

The library also provides several termination methods. These include terminate-upon-generation, in which the user specifies a certain number of generations for which the algorithm should run, and terminate-upon-convergence in which the user specifies a value to which the best of generation score should converge. The termination function can also be customized depending on the problem type.

The library has several selection strategies that are used to select parents for mating. For example, roulette wheel selection picks an individual depending on the magnitude of the fitness score relative to the rest of the population, while uniform selection picks individuals randomly from the population.

GAlib is thus very easy to use and connect to existing optimization problems. The user has the ability to customize any component in the library to suit the current problem. In addition, the library provides useful statistical information about the population that will help the user to easily analyze the performance of the algorithm.

7.2 Implementation Details of the Adaptive GASA algorithm

The adaptive hybrid GA and SA technique was tested on the MAP problem on BBNs described in chapter 4. The objective is to find a network assignment that maximizes the overall joint probability.

Without loss of generality, the following simplifying assumptions were made:

1- All nodes in the network are binary valued, i.e. each node can assume either True or False only.

2- The evidence set is empty, i.e. no nodes are pre-instantiated.

3- The maximum number of parents for each node in the network is 15 parents.

7.2.1 Genome Representation

The genome in our problem represents a candidate solution to the MAP problem. In our representation, the genome consists of two parts:

1- **BBN part**: which is a binary string consisting of truth assignments to all nodes in the network. Each gene in this part can have a value of either 0 or 1.

2- **SA part**: consisting of the parameters of SA that will be optimized during the search. The selected parameters are the annealing reduction factor α, and the probability of performing SA on that individual P_{SA}. Each of these two genes is a real valued gene taken from a specific range determined by the user. For example, The gene corresponding to the parameter α can take values in the range [0.990,0.999], while the gene corresponding to the parameter P_{SA} can take values in the range [0.0,1.0], or any other suitable probability range.

The following is an example of a genome in our implementation

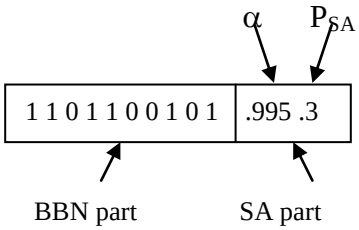


Figure 7. 1 The Genome Representation

7.2.2 Objective Function

The objective function is the joint probability corresponding to the truth assignment of the nodes as they appear in the current genome. To calculate the objective function, the conditional probability for each node in the network is calculated by finding the combined "truth values" of the parents of this node. This combined "truth values" is transformed to a binary value that corresponds to the entry in the conditional probability table (CPT) of the required node. The joint probability is then found by accumulating the product of all conditional probability values of all nodes.

For example, assume we want to calculate the conditional probability of node *x*, given that *x* has three parents whose truth assignments are *T F F*. The binary value "100" is calculated from the truth assignments of the parents, and entry number *4* of the CPT of node *x* is retrieved. If *x* has a truth assignment of *F* rather than *T*, the probability value is negated before being multiplied with other conditional probabilities.

Observe that the extra genes, which correspond to SA parameters, do not have a direct role in the calculation of genome objective. Instead, they play an indirect role, because their presence affects the offspring created by an annealing operator that uses these parameters.

7.2.3 Initialization

GAlib initializes a population of genomes (solutions), by creating several genomes of the predefined structure. The initialization uses random gene values with a uniform distribution. The user can specify a seed for randomization. Doing so, the processing will be exactly the same every time the program is run.

7.2.4 The Genetic Algorithm

The genetic algorithm selected was a steady state GA (described in section 7.1.2), with overlapping populations. Each generation, 20% of individuals in the population, which are the worst among the population, are replaced by new offspring.

Selecting parents for mating is based on Roulette Wheel selection, in which selection is based on the fitness of the individual, i.e. higher fitness individuals have a higher chance of being selected.

The termination criterion was to terminate upon convergence, i.e. when the maximum fitness has not changed for a specified number of generations.

7.2.5 Outline of the Hybrid GA-SA Algorithm

1- Read the BBN network, from an input file.

2- Create all data structures needed for processing.

3- Define a genome representation.

4- Define the objective function.

5- Define GA parameters and functions: probability of mutation (P_{mut}), probability of crossover (P_{cross}), population size (n), termination criterion, initialization method, mutation method, and crossover method.

6- Initialize a population of individuals by cloning the required number of genomes.

7- Initialize the number of generations.

8- Do *n/2* times

{

- Select two individuals *parent1* & *parent2* for mating
- With probability P_{cross}, perform crossover between *parent1* and *parent2* to produce two children.
  - *child1 = Crossover(parent1,parent2)*
  - *child2 = Crossover(parent1,parent2)*
- If crossover is not performed
  - *child1=parent1*
  - *child2=parent2*
- Perform mutation on *child1* with probability P_{mut}.
- Perform mutation on *child2* with probability P_{mut}.
- Insert resulting children in the population according to the predefined replacement strategy

}

9- If termination is reached, terminate the algorithm.

10- Otherwise, increment number of generations and return to step 8.

7.2.6 Crossover Method:

The crossover method used is the cluster-based crossover, described in (Rojas-Guzman & Kramer, 1993). In this technique, a random node is selected as the center of a cluster, and then all nodes that fall within a certain number of links away from the center

node are exchanged between the two children. The required number of links is a user-defined parameter called *ClusterLimit.*

A 2-d array, called *Cost* array, is created from the network under consideration. This array contains the shortest path between each pair of nodes in the network, in terms of number of links between them. The *Cost* array is created only once at the beginning of the run.

The following steps are performed during crossover for the BBN part.

1- *child1= parent1* // child1 becomes a copy of parent1

child2= parent2 // child2 becomes a copy of parent2

2- Select a random node as the center of the cluster, call this node the *Root*.

3- Do the following for every node *i* in the network.

If (*Cost [Root][i]* $\leq$ *Cluster_Limit*) // if node i falls within the cluster

child1 inherits gene *i* from *parent2*

child2 inherits gene i from *parent1*

For the SA part, which consists of two genes corresponding to α and P_{SA}, regular one point crossover is performed with probability $P_{SAcross}$

7.2.7 Mutation Method

The mutation function is the part in which SA comes to play. The following steps are performed during mutation:

1. Check the value of the gene corresponding to P_{SA} in the individual to be mutated.
2. Perform SA on the current individual (solution) with probability P_{SA} specific to this individual.
3. If SA is not performed call regular mutation operator.

7.2.8 Regular Mutation Operator

The regular mutation operator is called in two cases: first, to create a random move during annealing, and second, when SA is not performed and regular GA mutation operator is to be performed on the current individual.

The regular mutation operator is the flip bit mutation defined in (Goldberg, 1989). Each bit in the BBN part of the genome is flipped with probability P_{mut}. The SA part, on the other hand, is mutated, with probability P_{SAmut}, by adding or subtracting a random fraction of the range from which the gene is defined to the original gene value. The maximum allowed change is 20% of the current range, i.e. the new gene value is calculated as follows:

New gene value = old gene value + { r × (MaxValue-MinValue) × 0.2}

Where r is a random real number between 0 and 1, MaxValue is the maximum allowed gene value, MinValue is the minimum allowed gene value.

7.2.9 Simulated Annealing Operator

The simulated annealing operator performs a full annealing schedule on the current solution. The neighborhood operator used to generate a random neighbor of the current solution is identical to the regular mutation operator of GA, except that only the BBN part of the genome is mutated, the SA part is left unchanged, when a random move is created. The idea is that while SA is performed, we do not want to change the annealing schedule used to initialize SA at the beginning, i.e. it makes no sense to change the annealing parameters (P_{SA} and α) that we started with. In addition, we want all random solutions created during SA to have the same annealing parameters as their seed (starting solution). When the final solution replaces the initial solution, the SA part will

be the same as the initial solution, while the BBN part will be different. The fitness of the resulting solution reflects whether the annealing parameters used were good or bad.

As mentioned above, the probability of performing SA, and the reduction factor α are taken from the current solution. It remains to determine the initial temperature value. One of the options that were considered when the algorithm was designed was to include the initial temperature among the SA parameters that will evolve. It was found, however, that it is very difficult to define a suitable temperature range that will work for all solutions and for all phases of the run. The initial temperature is very problem specific and very greatly affects the final result. The alternative was to calculate the initial temperature adaptively during the run. When SA is performed, a large change in fitness (Δ) is calculated as:

Δ = *Objective (current solution) – smallest possible fitness.*

The smallest possible fitness was set to the extreme case of zero.

Thus Δ = *Objective (current solution)* *(1)*

From the equation of calculating the probability of acceptance

$P_{accept} = exp(-\Delta/T)$ *(2)*

We can calculate the initial temperature T_0 as

$T0 = -\Delta / ln(P_{accept})$ *(3)*

The initial probability of acceptance P_{accept} was then set using trial and error to some small value like 0.05 or 0.1.

Calculating the initial temperature using the above method gives initial temperature values that are smaller than usual. This in fact what we need in the current algorithm. The initial solution with which SA starts is usually a good solution, because it

has gone through several modifications by GA operators. Starting the annealing process with high temperature values may destroy the good starting solution, because any bad move may be accepted. On the other hand, when the initial temperature is relatively small, the good starting solution is protected, since the probability of accepting a very bad move becomes lower.

The SA operator performs the following steps:

1- Initialize the initial temperature value T_0 using the above method.

2- Set current temperature $T = T_0$

3- Determine the annealing reduction factor α from the corresponding gene in the current solution.

4- While stagnation is not reached do the following.

- Create a new solution using regular GA mutation.
- If the new solution improves fitness replace current solution
- Otherwise, replace current solution with probability $exp\ (-\Delta/T)$
- Decrement current temperature by setting $T = \alpha T$

Observe that the algorithm performs only one iteration per temperature. This choice was made because the cooling factor α is chosen sufficiently high to guarantee very slow cooling, which is equivalent to performing several iterations per temperature with faster cooling. The termination criterion of the algorithm is reaching stagnation when the fitness of the current solution does not change for a specific number of iterations.

Chapter 8

8 Results and Discussion

8.1 How the Algorithm was Tested

The technique was tested by comparing three versions of the algorithm:

1- GA-alone: A regular genetic algorithm without simulated annealing.

2- Non-adaptive GASA: A fixed genetic-annealing algorithm without evolving parameters.

3- Adaptive GASA: a hybrid GA-SA algorithm with evolving SA parameters.

Problems Faced During Testing

One of the problems faced during implementation and testing of the program was the **premature convergence problem**, which is one of the classical problems in GAs. It was found that the algorithm converges very rapidly to a sub-optimal solution. This problem had a more profound effect on the adaptive version than the other two versions of the program. This is due to the fact the adaptation and the learning process requires an adequate number of generations to demonstrate its effect. Premature convergence would stop the learning process before the parameters reach their optimal values.

The cause of the problem was the large number of duplicates that result during processing. A sub-optimal solution could easily dominate the whole population, and diversity is lost as a result. To overcome the problem and increase diversity in the population we did the following: an individual resulting from crossover was first checked for having a duplicate in the population. If this was the case, that individual is forced to undergo mutation before being inserted in the population. The process is repeated as long as the individual still have copies in the population.

The second problem was the task of **parameter adjustment**. The algorithm contains so many parameters, some of them are related to the GA like probability of mutation, probability of crossover, population size...etc. Other parameters are related to SA like initial temperature, reduction factor α, probability of performing SA, and the number of iterations required for declaring stagnation. Other parameters are related to the BBN problem like the cluster limit parameter. Finally, other parameters are related to adaptation like probabilities of performing crossover and mutation for SA parameters (genes), and the rate with which these parameters should change during mutation.

During experimentation, it was possible to identify the critical parameters that affect performance. These parameters are: population size, the number of generations required for declaring convergence, probability of GA mutation, initial SA temperature, SA reduction factor α, and probability of performing SA. These parameters were adjusted for each data file separately. All other parameters were fixed for all data files after approximately finding the best possible parameters, since they do not have much effect on performance.

8.2 Data Files

Each of the three versions was tested on three different data sets. Each data set represented a different BBN characteristic.

Set A: consists of three BBNs, each with a large number of cycles. Three data files were created manually and were called:

"50a": which consists of a 50-node subset chosen from the 80-node network shown in Appendix B.

“60a”: which consists of a 60-node subset from the original 80-node network shown in Appendix B.

“70a”: which consists of a 70-node subset from the original 80-node network shown in Appendix B.

Set B: consists of three BBNs, each is a layered network and each node in the network has 3 to 5 parents. These three networks correspond to network structures with a large number of cycles as well, but in a layered structure. The first data file, which contains 70 nodes, is called “70b” and is shown in Appendix B, the second data file is a subset form the first file with 60 nodes, and is called “60b”. The third data file is a subset from the second file with 50 nodes, and is called “50b”. The files in this set were also created manually.

Set C: consists of four BBNs, each is characterized by having nodes with a very large number of parents. The maximum number of parents was 15 nodes. These files correspond to network structures with heavy connectivity. The files were created using a program that creates random BBN files. The files were a 60-node network called “60c”, a 70-node network called “70c”, a 90-node network called ”90c”, and a 100-node network called “100c”.

8.3 The Testing Process

The following steps were carried out to test the algorithm on each of the above mentioned data files.

1. The same random seed was selected for all three versions. This will make the processing identical each time the program is run.

2. The following GA parameters were fixed for all data files

Crossover rate = 0.99

Percentage of population overlap = 0.2

The number of links defining a cluster = 1

3. Other parameters of the GA-alone version were adjusted to give the best possible result. These parameters are **population size**, **mutation rate** and the **number of generations required for declaring convergence**.

4. The best set of GA parameters was used in both the GASA adaptive and non-adaptive versions.

5. For the non-adaptive version, the following parameters were adjusted to give the best possible result:

Initial SA temperature: which was calculated heuristically from a large Δ and a small initial probability of acceptance.

SA reduction factor α : which was set to some value between 0.992 and 0.996

SA rate (Probability of performing SA): which was set to some value between 0.002 and 0.2 depending on the population size. A large population size required a smaller SA rate to avoid very large processing time.

6. for the adaptive version the following parameters were adjusted for best possible result:

The reduction factor range: which was set to [0.990,0.999] for all data files.

The SA probability range: which was set differently for each date file depending on the population size, as in the non-adaptive version, to avoid large processing time. The largest range was [0.005, 0.05]

The task of adjusting parameters for the adaptive version was much easier than for the non-adaptive version. The non-adaptive version required finding one optimum value for each parameter, while the adaptive version only required suggesting a suitable range of parameter values. In addition, the adaptive version did ot require adjusting the initial temperature value, because this value was calculated heuristically from the fitness value of the current solution as explained in the implementation chapter (chapter 7).

8.4 Crossover Effect

The effect of crossover was tested by comparing three types of crossover

1- Regular one-point crossover
2- Cluster-based crossover, explained in section 7.2.6
3- A modified version of cluster-based crossover, in which a child that has a duplicate in the population is forced to undergo mutation before being inserted in the population. We call this type of crossover "modified cluster-based crossover".

The GA alone version was tested, using each type of crossover, on 6 data files: 50a, 50b, 60b, 60c, 70a, and 70b.

8.5 Summary of results

The results obtained my be summarized as follows

8.5.1 Results for Set A

Maximum Fitness

- All three versions obtained the same result in 2/3 cases (50a,70a)
- The GA alone version and the adaptive version reached the same result while the non- adaptive GASA obtained a slightly less value in 1/3 cases (60a).

Average Fitness

- GA alone obtained the best final average in 3/3 cases (50a, 60a, 70a), although the average fitness grew faster in the adaptive version than in the other two versions. However, since the adaptive version converged in a less number of generations, it did not reach the same final average value as GA alone. See for example average fitness chart 60a (figure 8.5)
- The non-adaptive version performed better than GA alone, in terms of the rate of average fitness improvement but not the final average value, in 1/3 cases (70a).

Evolving Parameters

- The average SA factor tends towards decreasing in all 3/3 cases (50, 60a, 70a), with more fluctuations in 1/3 cases (60a).
- The average SA probability shows a slight tendency towards increasing at the beginning of the run, and then stabilizes throughout the run in 3/3 cases (50a, 60a, 70a), with more fluctuations noticed in 1/3 cases (70a).

8.5.2 Results for Set B

Maximum Fitness

- All three versions obtained the same result in 2/3 cases (50b,60b)
- The adaptive version obtained better result than the other two versions in 1/3 cases (70b).
- The non-adaptive version obtained better result than GA alone in 1/3 cases(70b)

Average Fitness

- GA alone obtained the best final average in 1/3 cases (50b)
- Non-adaptive GASA obtained the best final average in 1/3 cases (60b)
- Adaptive GASA obtained the best final average in 1/3 cases (70b)
- The average fitness grew much faster in the adaptive version than in the other two versions (See for example average fitness chart 70b figure 8.21). However, since the adaptive version converged in a less number of generations, it did not reach the same final average value as GA alone.
- The average fitness grew faster in the non-adaptive version than GA alone in 2/3 cases (50b, 60b).

Evolving Parameters

- No specific pattern can be observed for the SA factor, In case 50b, it started by increasing and then started to decrease. In case 60b, the tendency was towards increasing. In case 70b, the tendency was towards slight decreasing.
- The average SA probability showed a slight tendency towards increasing at the beginning of the run and then stabilizes throughout the run in all three cases (50b, 60b, 70b).

8.5.3 Results for Set C

Maximum Fitness

- The adaptive version obtained the same result as GA alone, but better than non-adaptive GASA, in 1/4 cases (60c).
- The adaptive version obtained better result than both GA and non-adaptive GASA in ¾ cases (70c, 90c, 100c).
- The non-adaptive version obtained better result than GA alone in ¾ cases (70c, 90c, 100c).
- GA alone obtained better result than non-adaptive version in ¼ cases (60c).

Average Fitness

- The adaptive GASA obtained much better average fitness, in terms of both the final value and the rate of improvement, in 4/4 cases (60c, 70c, 90c, 100c).
- The non-adaptive version obtained better results than GA alone, in terms of final average fitness, in ¾ cases (70c, 90c, 100c). The rate of average fitness improvement was similar for both versions in these three cases.
- GA alone obtained better result than non-adaptive GASA, in terms of both final average fitness value and rate of average fitness improvement, in ¼ cases (60c).

Evolving Parameters

- In ¾ cases (60c, 70c, 100c), the average SA factor started by fluctuation between increasing and decreasing at the beginning of the run, and then tended towards increasing at the end of the run.
- In ¼ cases (90c), the average SA factor started by increasing and then started to decrease until the end of the run.
- The average SA probability showed a slight tendency towards increasing at the beginning of the run and then stabilized throughout the run in all four cases (60c, 70c, 90c, 100c).

8.5.4 Effect of Crossover

- The cluster modified crossover operator performed better than the other two types of crossover in 4/6 cases (60b, 60c, 70a, 70b). It gave the same result as the cluster-based crossover operator in 1/6 cases (50a), and the same result as the one-point crossover operator in 1/6 cases (50b).
- The cluster-based crossover operator performed better than one-point crossover in 3/6 cases (50a, 60c, 70a).
- One-point crossover performed better than cluster-based crossover in 3/6 cases (50b, 60b, 70b).

8.6 Experimental Results

SET A

Data File 50_a

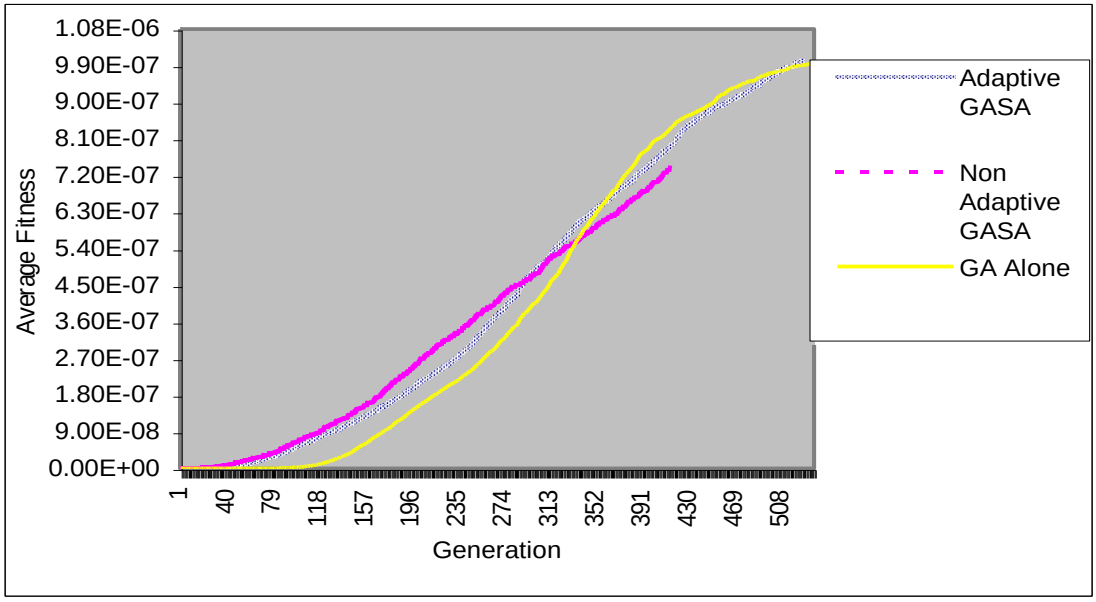


Figure 8. 1 Average Fitness Chart 50a

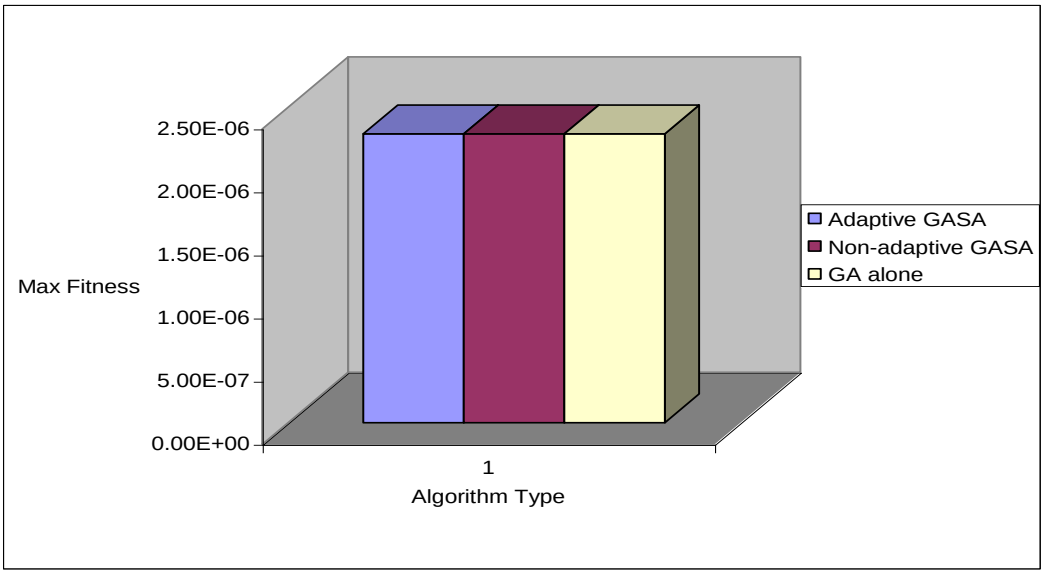


Figure 8. 2 Maximum Fitness Chart 50a

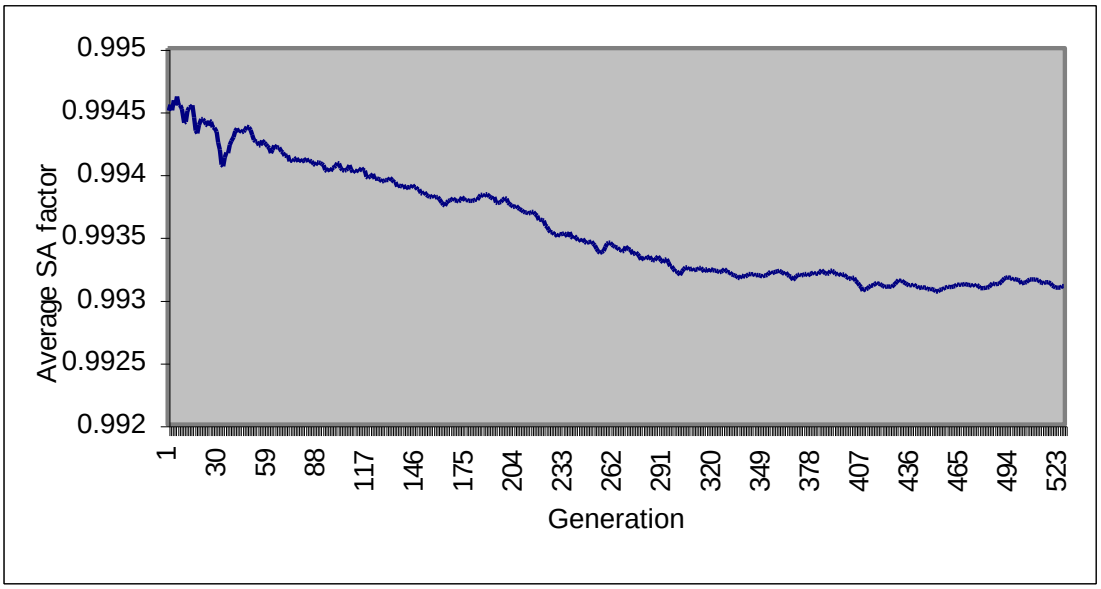


Figure 8. 3 Evolving Average SA factor 50a

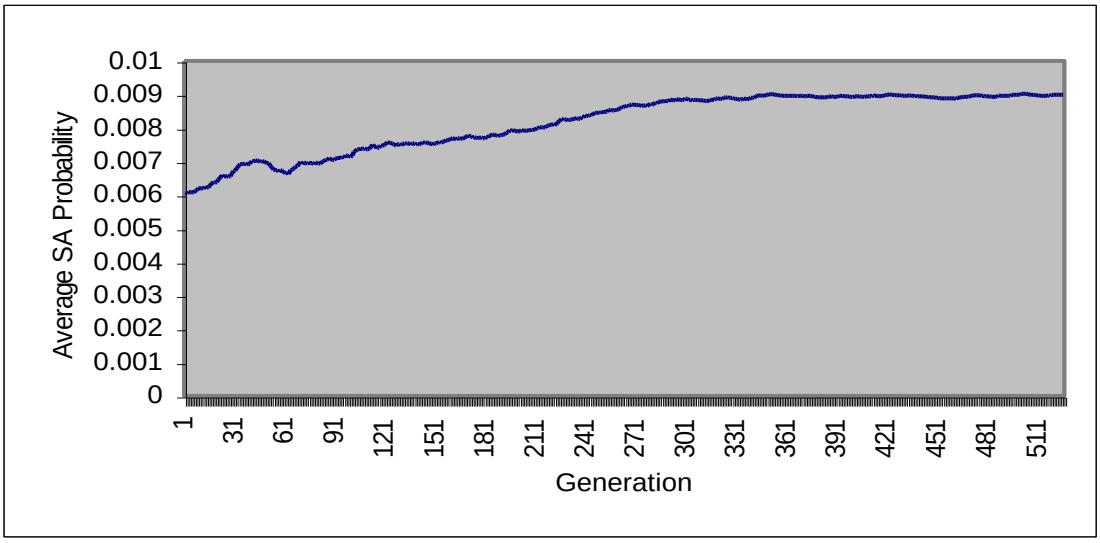
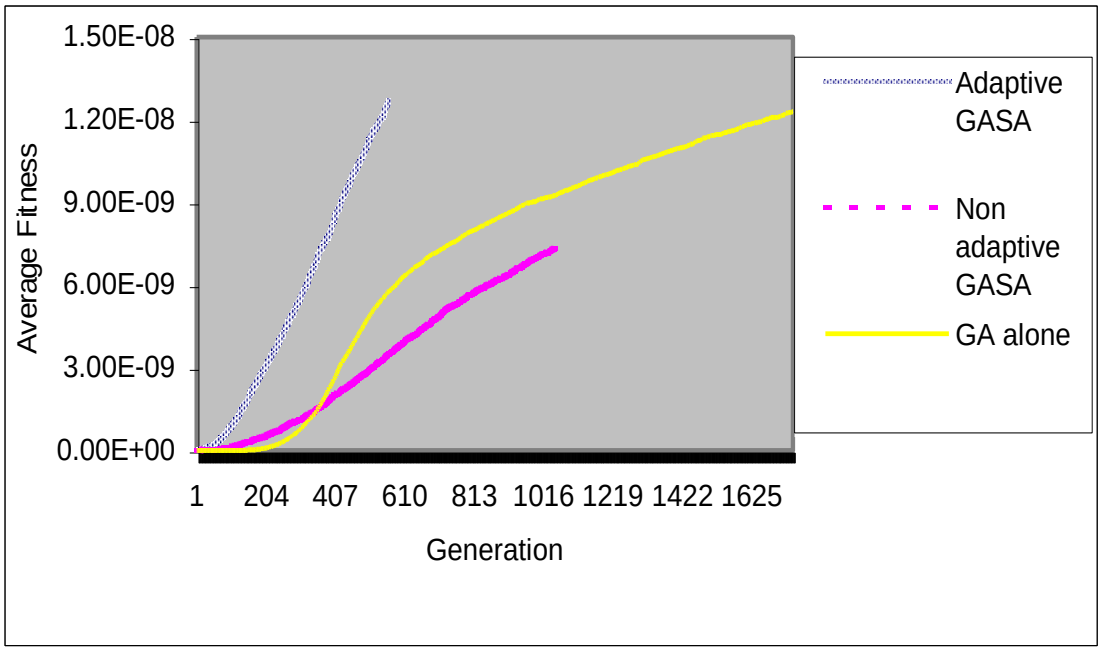


Figure 8. 4 Evolving Average SA Probability 50a

Data File 60_a

Figure 8. 5 Average Fitness Chart 60a



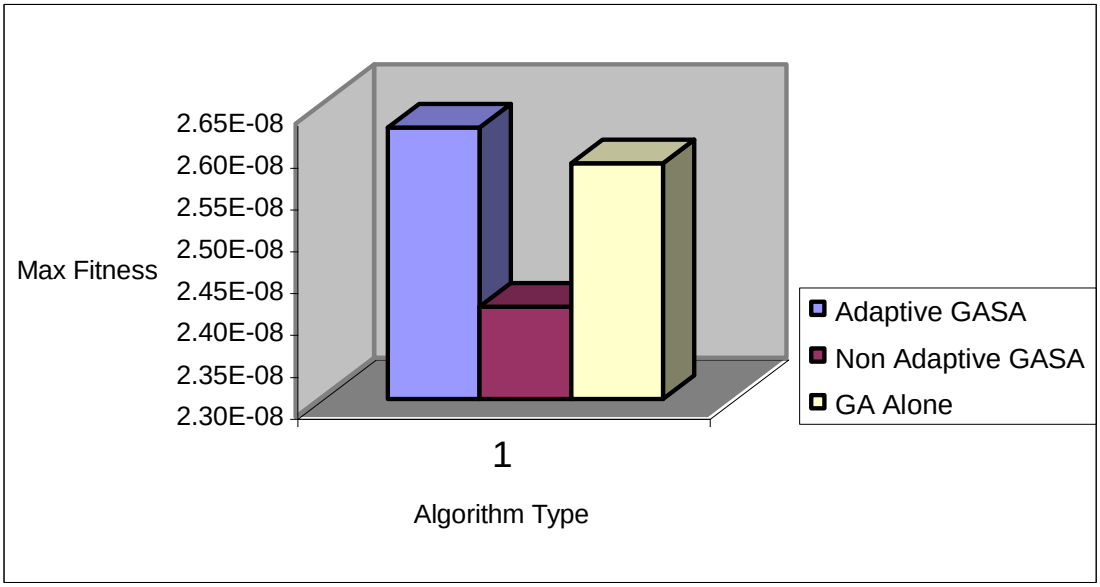


Figure 8. 6 Maximum Fitness Chart 60a

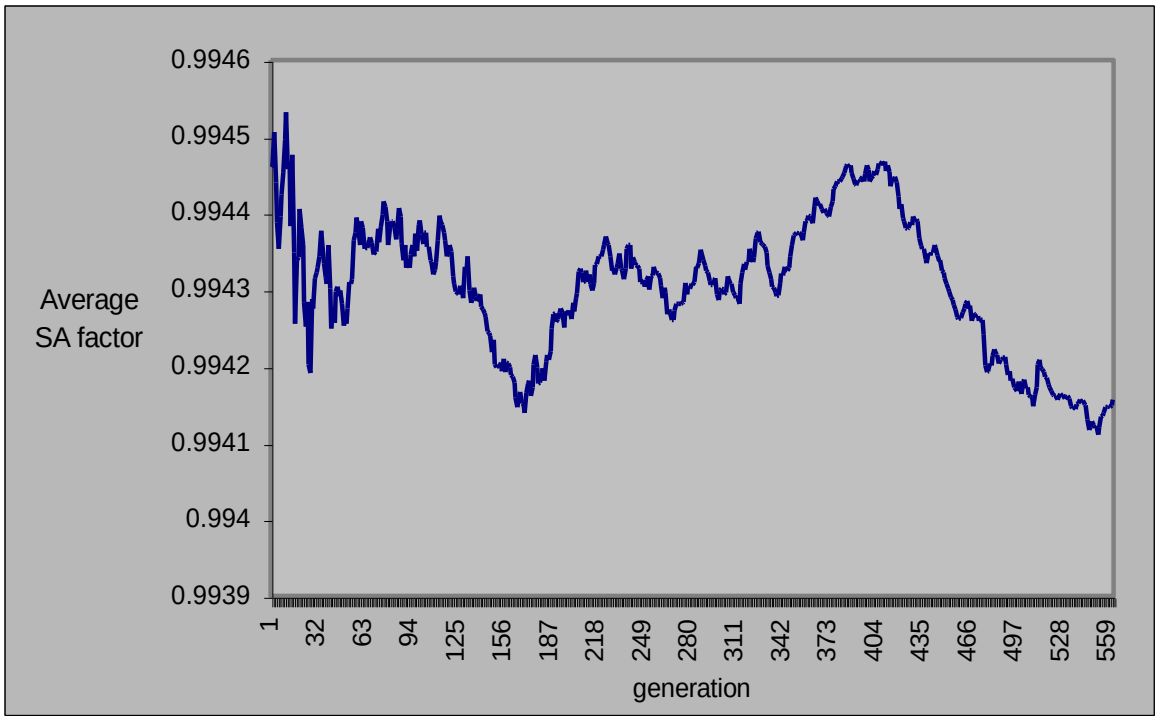


Figure 8. 7 Evolving Average SA Factor 60a

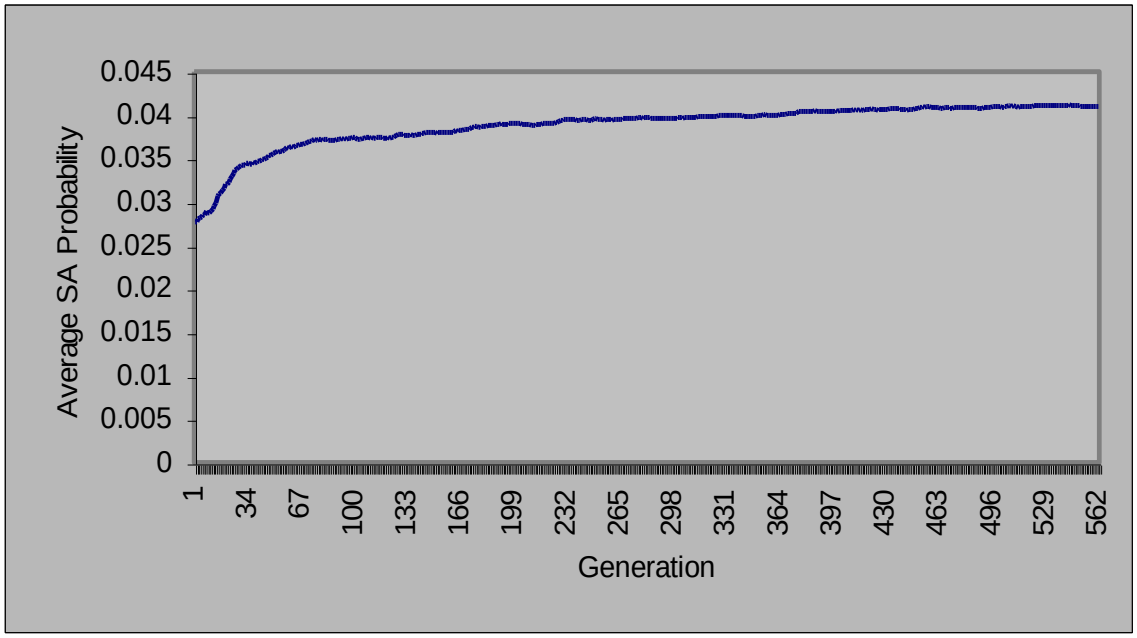


Figure 8. 8 Evolving Average SA Probability 60a

Data File 70_a

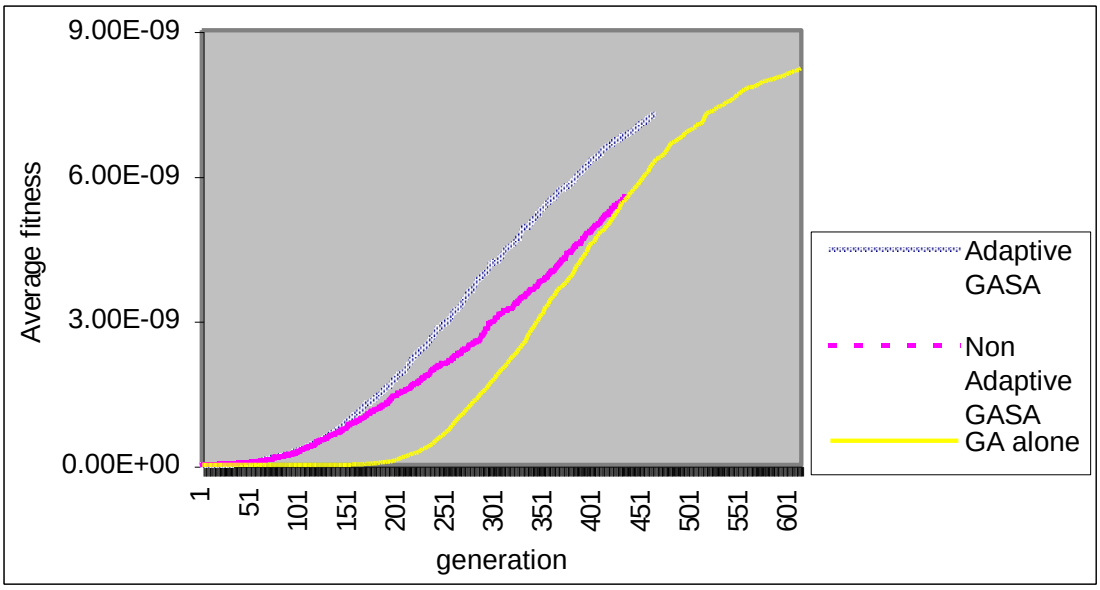


Figure 8. 9 Average Fitness Chart 70a

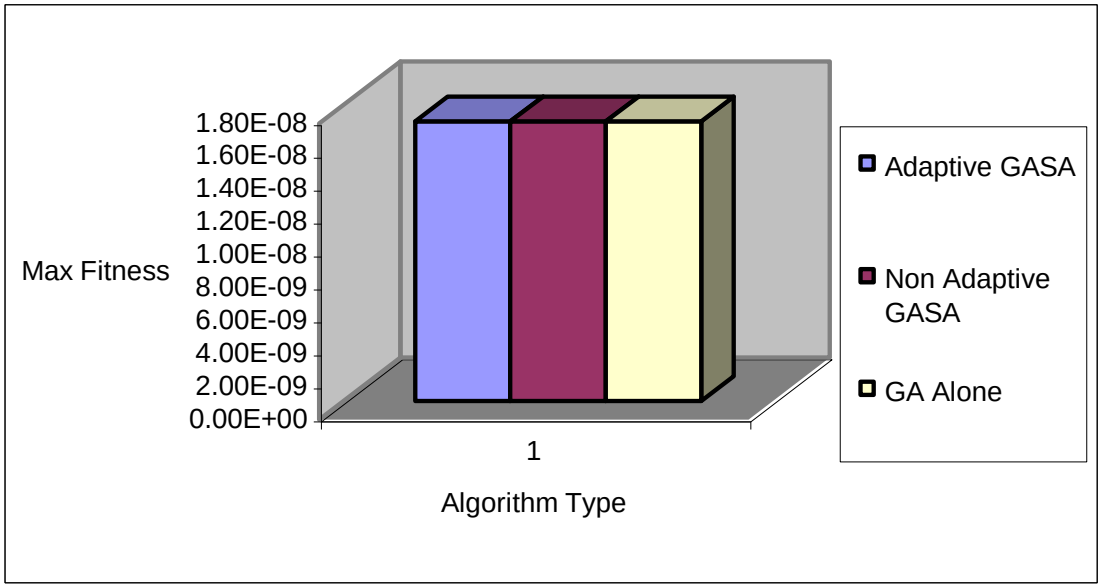


Figure 8. 10 Maximum Fitness Chart 70a

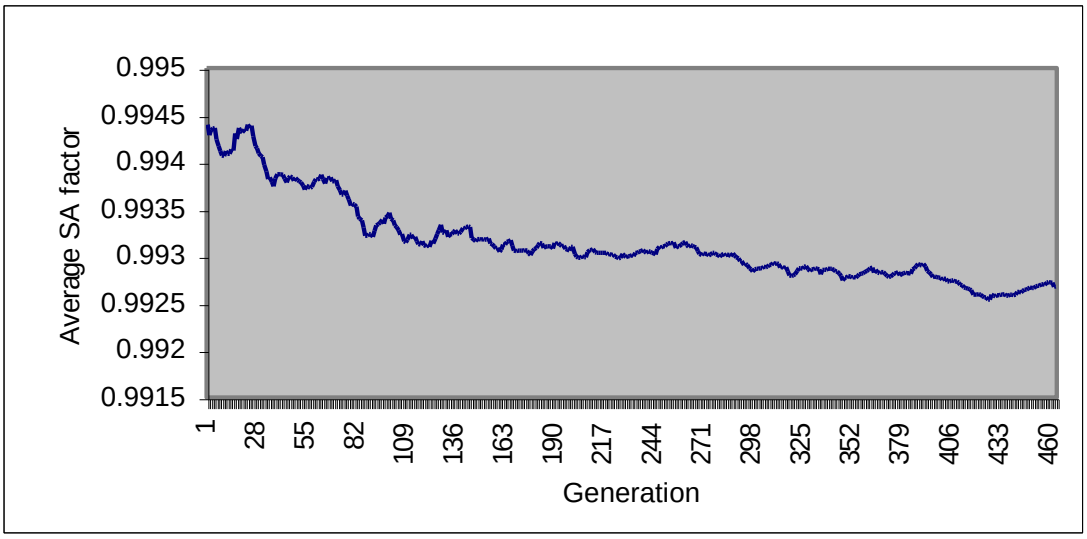


Figure 8. 11 Evolving Average SA Factor 70a

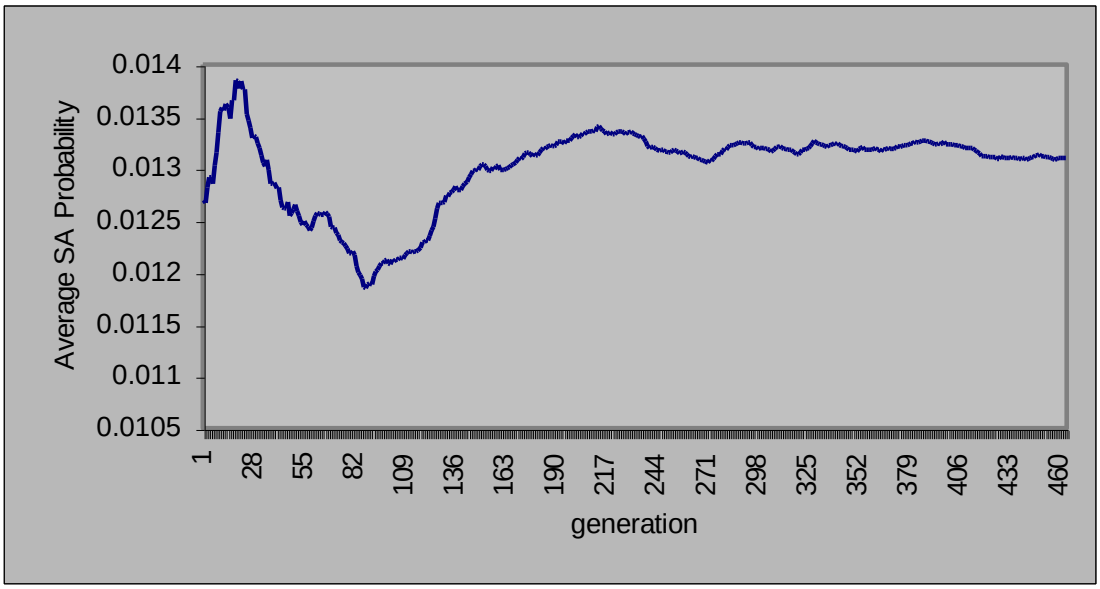


Figure 8. 12 Evolving Average SA Probability 70a

SET B

Data File 50_b

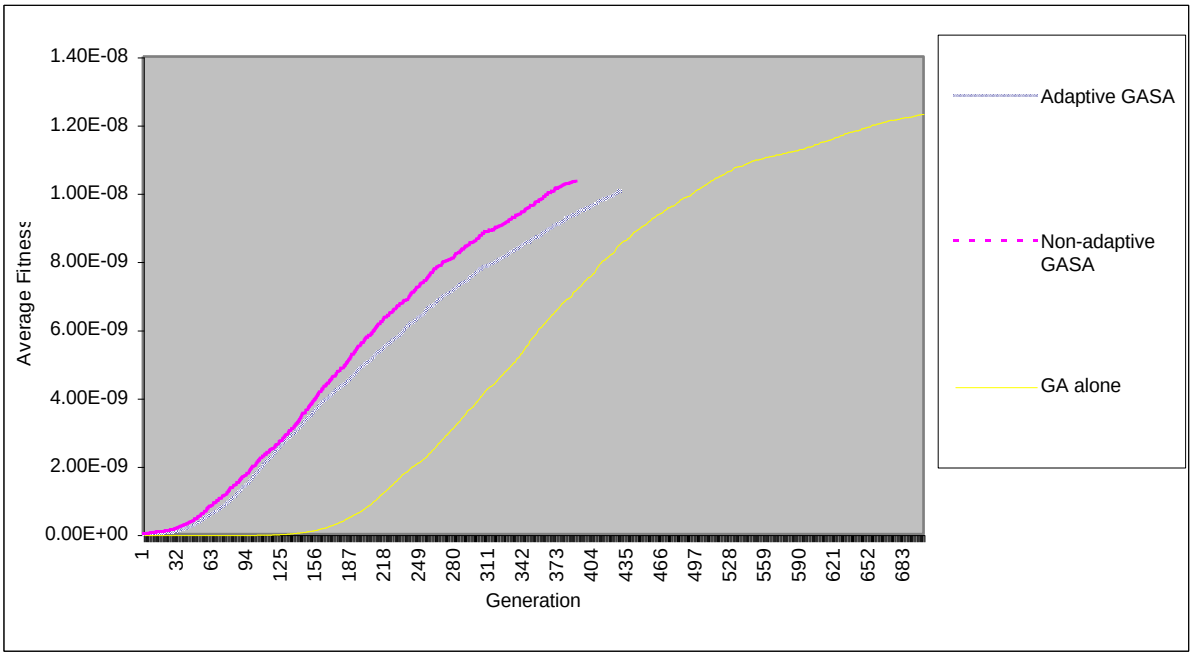


Figure 8. 13 Average Fitness Chart 50b

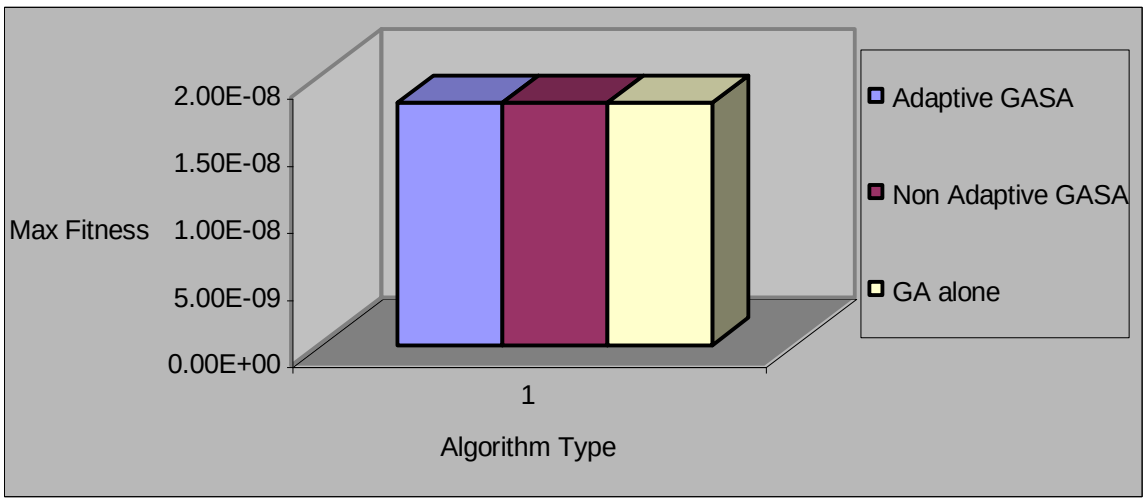


Figure 8. 14 Maximum Fitness Chart 50b

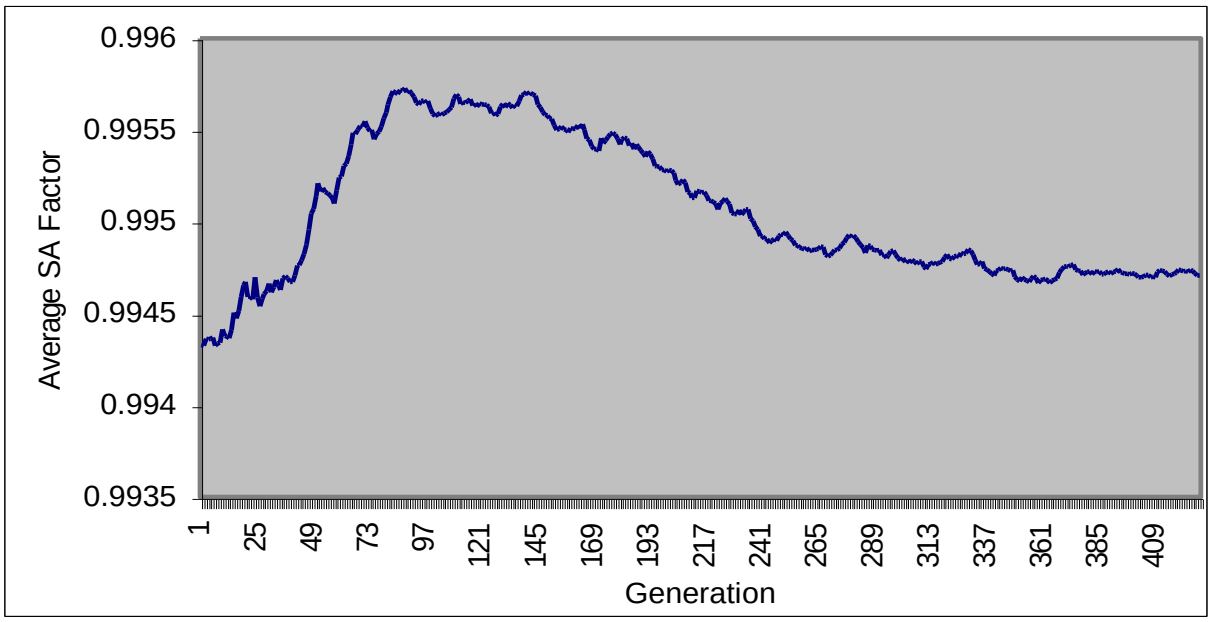


Figure 8. 15 Evolving Average SA Factor 50b

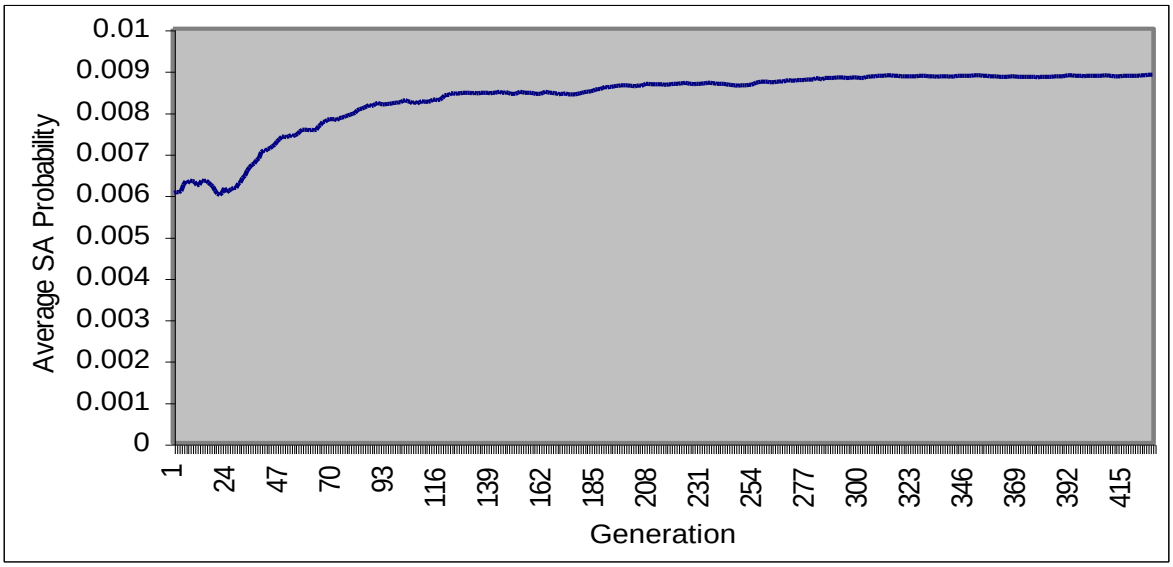


Figure 8. 16 Evolving Average SA Probability 50b

Data File 60_b

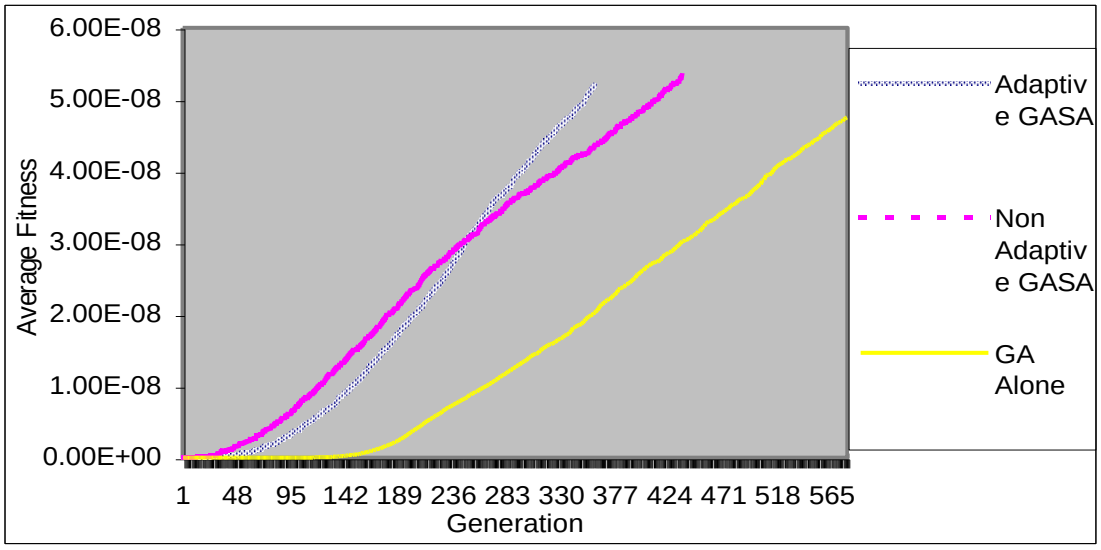


Figure 8. 17 Average Fitness Chart 60b

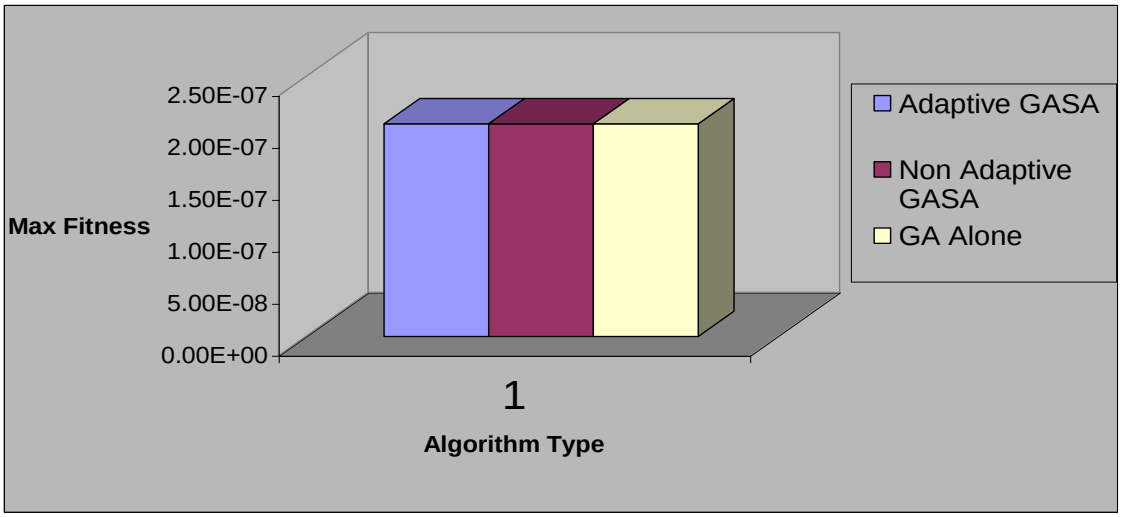


Figure 8. 18 Maximum Fitness Chart 60b

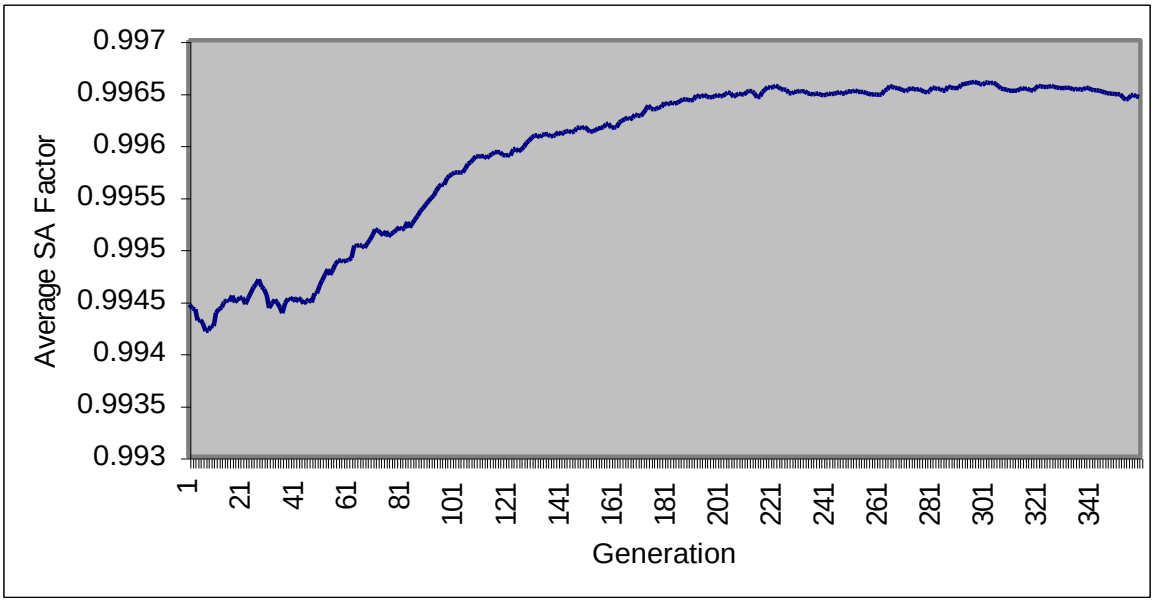


Figure 8. 19 Evolving Average SA Factor 60b

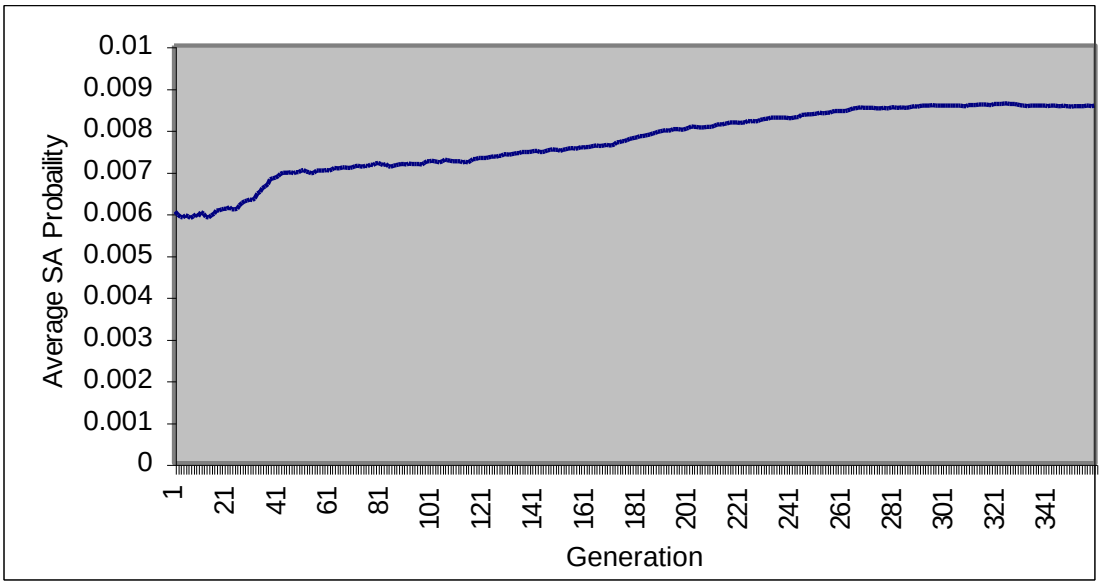


Figure 8. 20 Evolving Average SA Probability 60b

Data File 70_b

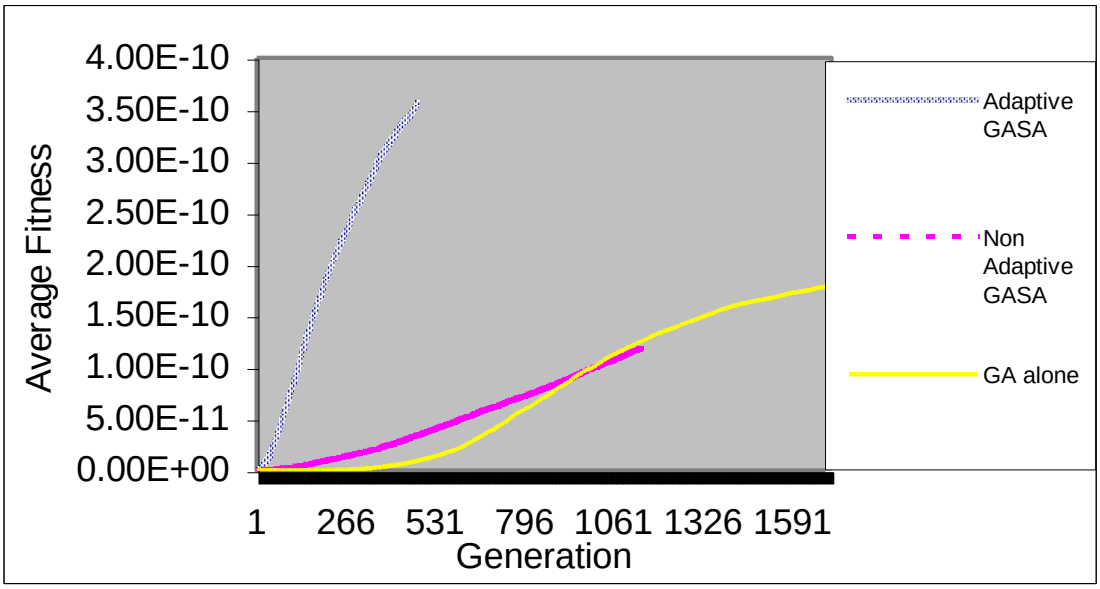


Figure 8. 21 Average Fitness Chart 70b

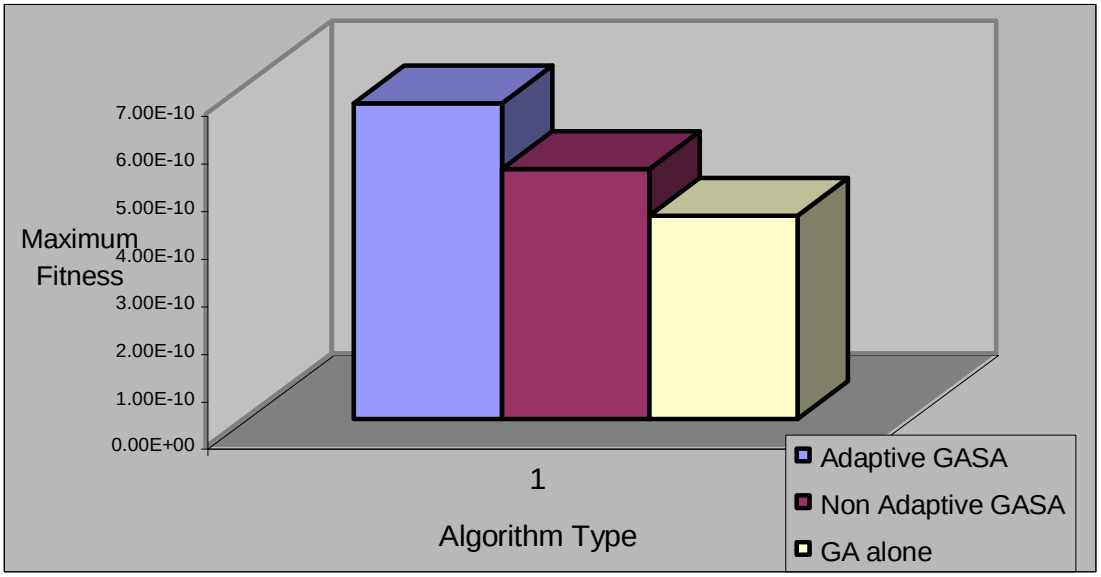


Figure 8. 22 Maximum Fitness Chart 70b

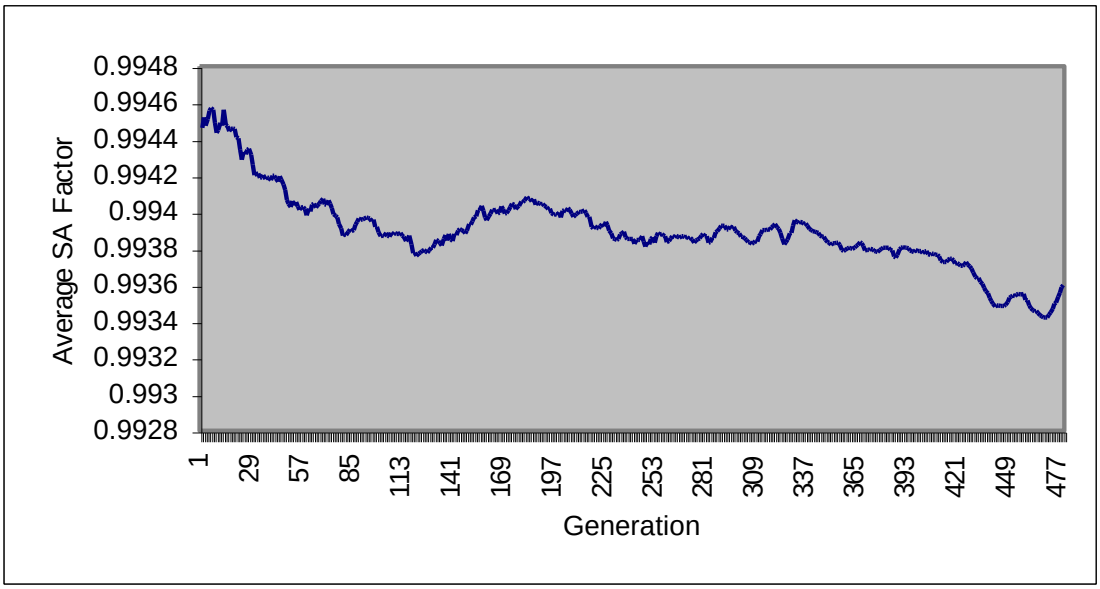


Figure 8. 23 Evolving Average SA Factor 70b

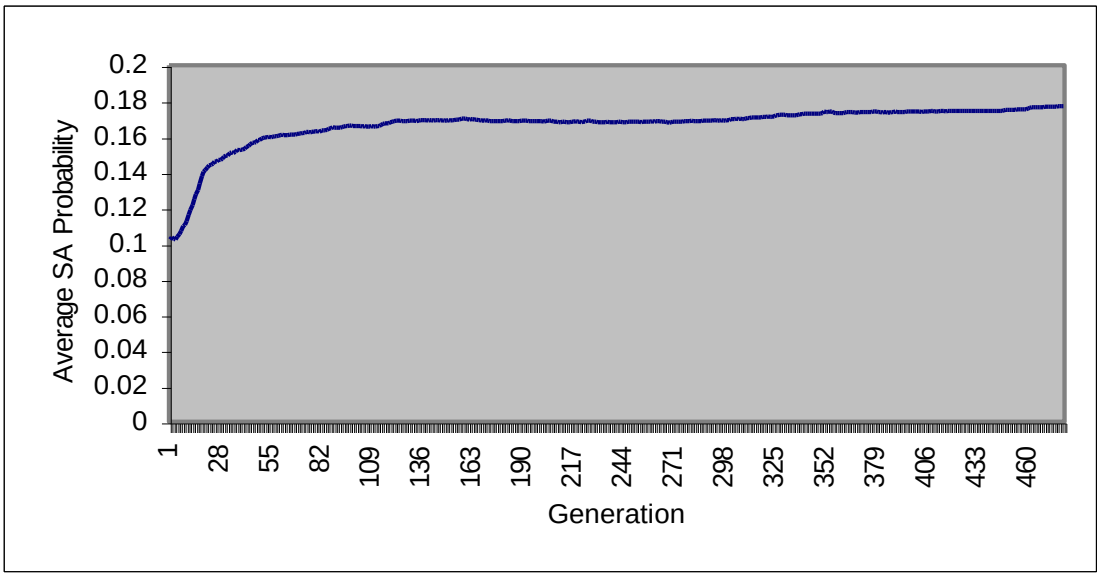


Figure 8. 24 Evolving Average SA probability 70b

SET C

Data File 60_c

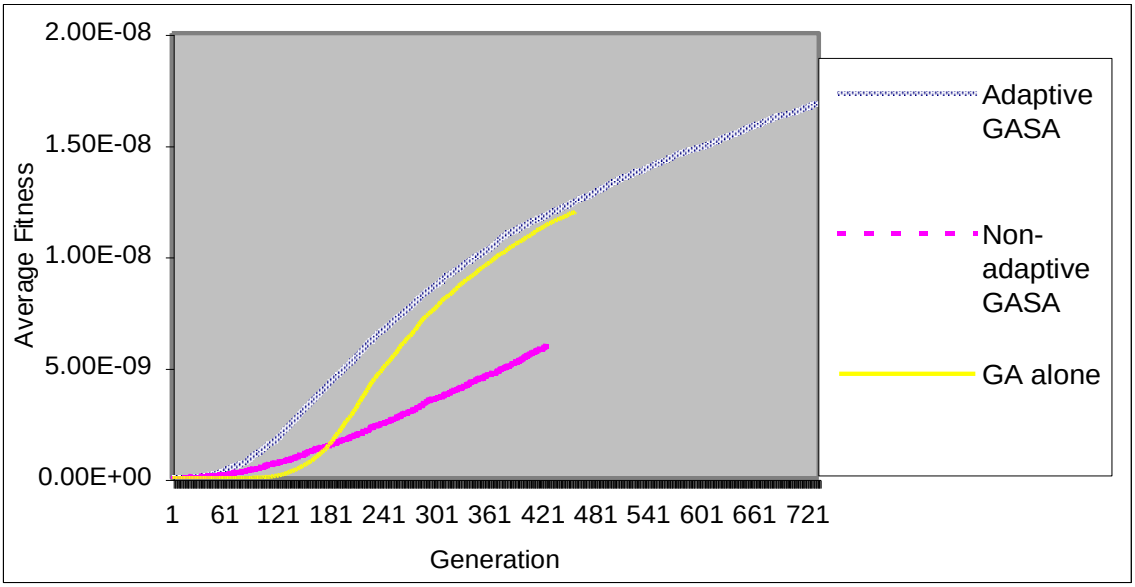


Figure 8. 25 Average Fitness Chart 60c

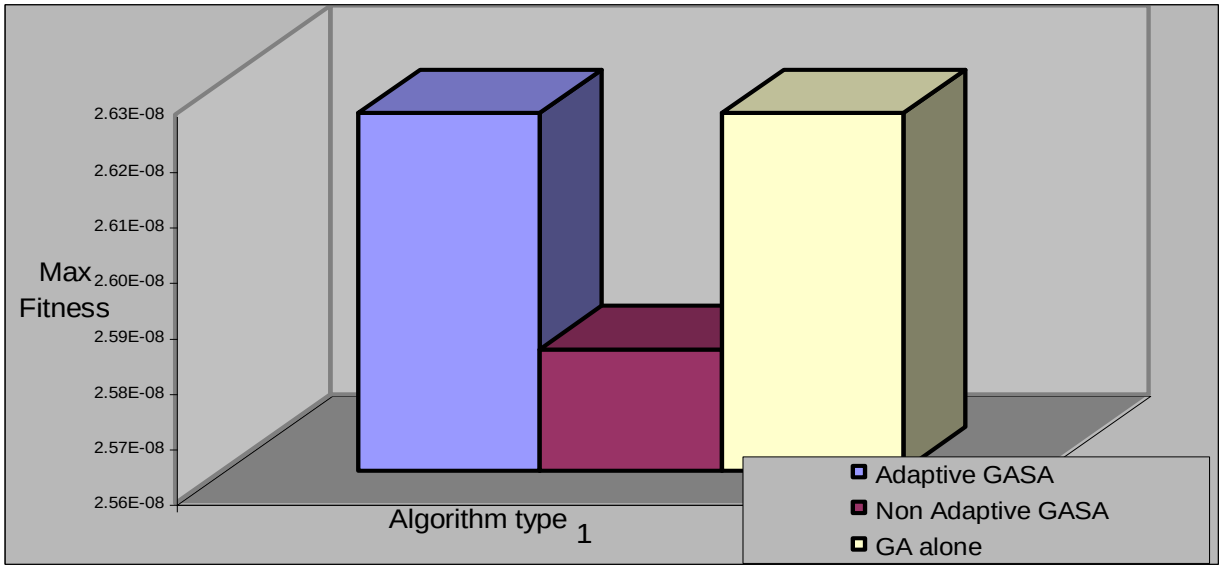


Figure 8. 26 Maximum Fitness Chart 60c

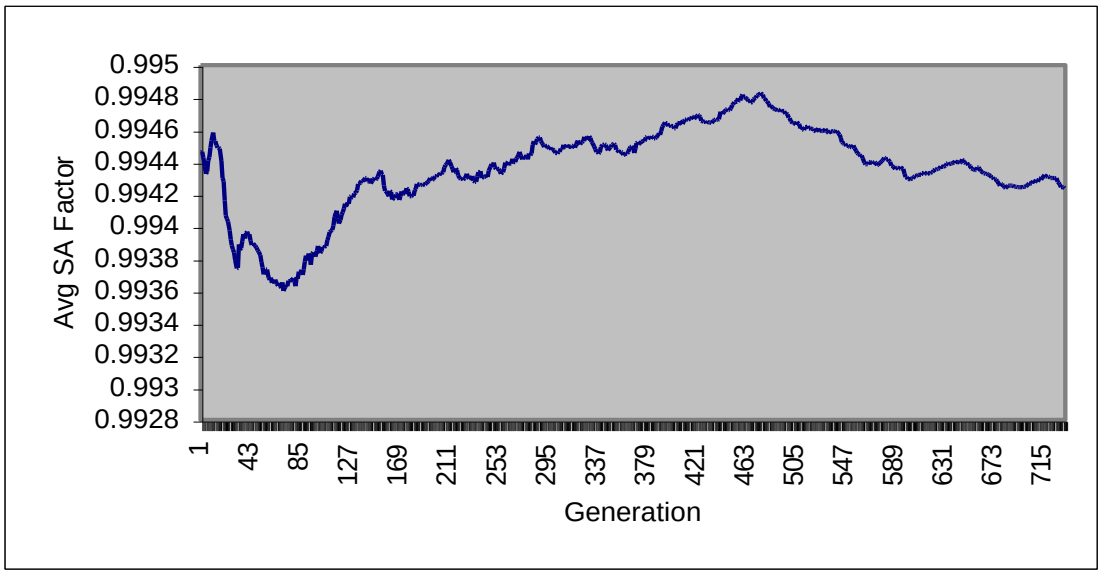


Figure 8. 27 Evolving Average SA Factor 60c

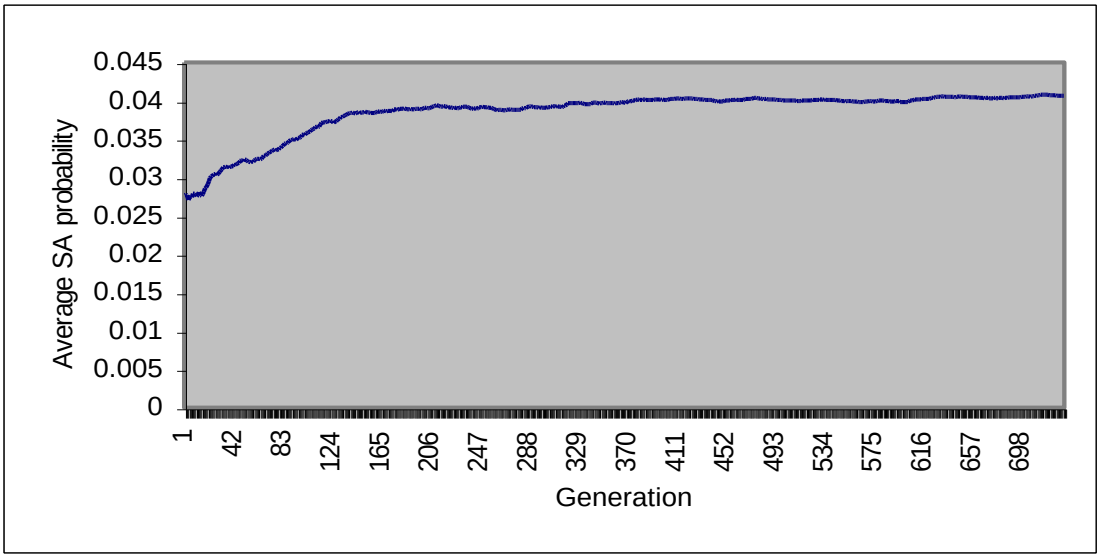


Figure 8. 28 Evolving Average SA Probability 60c

Data File 70_c

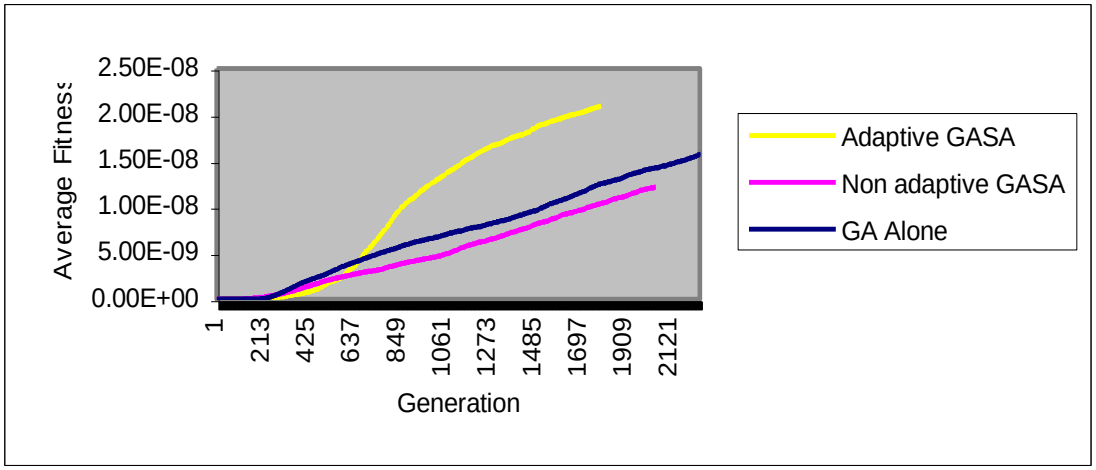


Figure 8. 29 Average Fitness Chart 70c

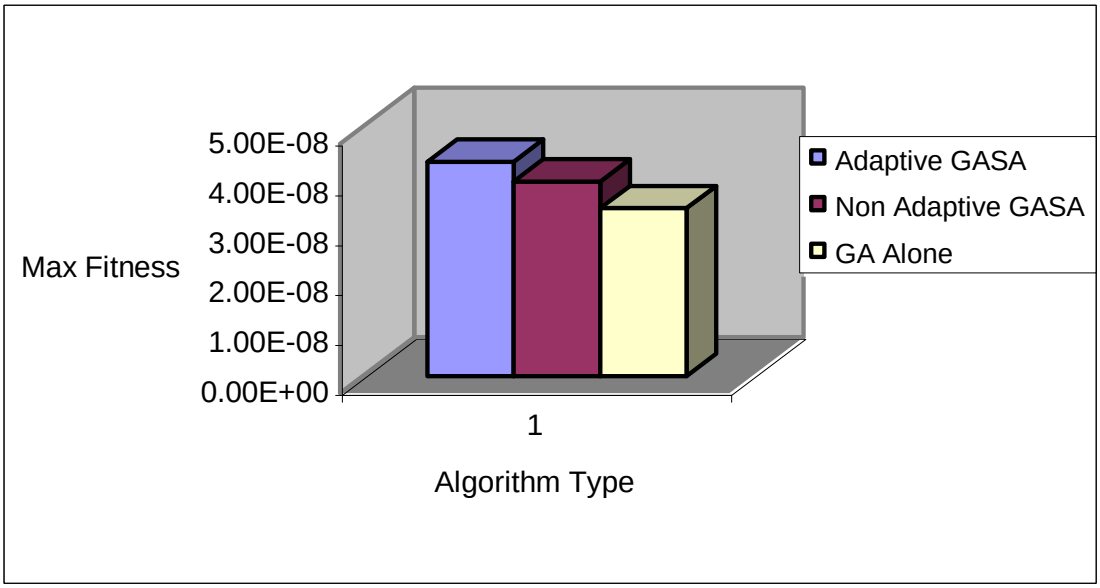


Figure 8. 30 Maximum Fitness Chart 70c

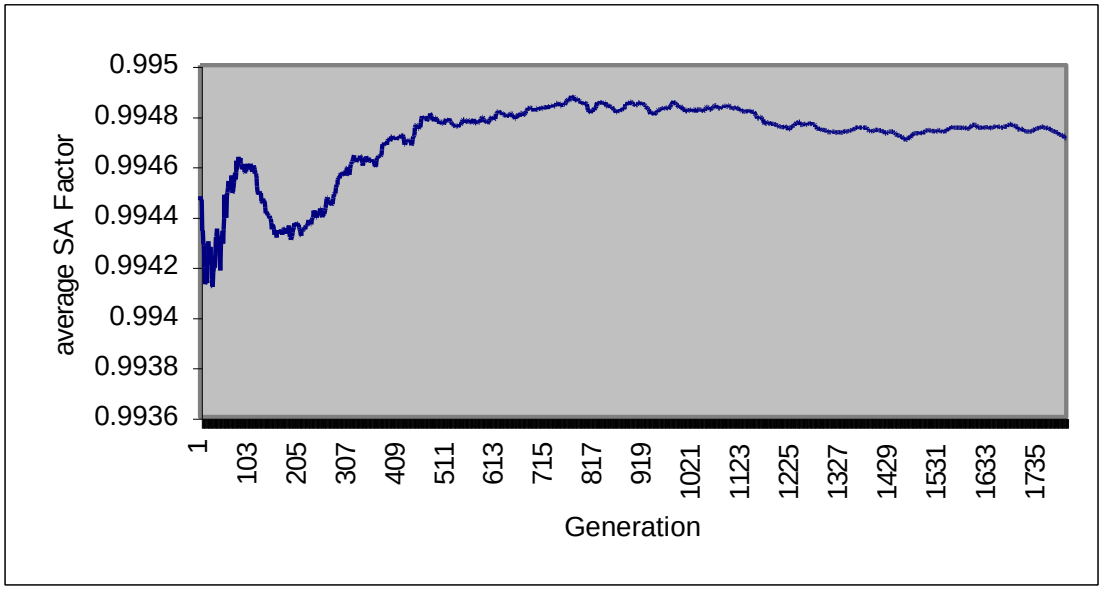


Figure 8. 31 Evolving Average SA Factor 70c

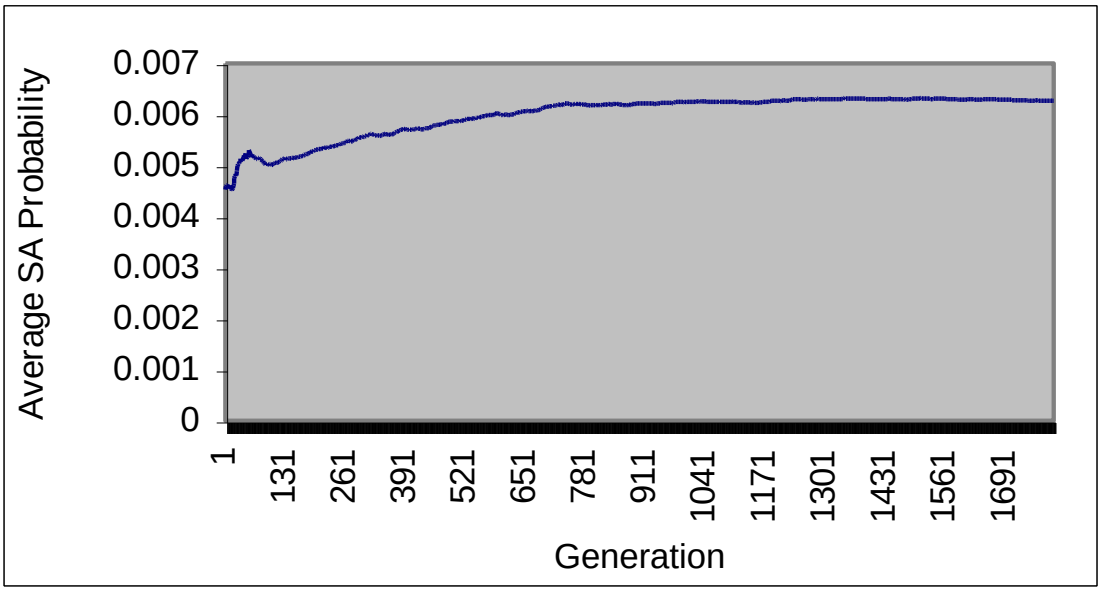


Figure 8. 32 Evolving Average SA Probability 70c

Data File 90_c

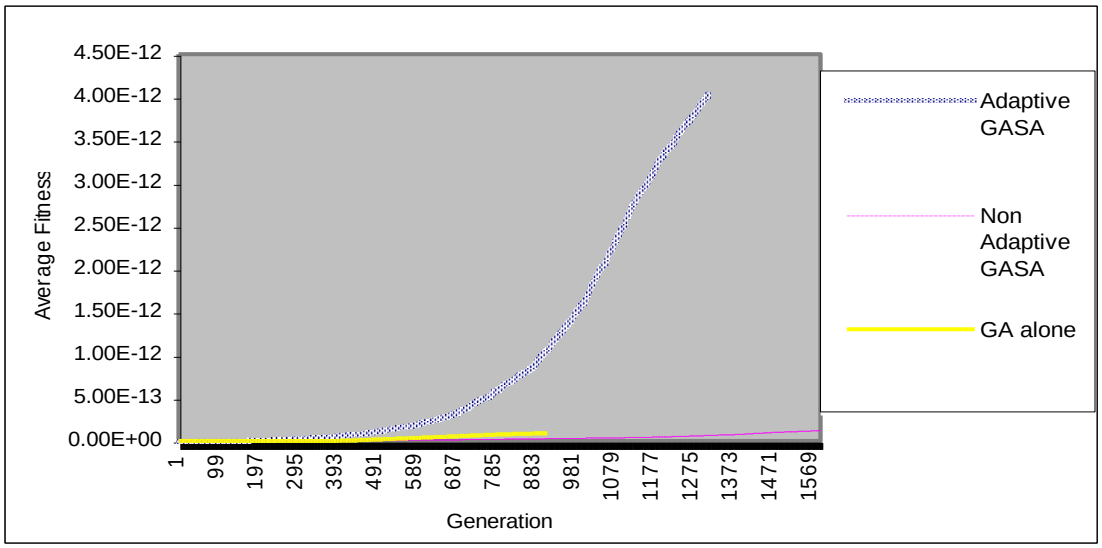


Figure 8. 33 Average Fitness Chart 90c

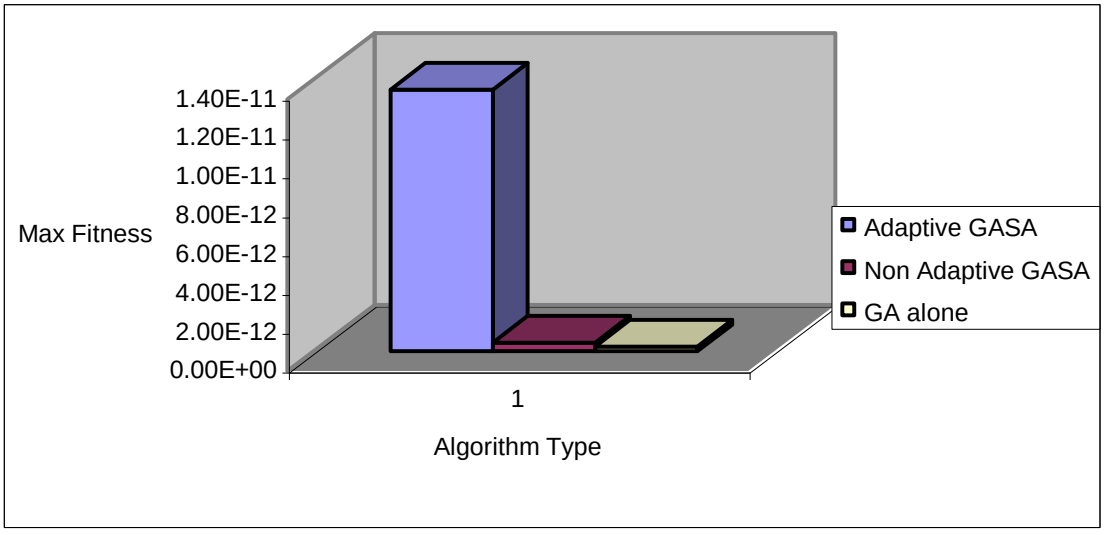


Figure 8. 34 Maximum Fitness chart 90c

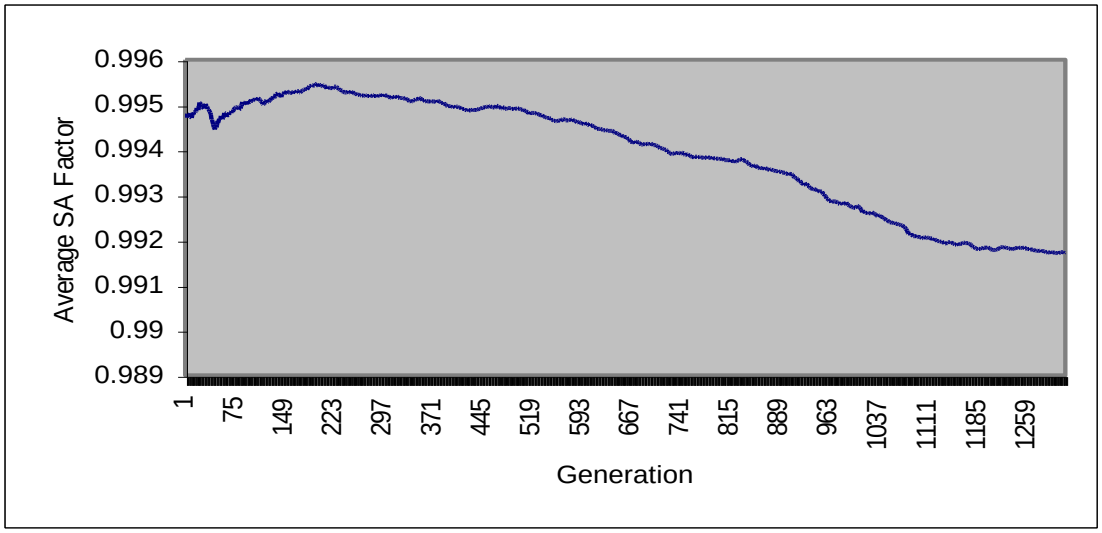


Figure 8. 35 Evolving Average SA Factor 90c

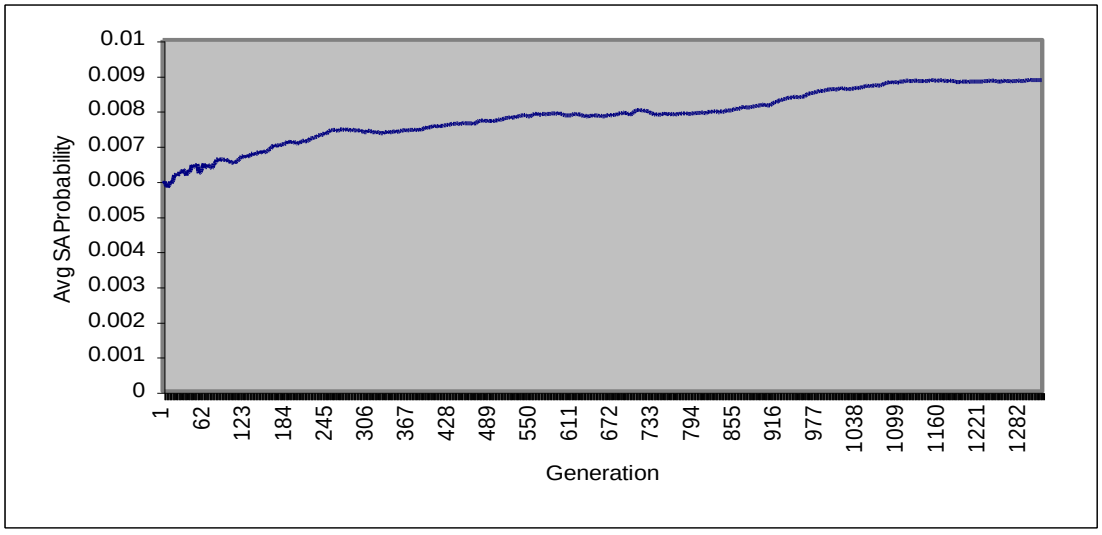


Figure 8. 36 Evolving Average SA Probability 90c

Data File 100_c

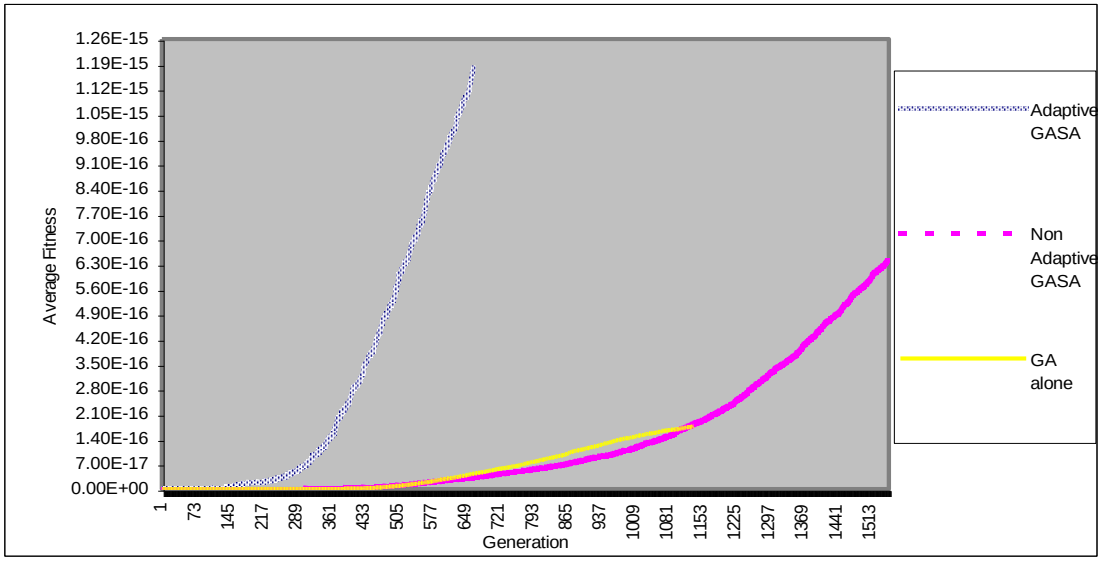


Figure 8. 37 Average Fitness Chart 100c

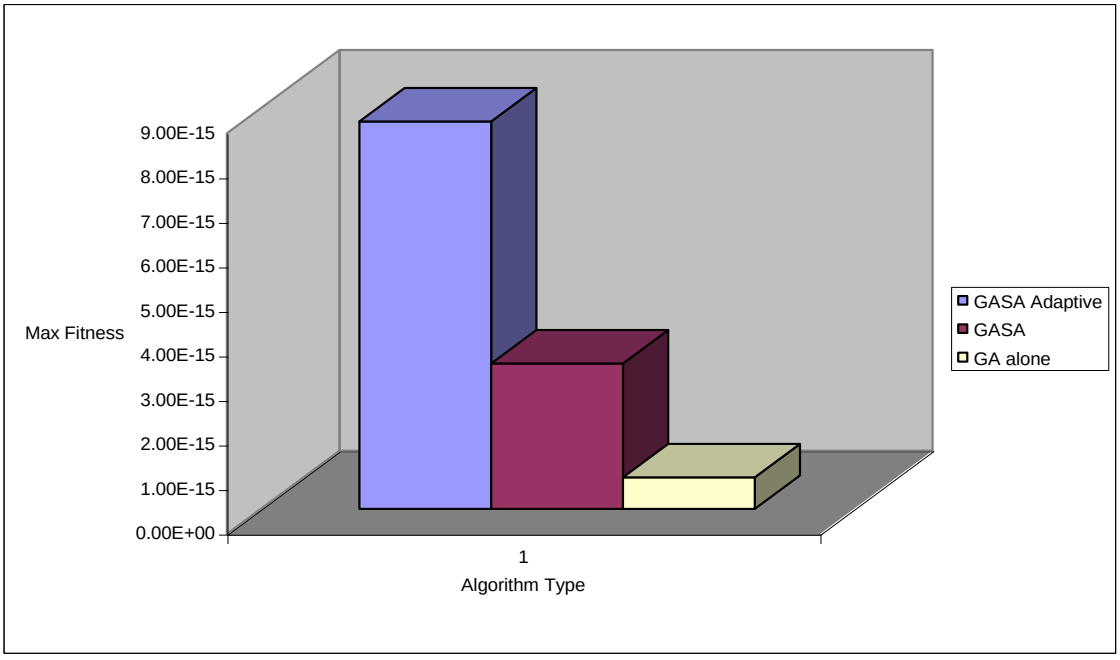


Figure 8. 38 Maximum Fitness Chart 100c

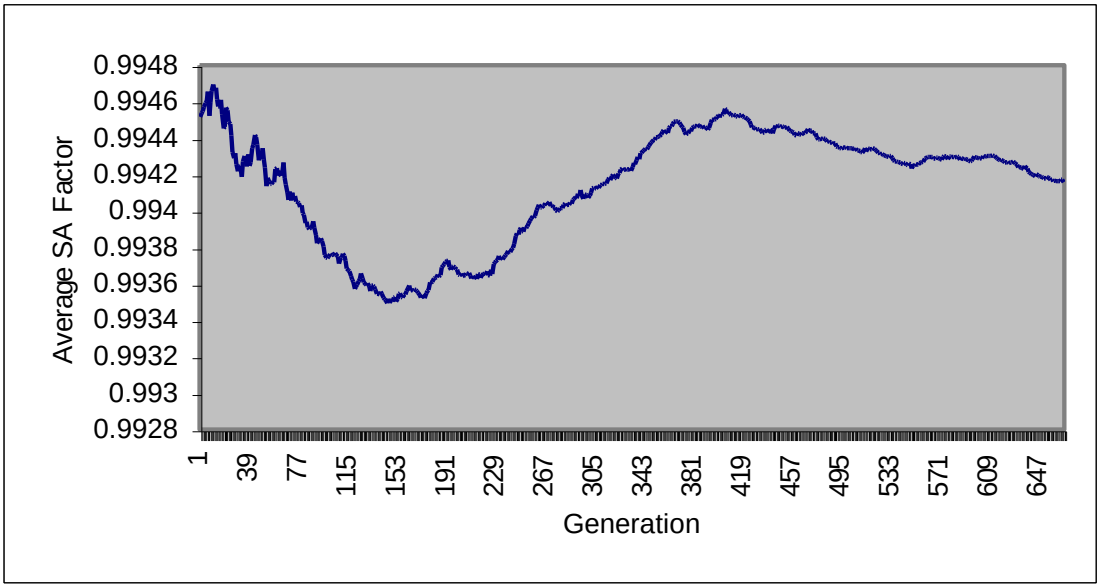


Figure 8. 39 Evolving Average SA Factor 100c

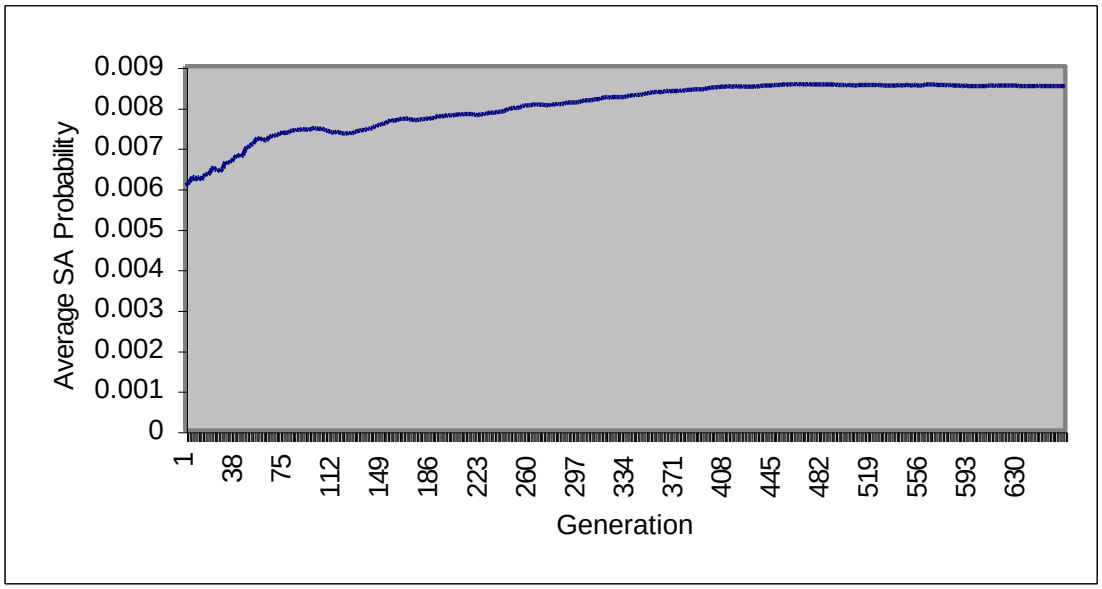


Figure 8. 40 Evolving Average SA Probability 100c

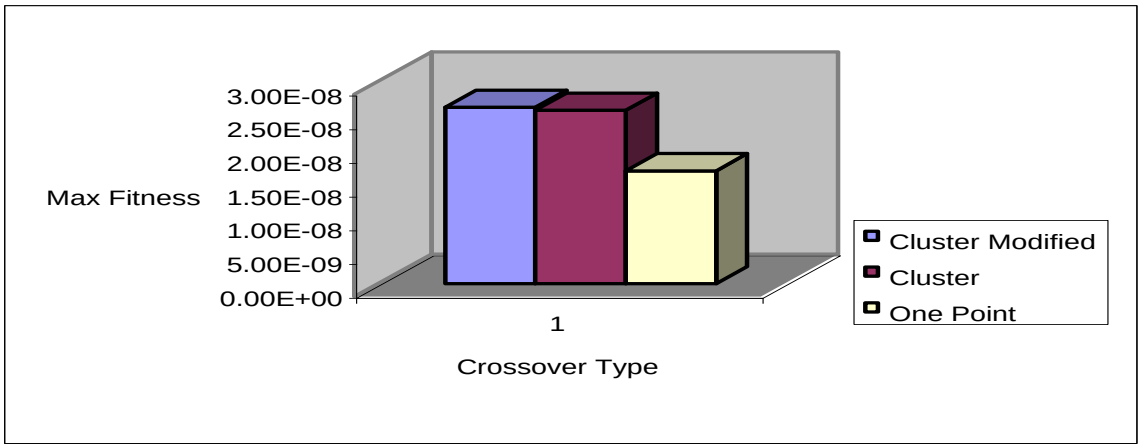


Figure 8. 41 Crossover Effect 60c

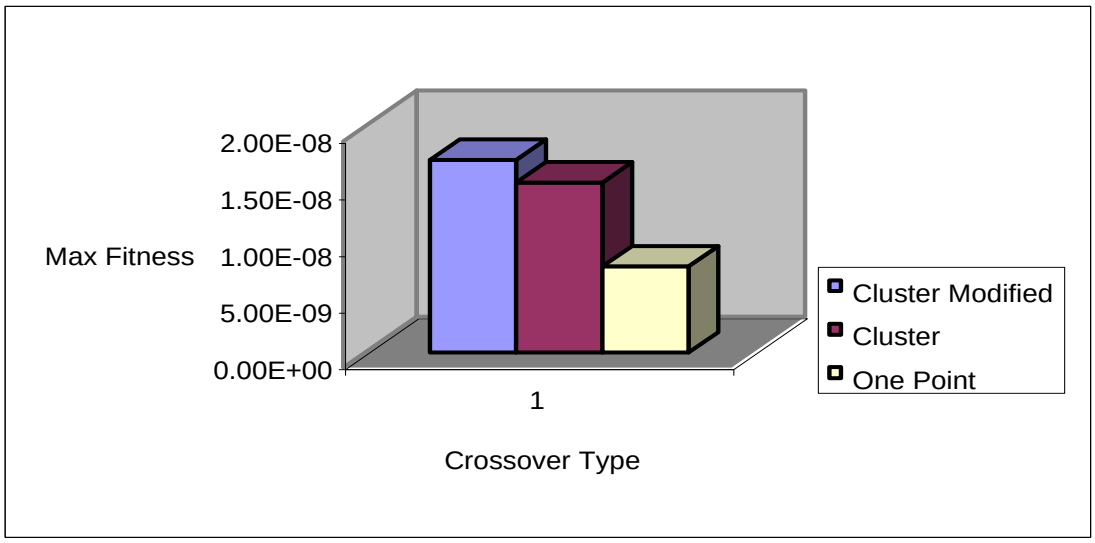


Figure 8. 42 Crossover Effect 70a

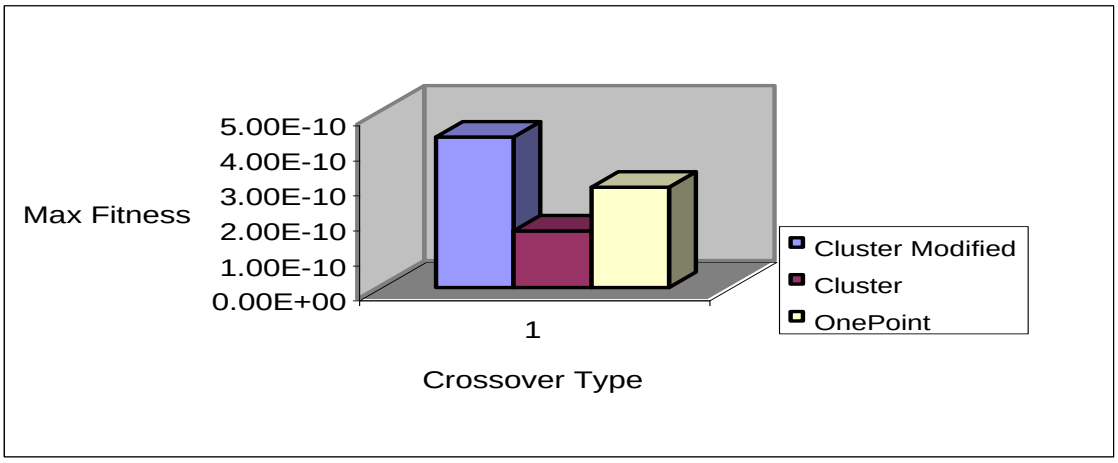


Figure 8. 43 Crossover Effect 70b

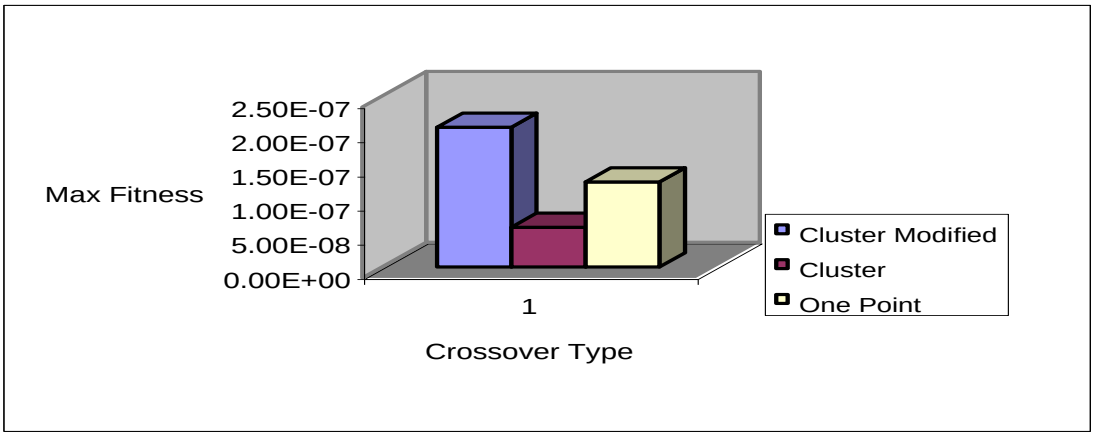


Figure 8. 44 Crossover Effect 60b

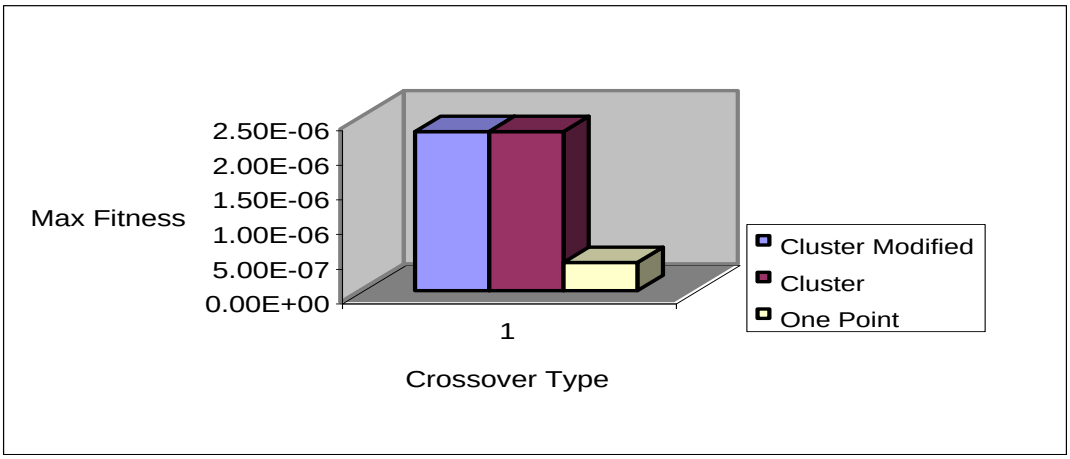


Figure 8. 45 Crossover Effect 50a

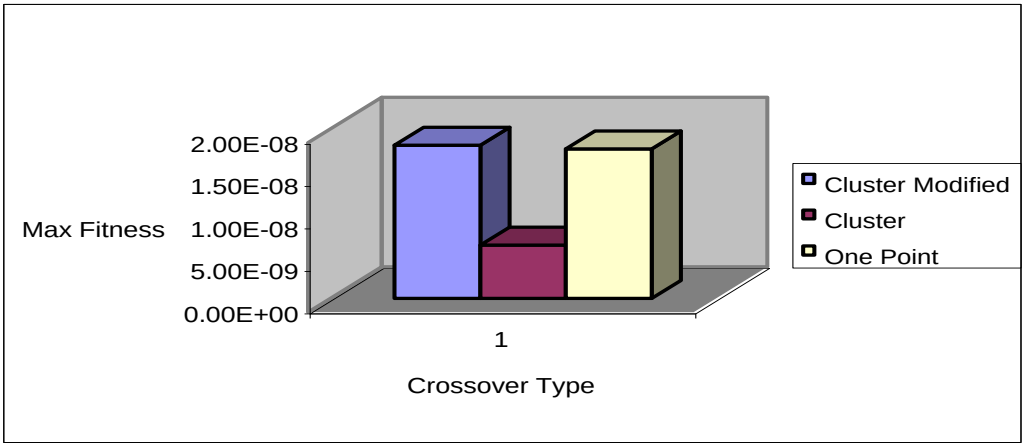


Figure 8. 46 Crossover Effect 50b

8.7 Discussion

The results obtained from comparing the three versions of the algorithm on the three data sets indicate that the adaptive GASA algorithm gives the same results as the best of the other two versions for sets A and B, while it outperforms the other two versions for set C.

For sets A and B, adding SA to the basic genetic algorithm did not offer much in terms of improving maximum fitness. In fact, in some cases the GA used alone was able to obtain better results. There could be several reasons for this observation. First, the structure of the BBN data files used in these two sets, which includes a large number of cycles, may offer more difficulty for annealing than for GA. Singly connected BBNs, in which the graph is also acyclic in the undirected sense, are easy to solve in linear time. The complexity generally increases with the number of cycles in the graph. More specifically, the complexity increases with the number of nodes that if removed would transform the graph to a singly connected graph. This set of nodes is called the node cut-set. Traditional algorithms are highly affected by the number of nodes in the node cut-set. We would expect GA to be affected similarly. However, there is no evidence that this is actually the case. Traditional GA is highly affected by gene interaction, called epistasis in GA terminology. Gene interaction in the BBN representation corresponds to high connectivity between nodes and not to the large number of cycles.

The above results, for sets A and B, indicate that having a large number of cycles did not affect the genetic algorithm in a way that would be expected. The GA alone was able to obtain results better or equal to the other two versions. Adding simulated annealing did not help in obtaining better results. In fact, in some cases degraded

performance. Adding adaptation to annealing helped to remedy the defects of annealing, and also helped a great deal in improving annealing performance in terms of improving average fitness rapidly. It did not help though in improving the final result (maximum fitness). This may indicate that the GA alone in these two sets performed so well that annealing as well as adaptation did not have much to offer.

The second possible reason that the non-adaptive GASA did not perform well in these two sets is the annealing parameters. The search space and the population size are very large in these problems. In addition, the range of fitness values changes very rapidly from one generation to another. All these factors make it difficult to find optimal annealing parameters that work well for all individuals in the population and all stages of the search process. It is inevitable that the chosen annealing parameters will not lead to improvement for some individuals and some particular situations of the search. In fact, annealing may degrade performance in these cases.

This observation is supported by noticing the improvement provided by adding adaptation to the algorithm. Besides being much easier to adjust, the adaptive parameters offered a great help in improving the performance of annealing. This improvement is clear in average fitness and also in maximum fitness to a lesser extent.

The results obtained for set C indicate that adding annealing to the basic genetic algorithm helped in obtaining better final results. Adding adaptation in this case had a very impressive effect in terms of improving both average and maximum fitness. This is particularly true for the data files with very large number of nodes (70c, 90c, 100c).

This behavior is more or less what we had expected when we started the research. Adding annealing would improve the quality of the search, and adding adaptation would

help even better. The fact that the performance met our expectations only in this set indicates that this particular data set offers a challenge for the GA alone version. The genetic algorithm alone was not able to reach results as good as the GA-SA hybrid algorithm, especially the adaptive version of it.

Data set C has two distinctive characteristics. First, the files in this set have a very large number of nodes. Second, each node has a very large number of parents. Which of these two features represented a difficulty for the GA alone?

Observing the results obtained for the first data file in this set (60c), we can see that both GA alone and adaptive GASA versions obtained the same result, while the non-adaptive GASA obtained an inferior result. This indicates that this data file did not offer much difficulty for GA alone similar to the case for sets A and B. This happened despite the fact that each node has a very large number of parents.

The other three data files (70c, 90c, 100c) are in fact the ones that were very difficult for the genetic algorithm alone. Adding annealing helped to improve the quality of the solution, but adding adaptation was really the magic spell that resulted in a great improvement in terms of both average and maximum fitness.

We can conclude from these results that, after all, the factor that actually affected the performance of GA is the number of nodes in the network. Of course, this is not surprising, since the MAP problem is NP-hard. What is surprising is that a large number of nodes in our case did not mean 30, 40, 50 or even 60 nodes, it meant something like 70, 80, 90 and 100 nodes. This number of nodes far exceeds the number of nodes tested in any previous research like (Rojas-Guzman & Kramer, 1993), (Abdelbar & Hedetniemi, 1997), and (Abdelbar & Attia, 1999).

The result obtained by Abdelbar & Attia (1999) indicated that the hybrid GASA algorithm performed better than GA alone when tested on a 50-node network. While the results obtained on 30-node and 40-node networks did not show significant improvement. Although no clear reasons were given for obtaining these results, we can now see that they do not contradict with the results obtained in this research. The fact that our results favored GA alone in many cases might be due to the improvements that had been added to the basic GA-alone algorithm in this research. We have given GA alone all the tools needed for optimal performance like large population size, an improved crossover operator, and a tool to increase diversity and avoid premature convergence.

It is very essential, however, to make an important distinction. When we talk about GA performance here we do not mean its performance in terms of obtaining the optimal result. All what concerns us here is the result obtained in comparison with the hybrid and adaptive hybrid techniques. Whether the GA is able to obtain the optimal result is still an open question.

We can thus summarize our observations by saying that for smaller networks the GA alone can do the required job of obtaining good solutions in a small amount of time. Adding annealing or adaptation in this case does not have much to offer, especially with the added processing time that they impose on the algorithm. On the other hand, obtaining good solutions for larger networks is a difficult task for GA alone. Adding annealing in this case may help in improving the quality of the solution. However, simulated annealing itself suffers from the problem of parameter adjustment. The solution that will alleviate this problem and at the same time improves the quality of the solution further is to add adaptation to the annealing parameters.

Before we finish this discussion it is important to provide some analysis for the behavior of the adaptive annealing parameters. For sets A and B the average SA factor in general tended towards decreasing in most cases. This may indicate that slower annealing in this case did not help in improving the solutions. For set C the tendency was towards increasing especially at the end of the run, which means that as we approach convergence, slower annealing gave better results, possibly because more fine search is needed. An exception was the case 90c in which the average SA factor started to decrease and continued to do so until the end of the run. One possible reason is that the process of obtaining good solutions was slow at the beginning of the run (see average fitness chart 90c figure 8.33). Faster annealing at the end of the run was sufficient to obtain an improvement of the relatively poor quality solutions obtained so far.

The average SA probability in all cases did not show a significant change. This may indicate that adaptation in this case did not have much to offer. Having a fixed annealing probability may achieve similar results.

It is also important to remember that adaptation is not restricted to these two parameters. The initial temperature value also changed adaptively for each individual solution, as explained in chapter 7. This of course also contributed to the improvement achieved in the adaptive version. Calculating the initial temperature value heuristically for each individual produces smaller initial temperature values that help to protect good solutions from disruption. Although the side effect of this is reducing diversity in the population, the task of maintaining diversity is left to the mutation and crossover operators of GA. Maintaining a balance between disruption and diversity is very critical for the performance of the algorithm.

Finally, the comparison of the different crossover operators was inconclusive. In some cases the cluster-based crossover operator performed better than the one point crossover operator. This, however, was not always the case. One point crossover performed better in set B. One possible reason is that the cluster limit may need adjustment for each individual case. The modified cluster based crossover operator performed better than both operators in most cases, which indicates the importance of increasing diversity in the population.

Chapter 9

9 Conclusions and Future Research

In this research we examined the potential of adding adaptation to a hybrid genetic-annealing approach to solving the MAP problem on Bayesian Belief Networks. In this context, annealing was used as a special case of GA mutation operator, which is intended to improve the quality of the solutions obtained by the basic genetic algorithm. Adaptation was introduced with the aim of facilitating the task of parameter adjustment for annealing, in addition to guiding the search towards better solutions.

The technique was tested by comparing three versions of the algorithm, GA alone, non-adaptive GASA and adaptive GASA, on a different BBNs with different structures. The results obtained indicate that the factor that highly affects solution quality is the number of nodes in the network. For smaller size networks GA alone can obtain good solutions in a small amount of time. Adding annealing does not help in obtaining better solutions in this case. Adding adaptation may speed up the rate of average fitness improvement, but does not help in obtaining better final results.

On the other hand, networks with a very large number of nodes represent a difficulty for GA alone. Adding annealing helps to improve the quality of the solution obtained. Adding adaptation provides a very significant improvement in terms of both average and maximum fitness.

It is thus clear that introducing adaptation has a great potential. It has offered a remedy for the defects of having fixed annealing schedule for all individuals in the population and all stages of the search. Its effect is more profound for difficult problems that GA alone cannot do much about.

Many areas in the research still need further investigation as well as improvement. First, processing time is still a major drawback for the hybrid GASA algorithm. One possible solution is to optimize the calculations used in the objective function of the solution, or to calculate the objective value for a solution only once. It is also possible to limit the maximum number of iterations performed by annealing without affecting the quality of the solution.

Second, we still need to know how far the solutions obtained are from the optimal solution. Although this task will not be easy, since it requires performing an exhaustive search for all these large networks, it will help to judge whether the solutions obtained are fair enough.

The factors that affect the performance of the GA still need further investigation. Although in our research the greatest effect was to the number of nodes in the network, there is still no sufficient evidence to eliminate the effect of other factors. Specifically, the effect the number of nodes in the node cut-set needs further testing.

The effect of problem specific crossover operators is still not clear. We need to test whether the extra cost of performing such operators is justified.

Finally, it remains to test the adaptive algorithm on other problem types. This will help in judging the robustness of the algorithm, and in deciding whether to adopt and apply it to other difficult combinatorial optimization problems.

References

1. Abdelbar, A.M. and Attia, S. "Finding Most Probable Explanations Using Simulated Annealing as a Genetic Operator", *Proceedings Third International Conference on Systemics, Cybernetics and Informatics*, Vol.8, pp. 131, Orlando, Florida 1999.

2. Abdelbar, A.M. and Hedetniemi, S.M. "A Parallel Hybrid Genetic Algorithm Simulated Annealing Approach to finding Probable Explanations on Bayesian Belief Networks", *Proceedings IEEE International Conference on Neural Networks*, Vol. I, pp. 450-455, 1997.

3. Abdelbar, A.M. and Hedetniemi, S.M. "Approximating MAPs for Belief Networks is NP-hard and Other Theorems", *Artificial Intelligence*, Vol. 102, pp. 21-38, 1998.

4. Ackley, D.H. "An empirical Study of Bit Vector Function Optimization", In Davis, L. (Ed.), Genetic Algorithms and Simulated Annealing, pp.170-204. Pitman,1987.

5. Adler, D. "Genetic Algorithms and Simulated Annealing: A Marriage Proposal", *IEEE International Conference on Neural Networks*, Vol. 2, pp. 1104-1109, 1993.

6. Attia, S.A. Applying Simulated Annealing as an Intelligent Genetic Mutation Operator for Fonding Most Probable Explanations on Bayesisn Belief Networks. MA Thesis, The American University in Cairo, School of Sciences and Engineering, Nov. 1999.

7. Baluja, Shumeet, "A Massively Distributed Parallel Genetic Algorithm," In CMU-CS-92-196R, 1992.

8. Beasley, D., Bull, D.R., and Martin, R. R. "An Overview of Genetic Algorithms: Part1, Fundamentals", *University Computing*, 15(2), pp.58-69, 1993.

9. Beasley, D., Bull, D.R., and Martin, R. R. "An Overview of Genetic Algorithms: Part2, Research Topics", *University Computing*, 15(4), pp.170-181, 1993.

10. Bland, J.A., and Dawson, G.P., "Tabu Search Applied to Layout Optimization", Report Department of Maths, Stats and O.R, Trent Polytechnic, UK, presented at CO89, Leeds, July 1989.

11. Booker, L. “Improving the Performance of Genetic Algorithms in Classifier Systems”, In Grefenstette, J.J. (ED.), *Proceedings 1st International Conference on Genetic Algorithms*, pp. 80-92, Lawrence Erlbaum Associates, 1985.

12. Boseniuk, T. and Ebeling W. “Optimization of NP-Complete Problems by Boltzmann-Darwin Strategies Including Life Cycles”, *Europhysics Letters*, 6(2), pp. 107-112, 1988.

13. Brown, D.E., Huntley, C.L. and Spillane, A.R. “A Parallel Genetic Heuristic for the Quadratic Assignment Problem”, *Proceedings 3rd International Conference on Genetic Algorithms*, pp. 406-415, October 1989.

14. Cantu-Paz, E. “ A Summary of Research on Parallel Genetic Algorithms”, In IlliGAL Technical Report No. 95007, 1995.

15. Chams, M., Hertz, A. and de Werra, D.“Some Experiments with Simulated Annealing for Colouring Graphs”, *Euoropean Journal of Operational Research*, Vol.32, pp.260-266, 1987.

16. Charniak, E.and Shimony, S.E. “Cost Based Abduction and MAP Explanation”, *Artificial Intelligence*, Vol. 66, pp.345-347, 1994.

17. Chen, H., Flann, N.S. and Watson, D. “Parallel Genetic Simulated Annealing: A Massively Parallel SIMD Algorithm”, *IEEE Transactions on Parallel and Distributed Systems*, Vol.9, pp.126-136, 1998.

18. Cho, H. J., Oh, S.Y. and Choi, D.H. “A New Evolutionary Programming Approach Based on Simulated Annealing with Local Cooling Schedule*”, IEEE Press*, pp.598-602, 1998.

19. Connolly, D.T. “An Improved Annealing Scheme for the QAP”, *Euoropean Journal of Operational Research,*Vol. 46, pp. 93-100, 1988.

20. Davidor, Y. “A Naturally Occuring Niche and Species Phenomenon: the Model and First Results ”, In Belew, R.K. and Booker, L.B. (EDs.), *Proceedings 4th International Conference on Genetic Algorithms*, pp. 257-263 , Morgan Kaufmann, 1991.

21. Davis T.E. and Principe, J.C. “A Simulated Annealing Like Convergence Theory for the Simple Genetic Algorithm”, *Proceedings 4th International Conference on Genetic Algorithms*, pp. 174-181, 1991.

22. Davis, L. “Adapting Operator Probabilities in Genetic Algorithms”, In Schaffer, J.D. (ED.), *Proceedings 3rd International Conference on Genetic Algorithms*, pp. 61-69 , Morgan Kaufmann, 1989.

23. Davis, L. “Bit Climbing, Representational Bias and Test Suite Design ”, In Belew, R.K. and Booker, L.B. (EDs.), *Proceedings 4th International Conference on Genetic Algorithms*, pp. 18-23 , Morgan Kaufmann, 1991.

24. Davis, L. “Job Shop Scheduling with Genetic Algorithms”, In Grefenstette, J.J. (ED.), *Proceedings 1st International Conference on Genetic Algorithms*, pp. 136-140, Lawrence Erlbaum Associates, 1985.

25. Davis, L. Handbook of Genetic Algorithms. Van Nostrand, 1991.

26. Dejong, K. “The Analysis and Behavior of a Class of Genetic Adaptive Systems”, Phd thesis, University of Michigan, 1975.

27. Durand, M.D. “Parallel Simulated Annealing: Accuracy vs. Speed in Placement”, *IEEE design and test of Computers,* 6(3), pp. 8-34, 1989.

28. Eglese, R.W. “Simulated Annealing: A Tool for Operational Research”, *European Journal of Operational Research,* Vol. 46 pp. 271-281, 1990.

29. Esbensen, H. “A Genetic Algorithm for Macro Cell Placement”, Proceedings of the European Design Automation Conference, pp.52-57, FRG, 1992.

30. Esbensen H. And Mazumder, P. “SAGA: A Unification of the Genetic Algorithm with Simulated Annealing and its Application to Macro Cell Placement”, *Proceedings 7th International Conference on VLSI Design,* pp. 211-214, 1994.

31. Glover, F. “Tabu Search:1”, *ORSA Journal on Computing,* Vol. 1(3), pp.190-206.

32. Glover, F. and Greenberg, H.J. “New Approaches for Heuristic Search: a Bilateral Linkage with Artificial Intelligence”, *Euoropean Journal of Operational Research,* Vol. 39, pp. 119-130, 1989.

33. Goldberg, D.E. Genetic Algorithms in Search Optimization and Machine Learning. New York: Addison- Wesley, 1989.

34. Goldberge, D.E. “A Note on Boltzmann Tournament Selection for Genetic Algorithms and Population Oriented Simulated Annealing”, *Complex Systems,* pp. 445-460, 1990.

35. Goodman, E.D., Lin S-C. And Punch, W.F.“Coarse-Grained Parallel Genetic Algorithms: Categorization and New Approach”, *Parallel and Distributed Processing,* Oct. 1994.

36. Green, J.W. and Supowit, K.J. “Simulated Annealing Without Rejected Moves”, *IEEE Transactions on Computer Aided Design.* Vol 5, pp. 221-228, 1986.

37. Grefenstette, J.J. “Incorporating Problem Specific Knowledge into Genetic Algorithms”, In Davis, L. (Ed.), Genetic Algorithms and Simulated Annealing, pp.42-60, Pitman, 1987.

38. Groover, L.K. “A New Simulated Annealing Algorithm for Standard Cell Placement”, *Proc IEEE International Conference on Computer Aided Design,* Santa Clara, pp. 378-380, 1986.

39. Hajek, B. “Cooling Schedules for Optimal Annealing”, *Mathematics of Operations Research,* Vol. 13, pp. 311-329, 1988.

40. Haralick, R. and Elliot, G. “Increasing Tree Search Efficiency for Constraint-Satisfaction Problems”, *Artificial Intelligence*, Vol. 14(3): pp.263-313, 1980.

41. Haykin, S. Neural Networks: a Comprehensive Foundation, New Jersey: Prentice-Hall, 1999.

42. Hertz, A. and de Werra, D. “Using Tabu Search Techniques for Graph Coloring”, *Computing*, Vol. 39, pp.345-351.

43. Holland J.H. Adaptation in Natural and Artificial Systems. MIT Press, 1975.

44. Holland, J. H., Booker, L.B And Goldberg, D. E. “Classifier Systems and Genetic Algorithms”, *Artificial Intelligence*, Vol. 40, pp. 235-282, 1989.

45. Jhonson, D.S., Aragon, C.R., McGeogh, L.A. and Schevon, C., “Optimization by Simulated Annealing: An Experimental Evaluation, Part I”, AT&T Bell Laboratories, Murray Hill, NJ, Preprint, 1987.

46. Jhonson, D.S., Aragon, C.R., McGeogh, L.A. and Schevon, C., “Optimization by Simulated Annealing: An Experimental Evaluation, Part II”, AT&T Bell Laboratories, Murray Hill, NJ, Preprint, 1988.

47. Kirpatrick, S., Gelatt, C.D. and Vecchi, M.P. “Optimization by Simulated Annealing”, *Science,* Vol. 220, pp.134-141, 1983.

48. Koakutsu, S., Kang M. And Dai, W.W. “Genetic Simulated Annealing and Application to Non-Slicing Floor Plan Design”, *Proceedings 5th ACM/SIGDA Physical Design Workshop*, pp. 134-141, 1996.

49. Lin, F.T., Kao, C.Y. and Hsu, C.C. "Incorporating Genetic Algorithms into Simulated Annealing", *Proceedings 4th International Symposium on Artificial Intelligence*, pp.290-297, 1991.

50. Lundy, M., and Mees, A. "Convergence of an Annealing Algorithm", *Mathematical Programming*, Vol. 34, pp.111-124, 1986.

51. Mahfoud S.W. and Goldberg, D. "Parallel Recombinative Simulated Annealing: A Genetic Algorithm" IlliGAL Report No *9200*, 1993.

52. Metropolis, N. Rosenbluth, A., Rosenbluth, M., Teller, A. and Teller E. "Equation of State Calculations by Fast Computing Machines", *Journal of Chemical Physics*, Vol. 21, pp. 1087-1092, 1953.

53. Michalewicz, Z. Genetic Algorithms + Data Structures = Evolution Programs, London: Springer-Verlag,1992.

54. Mitra, D., Rmeo, F. and Sangiovanni-Vincentelli, A.L. "Convergence of Finite Time Behaviour of Simulated Annealing", *Advances in Applied Probability*, Vol. 18, pp. 747-771, 1986.

55. Pearl, J. Heuristics: Intelligent Search Strategies for Computer Problem Solving, Massachusetts: Addison-Wesley, 1984.

56. Pearl, J. Probabilistic Reasoning in Intelligent Systems: Network of Plausible Inference, San Mateo: Morgan-Kaufmann, 1988.

57. Rojas-Guzman, C. and Kramer, M.A. "GALGO: A Genetic Algorithm Decision Support Tool for Complex Uncertain Systems Modeled with Bayesian Belief Networks", *Proceedings 9th International Conference on Tools with Artificial Intelligence*, pp. 368-375, 1993.

58. Russell, S. and Norvig, P. Artificial Intelligence: A Modern Approach. New Jersey: Prentice-Hall, 1995.

59. Sinclair, M.C. "Minimum Cost Routing and Wavelength Allocation Using a Genetic Algorithm/Heuristic Hybrid Approach" *Proceedings 6th IEEE Conf. On Telecommunications*, Edinburgh, UK, March/April 1998.

60. Stender, J. Parallel Genetic Algorithms: Theory & Applications, Amesterdam: IOS Press, 1993.

61. Stone, H.S. and Stone, J. "Efficient Search Techniques: an Empirical Study of the *n*-queens Problem". Technical Report RC 12057, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, 1986.

62. Tovey, C.A. “Simulated Simulated Annealing”, *American Journal of Mathematics and Managment Sciences*, Vol. 8, pp. 389-407, 1988.

63. Wall, Matthew, “GAlib 2.44.”In http://lancet.mit.edu/ga. Massachusetts Institute of Technology (MIT) Laboratory of Genetic Algorithms, 1999.

64. Wright, M.B. “Applying Stochastic Algorithms to a Locomotive Scheduling Problem”, *Journal of the Operational Research Society*, Vol. 40, pp.187-192.

APPENDIX A: Source Code

Source Code for GA alone

The declaration file “declaration.h”

```
#ifndef DECLARATION_H
#define DECLARATION_H
const int   MAXNODES = 70;
const int   MAXPARENTS= 15;
const int   MAXPROB  = 1<<MAXPARENTS;
#define     NETFILE "60_a_nod.net"
#define     SCOREFILE "scorefile_60a.out"
#define     RESFILE "myres_60a.out"
#define     TRUE  1
#define     FALSE 0

typedef int ParentArray[MAXPARENTS];
typedef float ProbArray[MAXPROB];
typedef struct NodeType
{
      int NumParents;
      ParentArray Parents;
      ProbArray   Prob;
};
typedef NodeType BeliefNetType[MAXNODES];

typedef int CostArray [MAXNODES][MAXNODES];
CostArray Cost;


int PopSize=1000;
int GenToConverge=500 // declare convergence after this number of
generations
float ProbMutation = 0.02;
float ProbCrossover=0.99;
float PercentReplace=0.2; //replace 20% of population each generation
float Cluster_Limit =1;

int     GenNum=0;
int   Infinity;
float FitArray[1000];

#endif
```

The main program for the GA alone version “GA.cpp”

```
#include <stdio.h>
#include "iostream.h"
#include <fstream.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include"ga.h"
#include"GARealGe.h"
#include"GARealGe.cpp"
#include "declaration.h"

int NumNodes;

BeliefNetType Net;
void ReadNet();
void FindCost();
void FindShortPaths();
float Objective(GAGenome&);
void MyInitializer(GAGenome&);
int FlipMutator(GAGenome& ,float );
int  MyCrossover(const  GAGenome& , const GAGenome& , GAGenome* ,
GAGenome* );
GAAlleleSetArray<float> allelset;
ofstream ScoreFile(SCOREFILE);

int main()
{
      unsigned int seed=5;
      GARandomSeed(seed);
      ReadNet();
      FindCost();
      FindShortPaths();

      for (int k=0; k<NumNodes; k++)
                  allelset.add(FALSE,TRUE,1);

            //define the genome as a 1d array of real with the values
            // of genes derived from the allelset array
            GA1DArrayAlleleGenome<float> genome(allelset,Objective);

            //define the GA methods and parameters
            genome.initializer(MyInitializer);
            genome.mutator(FlipMutator);
            genome.crossover(MyCrossover);
            GASteadyStateGA ga(genome);
            ga.populationSize(PopSize)  ;
            ga.nConvergence(GenToConverge);
            ga.terminator(GAGeneticAlgorithm::TerminateUponConvergence)
            ;
            ga.pMutation (ProbMutation);
            ga.pCrossover(ProbCrossover);
            ga.pReplacement(PercentReplace);
```

```
            //get current CPU time
            clock_t CPU_time;
            CPU_time=clock();
            ScoreFile<<"generation \t mean \t max \t min \t deviation
            \t diversity \n";
            ga.scoreFilename(SCOREFILE);
            ga.scoreFrequency(1); //keep the scores of every
            //generation
            ga.flushFrequency(1); //flush scores every generation
            ga.selectScores(GAStatistics::AllScores);
            ga.initialize();
            while (!ga.done())
            {
                  GenNum= ga.statistics().generation();

                  GAPopulation current_pop = ga.population();

                  for (int num1=0;num1<PopSize;num1++)

                  {
                        FitArray[num1]=Objective(current_pop.individual
                        (num1));
                  }
                  ga.step();

            }
            ga.flushScores();

            CPU_time=clock() - CPU_time;
            cout<<" \n\n THE GA FOUND : "
            <<ga.statistics().bestIndividual() << "\n";
            cout<<ga.statistics().bestIndividual().score() << "\n\n";
            cout<<"Processing Time=  "<<(CPU_time/1000)/60<<"  mins\n";

            //write results to file
            ofstream ResultFile;
            ResultFile.open(RESFILE,ios::app);//append the results
            ResultFile<<"PopSize   = "<<PopSize<<"\t";
            ResultFile<<"NumGenerations= "<<GenNum+1<<"\t";
            ResultFile<<"ProbMutation= "<<ProbMutation<<"\t";
            ResultFile<<"ProbCrossover= "<<ProbCrossover<<"\n";
            ResultFile<<"ClusterLimit= "<<Cluster_Limit<<"\t";
            ResultFile<<"GenToConverge= "<<GenToConverge<<"\t";
            ResultFile<<"Bestindividual =
            "<<ga.statistics().bestIndividual()<<"\n";
            ResultFile<<"Processing Time=  "<<(CPU_time/1000)/60<<"
            mins\n";
            ResultFile<<"BestScore  =
            "<<ga.statistics().bestIndividual().score()<<"\n";


      ScoreFile.close();

      return 0;
}
//////////////////////////////////////////////////
```

```
GABoolean MyTerminator(GAGeneticAlgorithm & ga)
{
  if(ga.statistics().current(GAStatistics::Maximum) ==
     ga.statistics().current(GAStatistics::Minimum)  )
    return gaTrue;
  else
    return gaFalse;

}
////////////////////////////////////////////
void MyInitializer(GAGenome& g)
{

      GA1DArrayAlleleGenome<float>& genome
=(GA1DArrayAlleleGenome<float>&)g;

      for (int i=0;i<NumNodes;i++)
            genome.gene(i, float(GARandomInt(0,1)));

}
////////////////////////////////////////////////////
void ReadNet()
{
ifstream InFile(NETFILE);
if (!InFile)
    {
       cout<<"Could Not Read Input File"<<NETFILE<<"\n";
       exit(1);
    }
InFile>>NumNodes; //read number of nodes
for (int node=0; node<NumNodes; node++)
{
      InFile>>Net[node].NumParents; // read number of parents for this
node
      for (int p=0; p<Net[node].NumParents;p++) // read parent numbers
for this node
      {
            InFile>>Net[node].Parents[p];
            Net[node].Parents[p]--; // decrement parent number to start
from 0 instead of 1
      }//end for p
      int   probs = (1<< Net[node].NumParents) ; // calculate number of
entries in prob table
  for (int pr=0; pr<probs; pr++)  //read probabilities
         InFile>> Net[node].Prob[pr];

} //end for node
InFile.close();

}// end ReadNet



////////////////////////////////////////////////
```

```
void FindCost()
{

Infinity=NumNodes;
//initialize Cost Array
for(int i=0;i<MAXNODES;i++)
      for(int j=0;j<MAXNODES;j++)
      {
            if(i==j) Cost[i][j]=0;
            else Cost[i][j]=Infinity;
      }


for (int node=0;node<NumNodes;node++)
      {

            for(int p=0;p<Net[node].NumParents;p++)
            {

                  int this_parent =Net[node].Parents[p];
                        Cost[this_parent][node]=1;

            }
      }
}

////////////////////////////////////////////////
int min (int val1, int val2)
{
      int minimum;
      if(val1<=val2) minimum=val1;
      else           minimum=val2;
return minimum;
}
////////////////////////////////////////////////

void FindShortPaths() //find the shortest path between all nodes in the
net
{

for (int k=0;k<NumNodes;k++)
      for (int i=0;i<NumNodes;i++)
       for (int j=0;j<NumNodes;j++)
            Cost[i][j]=min(Cost[i][j],Cost[i][k]+Cost[k][j]);

}

////////////////////////////////////////////////
////////////////////////////////////////////////



float Objective(GAGenome& g) //calculate the objective value of the
genome
{
      float score=1.0;
```

```
GA1DArrayAlleleGenome<float> genome =(GA1DArrayAlleleGenome<float>&) g;

for (int node=0; node<NumNodes;node++)
      {
          int power=1;
            int index=0;
            int node_truth = (int) genome.gene(node);
            for(int p= Net[node].NumParents -1; p>=0 ; p--)
            {
            int parent_num= Net[node].Parents[p];
                  int parent_truth = int (genome.gene(parent_num));
                   index = index + parent_truth*power;
                   power = power*2;
            }//end for p

            if (node_truth== TRUE)
                  score = score*Net[node].Prob[index];
            else score = score* (1- Net[node].Prob[index]);
      }//end for node

      return score;
}



//////////////////////////////////////////////////

//////////////////////////////////////////////////

int FlipMutator(GAGenome& g, float pmut)
{

      GA1DArrayAlleleGenome<float>& child
=(GA1DArrayAlleleGenome<float>&)g;
      if (pmut<=0.0) return(0);
      int nMut=0;
      for (int i=0; i<NumNodes; i++)
      {
            if (GAFlipCoin(pmut))
            {
              child.gene(i, ((child.gene(i) == 0) ? 1 : 0));
              nMut++;
            }

      }

return(nMut);
}

//////////////////////////////////////////////

int  MyCrossover(const  GAGenome& g1, const GAGenome& g2, GAGenome* c1,
GAGenome* c2)
{

GA1DArrayAlleleGenome<float>& mom =(GA1DArrayAlleleGenome<float>&)g1;
```

```
GA1DArrayAlleleGenome<float>&dad=(GA1DArrayAlleleGenome<flo)g2;
GA1DArrayAlleleGenome<float>&
child1=(GA1DArrayAlleleGenome<float>&)*c1;
      GA1DArrayAlleleGenome<float>& child2
=(GA1DArrayAlleleGenome<float>&)*c2;
//select a random root
int root=GARandomInt(0,NumNodes-1);
child1.copy(mom); //child1 is a copy of parent1
child2.copy(dad); //child2 is a copy of parent2
// now exchange the cluster the random root betweeen
// the children


for (int i=0;i<NumNodes;i++)
{
      if (Cost[root][i]<=Cluster_Limit)
  {
        child1.gene(i,dad.gene(i));
        child2.gene(i,mom.gene(i));
  }//end if

}//for

// if the children have duplicates in the population , force a change
in them
float fit1=Objective(child1);
float fit2=Objective(child2);

for (int count=0;count<PopSize;count++)
{
      if (fit1==FitArray[count])
            FlipMutator(child1,ProbMutation*2);
      if (fit2==FitArray[count])
            FlipMutator(child2,ProbMutation*2);
}//for

      return 1;
}


////////////////////////////////////////////////

////////////////////////////////////////////////
```

Source Code for Non-Adaptive GASA

Declaration File “declaration.h”

```
#ifndef DECLARATION_H
#define DECLARATION_H
const int   MAXNODES = 70;
const int   MAXPARENTS= 15;
const int   MAXPROB  = 1<<MAXPARENTS;
#define     NETFILE "60_a_nod.net"
#define     SCOREFILE "scorefile_60a.out"
#define     RESFILE "myres_60a.out"
#define     TRUE  1
#define     FALSE 0
#define     MAXSTAGNATION   500 // no. Of iterations required to
                declare stagnation in SA
#define           PROB_SA     0.005 // probability of   performing SA
#define           SAFACTOR      0.993 // reduction factor for SA
#define           INIT_TEMP   0.00005 // initial temperature for SA
typedef int ParentArray[MAXPARENTS];
typedef float ProbArray[MAXPROB];
typedef struct NodeType
{
      int NumParents;
      ParentArray Parents;
      ProbArray   Prob;
};
typedef NodeType BeliefNetType[MAXNODES];
typedef CostArray [MAXNODES][MAXNODES];
typedef int DescendArray[MAXNODES][MAXNODES] ;
CostArray Cost;
CostArray Children;
int Child_Count[MAXNODES];
int Processed[MAXNODES];
DescendArray Descend;
DescendArray GrandParents;
int PopSize=1000;
int GenToConverge=500; // to converge look back this number of
generations
float ProbMutation = 0.02;
float ProbCrossover=0.99;
float PercentReplace=0.2; //replace 20% of population each generation
float Cluster_Limit =1;
float Psam= 0.01; // probability of flipping a bit when creating a
random move during SA
int     GenNum=0;
int   Infinity;
int NMut;
float FitArray [1000];
#endif
```

The main program for the GASA non adaptive version “GASA.cpp”

```
#include <stdio.h>
#include "iostream.h"
#include <fstream.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include"ga.h"
#include"GARealGe.h"
#include"GARealGe.cpp"
#include "declaration.h"
int NumNodes;

BeliefNetType Net;
void ReadNet();
void FindCost();
void FindShortPaths();
float Objective(GAGenome&);
void MyInitializer(GAGenome&);
int MyMutation(GAGenome& ,float );
int  MyCrossover(const  GAGenome& , const GAGenome& , GAGenome* ,
GAGenome* );

      GAAlleleSetArray<float> allelset;
      ofstream ScoreFile(SCOREFILE);


int main()
{
      unsigned int seed=5;
      GARandomSeed(seed);
      ReadNet();
      FindCost();
      FindShortPaths();


            for (int k=0; k<NumNodes; k++)
                  allelset.add(FALSE,TRUE,1);
            //define the genome as a 1d array of real with the values
            // of genes derived from the allelset array
            GA1DArrayAlleleGenome<float> genome(allelset,Objective);

            genome.initializer(MyInitializer);
            genome.mutator(MyMutation);
            genome.crossover(MyCrossover);
            GASteadyStateGA ga(genome);
            ga.populationSize(PopSize)  ;

            //stop when the score of the best genome has
            //not changed for the specified number of generations
            ga.nConvergence(GenToConverge);

      ga.terminator(GAGeneticAlgorithm::TerminateUponConvergence);
            ga.pMutation (ProbMutation);
```

```
            ga.pCrossover(ProbCrossover);
            ga.pReplacement(PercentReplace);


            //get current CPU time
            clock_t CPU_time;
            CPU_time=clock();
            ScoreFile<<"generation \t mean \t max \t min \t deviation
\t diversity \n";

            ga.scoreFilename(SCOREFILE);
            ga.scoreFrequency(1); //keep the scores of every generation
            ga.flushFrequency(1); //flush scores every generation
            ga.selectScores(GAStatistics::AllScores);
            ga.initialize(seed);
            //ga.evolve(); //start processing
            while (!ga.done())
            {
              GenNum= ga.statistics().generation();
                  GAPopulation current_pop = ga.population();

                  for (int num1=0;num1<PopSize;num1++)

                  {

      FitArray[num1]=Objective(current_pop.individual(num1));
                  }
              ga.step();

            }
            ga.flushScores();

            CPU_time=clock() - CPU_time;
            cout<<" \n\n THE GA FOUND : "
<<ga.statistics().bestIndividual() << "\n";
            cout<<ga.statistics().bestIndividual().score() << "\n\n";
            cout<<"Processing Time=  "<<(CPU_time/1000)/60<<"  mins\n";

            //write results to file
            ofstream ResultFile;
            ResultFile.open(RESFILE,ios::app);//append the results
            ResultFile<<"PopSize   = "<<PopSize<<"\t";
            ResultFile<<"InitTemp= "<<INIT_TEMP<<"\t";
            ResultFile<<"NumGenerations= "<<GenNum+1<<"\t";
            ResultFile<<"ProbMutation= "<<ProbMutation<<"\t";
            ResultFile<<"ProbCrossover= "<<ProbCrossover<<"\n";
            ResultFile<<"MaxStagnation= "<<MAXSTAGNATION<<"\t";
            ResultFile<<"ProbSA= "<<PROB_SA<<"\t";
            ResultFile<<"SAFactor= "<<SAFACTOR<<"\n";
            ResultFile<<"ClusterLimit= "<<Cluster_Limit<<"\t";
            ResultFile<<"GenToConverge= "<<GenToConverge<<"\t";
            ResultFile<<"Psam= "<<Psam<<"\n";
            ResultFile<<"Bestindividual =
            "<<ga.statistics().bestIndividual()<<"\n";
            ResultFile<<"Processing Time=  "<<(CPU_time/1000)/60<<"
            mins\n";
```

```
            ResultFile<<"BestScore  =
            "<<ga.statistics().bestIndividual().score()<<"\n";

  ScoreFile.close();
      return 0;
}
////////////////////////////////////////////////
void MyInitializer(GAGenome& g)
{

      GA1DArrayAlleleGenome<float>& genome
=(GA1DArrayAlleleGenome<float>&)g;

      for (int i=0;i<NumNodes;i++)
            genome.gene(i, float(GARandomInt(0,1)));

}

////////////////////////////////////////////////////
void ReadNet()
{
ifstream InFile(NETFILE);
if (!InFile)
    {
       cout<<"Could Not Read Input File"<<NETFILE<<"\n";
       exit(1);
    }
InFile>>NumNodes; //read number of nodes
for (int node=0; node<NumNodes; node++)
{
      InFile>>Net[node].NumParents; // read number of parents for this
node
      for (int p=0; p<Net[node].NumParents;p++) // read parent numbers
for this node
      {
            InFile>>Net[node].Parents[p];
            Net[node].Parents[p]--; // decrement parent number to start
from 0 instead of 1
      }//end for p
      int   probs = (1<< Net[node].NumParents) ; // calculate number of
entries in prob table
  for (int pr=0; pr<probs; pr++)  //read probabilities
         InFile>> Net[node].Prob[pr];

} //end for node
InFile.close();

}// end ReadNet



////////////////////////////////////////////////

void FindCost()
{

Infinity=NumNodes;
```

```
//initialize Cost Array
for(int i=0;i<MAXNODES;i++)
      for(int j=0;j<MAXNODES;j++)
      {
            if(i==j) Cost[i][j]=0;
            else Cost[i][j]=Infinity;
      }


for (int node=0;node<NumNodes;node++)
      {

            for(int p=0;p<Net[node].NumParents;p++)
            {

                  int this_parent =Net[node].Parents[p];
                        Cost[this_parent][node]=1;

            }
      }

}

////////////////////////////////////////////////
int min (int val1, int val2)
{
      int minimum;
      if(val1<=val2) minimum=val1;
      else           minimum=val2;
return minimum;
}
////////////////////////////////////////////////

void FindShortPaths()
{

for (int k=0;k<NumNodes;k++)
      for (int i=0;i<NumNodes;i++)
       for (int j=0;j<NumNodes;j++)
            Cost[i][j]=min(Cost[i][j],Cost[i][k]+Cost[k][j]);

}

////////////////////////////////////////////////
void FindChildren()
{

for(int i=0;i<MAXNODES;i++)
      for(int j=0;j<MAXNODES;j++)
      Children[i][j]=-1;


for( i=0;i<MAXNODES;i++)
Child_Count[i]=0;


for (int node=0;node<NumNodes;node++)
```

```
      {

            for(int p=0;p<Net[node].NumParents;p++)
            {

                  int this_parent =Net[node].Parents[p];
                  int location= Child_Count[this_parent];

                        Children[this_parent][location]=node;
                    Child_Count[this_parent]++;
            }
      }
}

////////////////////////////////////////////////

float Objective(GAGenome& g)
{
      float score=1.0;

GA1DArrayAlleleGenome<float> genome =(GA1DArrayAlleleGenome<float>&) g;

for (int node=0; node<NumNodes;node++)
      {
          int power=1;
            int index=0;
            int node_truth = (int) genome.gene(node);
            for(int p= Net[node].NumParents -1; p>=0 ; p--)
            {
            int parent_num= Net[node].Parents[p];
                  int parent_truth = int (genome.gene(parent_num));
                   index = index + parent_truth*power;
                   power = power*2;
            }//end for p

            if (node_truth== TRUE)
                  score = score*Net[node].Prob[index];
            else score = score* (1- Net[node].Prob[index]);
      }//end for node

      return score;
}
////////////////////////////////////////////////
void FlipMutator(GAGenome& g, float pmut)
{ //regular GA mutation

      GA1DArrayAlleleGenome<float>& child
=(GA1DArrayAlleleGenome<float>&)g;
      if (pmut>0.0)
      {
      for (int i=0; i<NumNodes; i++)
      {
            if (GAFlipCoin(pmut))

              child.gene(i, ((child.gene(i) == 0) ? 1 : 0));
```

```
      }//for

}//if
}//flipmutator

//////////////////////////////////////////////////

int MyMutation(GAGenome& g,float pmut)
{  // mutation diverted to SA if needed


      cout<<"\n\n"<<GenNum<<"\n\n";
      GA1DArrayAlleleGenome<float>& genome
      =(GA1DArrayAlleleGenome<float>&)g;
      GA1DArrayAlleleGenome<float> newgenome(allelset,Objective);
      //GA1DArrayAlleleGenome<float> bestgenome(allelset,Objective);
      float oldfitness;
      float newfitness;
      //float bestscore;

       if(GAFlipCoin(PROB_SA))
      {

      float T0 = INIT_TEMP;
      float factor  = SAFACTOR;
      float T= T0;
      int stagnation=0;
      while( (stagnation< MAXSTAGNATION))
            {
                  stagnation++;
                  newgenome.copy(genome);
                  FlipMutator(newgenome,Psam);
                  oldfitness= Objective(genome);
                  newfitness= Objective(newgenome);
                  if (newfitness<=oldfitness)
                  {

                  float paccept= exp (- (oldfitness-newfitness)/T);
                  paccept= paccept/(1.0+paccept);
                  if (GAFlipCoin(paccept))
                        {

                        genome.copy(newgenome);
                        if (newfitness!=oldfitness)
                              stagnation=0;

                        } //inner if
                              }// outer if
                        else {
                                    genome.copy(newgenome);
                                    stagnation=0;
                               }      //end else

       T= T*factor;
      } // end while

      } // end if
```

```
else // if SA not performed
 FlipMutator(genome,pmut);
return 1;
}


////////////////////////////////////////////////

int  MyCrossover(const  GAGenome& g1, const GAGenome& g2, GAGenome* c1,
GAGenome* c2)
{

      GA1DArrayAlleleGenome<float>& mom
=(GA1DArrayAlleleGenome<float>&)g1;
      GA1DArrayAlleleGenome<float>& dad
=(GA1DArrayAlleleGenome<float>&)g2;
      GA1DArrayAlleleGenome<float>& child1
=(GA1DArrayAlleleGenome<float>&)*c1;
      GA1DArrayAlleleGenome<float>& child2
=(GA1DArrayAlleleGenome<float>&)*c2;
//select a random root
int root=GARandomInt(0,NumNodes-1);
child1.copy(mom); //child1 is a copy of parent1
child2.copy(dad); //child2 is a copy of parent2
// now exchange the decendents of the random root betweeen
// the children


for (int i=0;i<NumNodes;i++)
{

      if (Cost[root][i]<Cluster_Limit)
  {
        child1.gene(i,dad.gene(i));
        child2.gene(i,mom.gene(i));
  }//end if

}//for

//if children have duplicates, force a change in them
float fit1=Objective(child1);
float fit2=Objective(child2);
for (int count=0;count<PopSize;count++)
{
      if (fit1==FitArray[count])
            FlipMutator(child1,ProbMutation*2);
      if (fit2==FitArray[count])
            FlipMutator(child2,ProbMutation*2);
}//for


      return 1;
}


////////////////////////////////////////////////
```

Source Code for Adaptive GASA

Declaration File “declaration.h”

```
#ifndef DECLARATION_H
#define DECLARATION_H
const int   MAXNODES = 70;
const int   MAXPARENTS= 15;
const int   MAXPROB  = 1<<MAXPARENTS;
#define     NETFILE "60_a_nod.net"
#define     STATISFILE "statis_60a.out"
#define     RESFILE "myres_60a.out"
#define     SCOREFILE "scorefile_60a.out"
#define     TRUE  1
#define     FALSE 0
#define     MINFACTOR   0.990 //min SA factor
#define     MAXFACTOR   0.999 //max SA factor
#define     MINSAPROB   0.005 //min SA probability
#define     MAXSAPROB   0.05  //max SA probability
#define     MAXSTAGNATION 500 // no. Of iterations needed to declare
stagnation
#define     PERCENT_CHANGE  0.2 //max change in SA parameters during
            mutation.
typedef int ParentArray[MAXPARENTS];
typedef float ProbArray[MAXPROB];
typedef struct NodeType
{
      int NumParents;
      ParentArray Parents;
      ProbArray   Prob;
};
typedef NodeType BeliefNetType[MAXNODES];
typedef int CostArray [MAXNODES][MAXNODES];
CostArray Cost;
int PopSize=1000;
int GenToConverge=500;
float ProbMutation = 0.02;
float ProbCrossover=0.99;
float PercentReplace=0.2;
float Cluster_Limit =1;
float ProbSACross=0.1; //probability of performing crossover of SA
parameters
float ProbSAMut= 0.05; // probability of performing mutation of SA
parameters.
float Psam=0.01; // probability of flipping a bit when creating a
random move for SA
float InitProb=0.01;// initial probability of acceptance used to
calculate T0 for SA
int     GenNum=0;
int   Infinity;
float FitArray[1000];
#endif
```

The main program for the GASA adaptive version “GASA_adpt.cpp”

```
#include <stdio.h>
#include "iostream.h"
#include <fstream.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#include"ga.h"
#include"GARealGe.h"
#include"GARealGe.cpp"

#include "declaration.h"

int NumNodes;

BeliefNetType Net;
void ReadNet();
void FindCost();
void FindShortPaths();
float Objective(GAGenome&);
void MyInitializer(GAGenome&);
int MyMutation(GAGenome& ,float );
int  MyCrossover(const  GAGenome& , const GAGenome& , GAGenome* ,
GAGenome* );

      GAAlleleSetArray<float> allelset;
      ofstream ScoreFile(SCOREFILE);

int main()
{
      unsigned int seed=5;
      GARandomSeed(seed);
      ReadNet();
      FindCost();
      FindShortPaths();
      float results [10000][4];

      for (int k=0; k<NumNodes; k++)
            allelset.add(FALSE,TRUE,1); //the BBN part has binary
            values  only
            allelset.add(MINFACTOR,MAXFACTOR);     // range of  cooling
      factor
            allelset.add(MINSAPROB,MAXSAPROB); // range of Prob of SA

            //define the genome as a 1d array of real with the values
            // of genes derived from the allelset array
            GA1DArrayAlleleGenome<float> genome(allelset,Objective);

            genome.initializer(MyInitializer);
            genome.mutator(MyMutation);
            genome.crossover(MyCrossover);
            GASteadyStateGA ga(genome);
```

```
ga.populationSize(PopSize) ;
ga.nConvergence(GenToConverge);
ga.terminator(GAGeneticAlgorithm::TerminateUponConvergence)
;
ga.pMutation (ProbMutation);
ga.pCrossover(ProbCrossover);
ga.pReplacement(PercentReplace);
ScoreFile<<"generation \t mean \t max \t min\n";
ga.scoreFilename(SCOREFILE);
ga.scoreFrequency(1); //keep the scores of every generation
ga.flushFrequency(1); //flush scores every generation
ga.selectScores(GAStatistics::Mean|GAStatistics::Minimum|GA
Statistics::Maximum);

//get current CPU time
clock_t CPU_time;
CPU_time=clock();
ga.initialize();

while (!ga.done())
{

      float sum1=0;
      float sum2=0;

      GenNum= ga.statistics().generation();
      GAPopulation current_pop = ga.population();
      GA1DArrayAlleleGenome<float>
      tempgenome(allelset,Objective);
      // the following loop is used to calculate the
      average of the genes representing SA parameters.
      for (int indiv=0; indiv<PopSize; indiv++ )
      {
        tempgenome= current_pop.individual(indiv);
        FitArray[indiv]=Objective(tempgenome);
        sum1+= tempgenome.gene(NumNodes);
        sum2+= tempgenome.gene(NumNodes+1);
      }

      results[GenNum][0]= sum1/PopSize;
      results[GenNum][1]= sum2/PopSize;
      results[GenNum][2]= current_pop.ave();
      results[GenNum][3]= current_pop.max();


  ga.step();

}
ga.flushScores();
ScoreFile.close();

CPU_time=clock() - CPU_time;
cout<<" \n\n THE GA FOUND : "
<<ga.statistics().bestIndividual() << "\n";
cout<<ga.statistics().bestIndividual().score() << "\n\n";
cout<<"Processing Time=  "<<(CPU_time/1000)/60<<"  mins\n";
```

```
	//write results to file
	ofstream ResultFile;
	ResultFile.open(RESFILE,ios::app);//append the results
	ResultFile<<"MINFACTOR = "<<MINFACTOR<<"\t";
	ResultFile<<"MAXFACTOR = "<<MAXFACTOR<<"\t";
	ResultFile<<"MINSAPROB = "<<MINSAPROB<<"\t";
	ResultFile<<"MAXSAPROB = "<<MAXSAPROB<<"\t";
	ResultFile<<"MAXSTAGNATION = "<<MAXSTAGNATION<<"\t";
	ResultFile<<"PopSize   = "<<PopSize<<"\t";
	ResultFile<<"NumGenerations= "<<GenNum<<"\n";
	ResultFile<<"GenToConverge= "<<GenToConverge<<"\t";
	ResultFile<<"ProbMutation= "<<ProbMutation<<"\t";
	ResultFile<<"ProbCrossover= "<<ProbCrossover<<"\n";
	ResultFile<<"ProbSAMut= "<<ProbSAMut<<"\t";
	ResultFile<<"Psam = "<<Psam<<"\t";
	ResultFile<<"InitProb = "<<InitProb<<"\t";
	ResultFile<<"ProbSACross= "<<ProbSACross<<"\t";
	ResultFile<<"ClusterLimit= "<<Cluster_Limit<<"\n";
	ResultFile<<"Bestindividual =
	"<<ga.statistics().bestIndividual()<<"\n";
	ResultFile<<"Processing Time=  "<<(CPU_time/1000)/60<<"
	mins\n";
	ResultFile<<"BestScore  =
	"<<ga.statistics().bestIndividual().score()<<"\n";
	ResultFile.close();


	ofstream StatisFile;
	StatisFile.open(STATISFILE,ios::app);//append the results
	StatisFile<<"Statistics for File : "<<NETFILE<<"\n\n";
	StatisFile.width(3);
	StatisFile<<"Gen"<<"\t";
	StatisFile.width(15);
	StatisFile<<"avg_SAf"<<"\t";
	StatisFile.width(15);
	StatisFile<<"avg_probSA"<<"\t";
	StatisFile.width(15);
	StatisFile<<"fit_avg"<<"\t";
	StatisFile.width(15);
	StatisFile<<"fit_max"<<"\n\n";

	for (int gen=0; gen<GenNum; gen++)
	{
		StatisFile.width(3);
		StatisFile<<gen<<"\t";

		for (int r=0;r<4;r++)
		{
		StatisFile.width(15);
		StatisFile<<results[gen][r]<<"\t";
		}
	StatisFile<<"\n";
	}
	StatisFile.flush();
	StatisFile.close();

return 0;
```

```
}
////////////////////////////////////////////////
void MyInitializer(GAGenome& g)
{

      GA1DArrayAlleleGenome<float>& genome
=(GA1DArrayAlleleGenome<float>&)g;

      for (int i=0;i<NumNodes;i++)
            genome.gene(i, float(GARandomInt(0,1)));
      genome.gene(NumNodes,GARandomFloat(MINFACTOR,MAXFACTOR));
      genome.gene(NumNodes+1,GARandomFloat(MINSAPROB,MAXSAPROB));

}

////////////////////////////////////////////////////
void ReadNet()
{
ifstream InFile(NETFILE);
if (!InFile)
    {
       cout<<"Could Not Read Input File"<<NETFILE<<"\n";
       exit(1);
    }
InFile>>NumNodes; //read number of nodes
for (int node=0; node<NumNodes; node++)
{
      InFile>>Net[node].NumParents; // read number of parents for this
node
      for (int p=0; p<Net[node].NumParents;p++) // read parent numbers
for this node
      {
            InFile>>Net[node].Parents[p];
            Net[node].Parents[p]--; // decrement parent number to start
from 0 instead of 1
      }//end for p
      int   probs = (1<< Net[node].NumParents) ; // calculate number of
entries in prob table
  for (int pr=0; pr<probs; pr++)  //read probabilities
         InFile>> Net[node].Prob[pr];

} //end for node
InFile.close();

}// end ReadNet



////////////////////////////////////////////////

void FindCost()
{

Infinity=NumNodes;
//initialize Cost Array
for(int i=0;i<MAXNODES;i++)
      for(int j=0;j<MAXNODES;j++)
```

```
      {
            if(i==j) Cost[i][j]=0;
            else Cost[i][j]=Infinity;
      }


for (int node=0;node<NumNodes;node++)
      {

            for(int p=0;p<Net[node].NumParents;p++)
            {

                  int this_parent =Net[node].Parents[p];
                        Cost[this_parent][node]=1;

            }
      }
}

////////////////////////////////////////////////
int min (int val1, int val2)
{
      int minimum;
      if(val1<=val2) minimum=val1;
      else           minimum=val2;
return minimum;
}
////////////////////////////////////////////////

void FindShortPaths()
{

for (int k=0;k<NumNodes;k++)
      for (int i=0;i<NumNodes;i++)
       for (int j=0;j<NumNodes;j++)
            Cost[i][j]=min(Cost[i][j],Cost[i][k]+Cost[k][j]);

}



float Objective(GAGenome& g)
{
      float score=1.0;

GA1DArrayAlleleGenome<float> genome =(GA1DArrayAlleleGenome<float>&) g;

for (int node=0; node<NumNodes;node++)
      {
          int power=1;
            int index=0;
            int node_truth = (int) genome.gene(node);
            for(int p= Net[node].NumParents -1; p>=0 ; p--)
            {
            int parent_num= Net[node].Parents[p];
                  int parent_truth = int (genome.gene(parent_num));
                   index = index + parent_truth*power;
```

```
                power = power*2;
            }//end for p

            if (node_truth== TRUE)
                  score = score*Net[node].Prob[index];
            else score = score* (1- Net[node].Prob[index]);
      }//end for node

      return score;
}



 void mutate_regular(GAGenome& g,int mutSAparams, float pmut)
{ //regular GA mutation
      GA1DArrayAlleleGenome<float>& child
=(GA1DArrayAlleleGenome<float>&)g;
int nMut=0;
if (pmut>0)
{
      for (int i=0; i<NumNodes; i++)
      {
            if (GAFlipCoin(pmut))
            {
              nMut++;
              child.gene(i, ((child.gene(i) == 0) ? 1 : 0));
            }

      }// for

}// if


      float randval,geneval,new_geneval,minval,maxval;

      if ((mutSAparams==TRUE) ) //if mutation of SA parameters is
      required
      {
            for (int location=NumNodes;location<=NumNodes+1; location++
            )
            {
            if (GAFlipCoin(ProbSAMut))
            {
                  if (location==NumNodes)
                  {
                  minval=MINFACTOR;
                  maxval=MAXFACTOR;

                  }
                  else
                  {
                  minval=MINSAPROB;
                  maxval=MAXSAPROB;
                  }

                   randval= GARandomFloat(-1.0,1.0);
```

```
                  geneval= child.gene(location);
                 new_geneval=
                 float(geneval+randval*PERCENT_CHANGE*(maxval-
                 minval)*geneval);
                  if(new_geneval>maxval) new_geneval=maxval;
                  else if (new_geneval< minval) new_geneval=minval;
                 child.gene(location,new_geneval);

            }// if GAflipcoin(ProbSAMut)

            }// for

      } // if mutSAparams
}

////////////////////////////////////////////////
int MyMutation(GAGenome& g,float pmut)
//mutation diverted to SA if needed
{
      cout<<"\n\n"<<GenNum<<"\n\n";
      GA1DArrayAlleleGenome<float>& genome
      =(GA1DArrayAlleleGenome<float>&)g;
      GA1DArrayAlleleGenome<float> newgenome(allelset,Objective);
      GA1DArrayAlleleGenome<float> bestever(allelset,Objective);

      float pSA= genome.gene(NumNodes+1); //take PSA from the specified
      gene in the current genome
      float oldfitness;
      float newfitness;

      if(GAFlipCoin(pSA))
      {

            float T0=-1*Objective(genome)/(log(InitProb));
            float factor= genome.gene(NumNodes);
            float T= T0;
            int stagnation=0;
            while( stagnation< MAXSTAGNATION)
                  {
                        stagnation++;
                        newgenome.copy(genome);
                        mutate_regular(newgenome,FALSE,Psam);

                        oldfitness= Objective(genome);
                        newfitness= Objective(newgenome);
                        if (newfitness<= oldfitness)
                        {
                              float paccept=exp (- (oldfitness
                              newfitness)/T);
                              paccept= paccept/(1.0+paccept);
                              if (GAFlipCoin(paccept))
                              {

                               genome.copy(newgenome);
                              if (newfitness != oldfitness)
                                    stagnation=0;
                              }//inner if (flipcoin)
```

```
                              }//  if (newfitness < oldfiyness)
                        else { // if newfitness>oldfitness accept new
                  solution
                                    genome.copy(newgenome);
                                    stagnation=0;
                               }      //end else

       T= T*factor;  //reduce temperature
      } // end while

      } // end if GAFlipCoin(pSA)
else // if SA not performed
 mutate_regular(genome,TRUE,pmut);

return 1;
}

////////////////////////////////////////////////////
////////////////////////////////////////////////

int  MyCrossover(const  GAGenome& g1, const GAGenome& g2, GAGenome* c1,
GAGenome* c2)
{

      GA1DArrayAlleleGenome<float>& mom
=(GA1DArrayAlleleGenome<float>&)g1;
      GA1DArrayAlleleGenome<float>& dad
=(GA1DArrayAlleleGenome<float>&)g2;
      GA1DArrayAlleleGenome<float>& child1
=(GA1DArrayAlleleGenome<float>&)*c1;
      GA1DArrayAlleleGenome<float>& child2
=(GA1DArrayAlleleGenome<float>&)*c2;
//select a random root
int root=GARandomInt(0,NumNodes-1);
child1.copy(mom); //child1 is a copy of parent1
child2.copy(dad); //child2 is a copy of parent2
// now exchange the decendents of the random root betweeen
// the children

for (int i=0;i<NumNodes;i++)
{

      if (Cost[root][i]<Cluster_Limit)
  {
        child1.gene(i,dad.gene(i));
        child2.gene(i,mom.gene(i));
  }//end if

}//end for
// if children have duplicates, force a change in them
float fit1=Objective(child1);
float fit2=Objective(child2);
for (int count=0;count<PopSize;count++)
{
      if (fit1==FitArray[count])
            mutate_regular(child1,TRUE,ProbMutation*2);
```

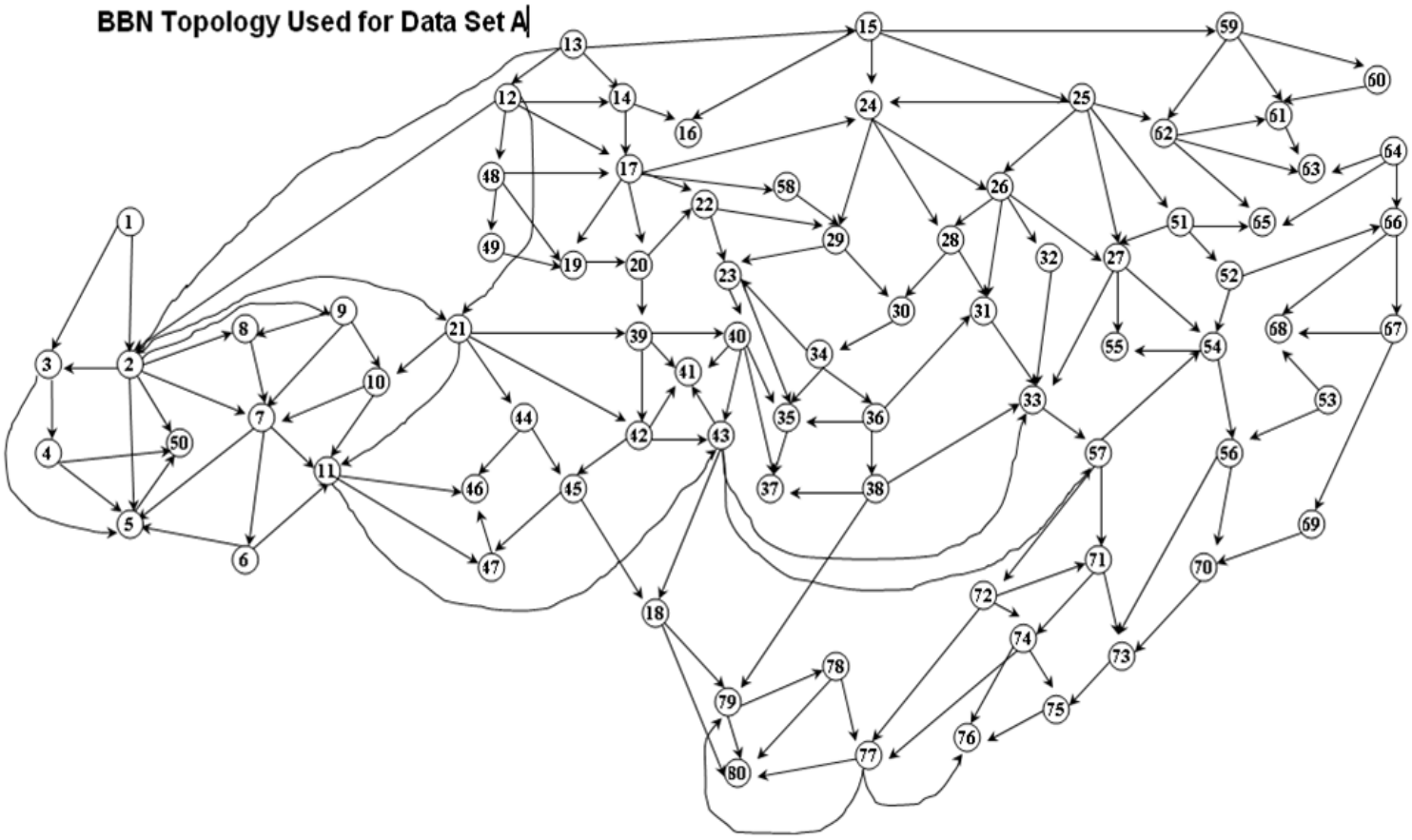
```
      if (fit2==FitArray[count])
            mutate_regular(child2,TRUE,ProbMutation*2);
}//for

// now perform crossover for the SA parameters
if (GAFlipCoin(ProbSACross))
{
      int cutpoint=GARandomInt(NumNodes-1,NumNodes);
      for (int i=cutpoint+1;i<= NumNodes+1;i++)
      {
        child1.gene(i,dad.gene(i));
        child2.gene(i,mom.gene(i));

      }
}

      return 1;
}
```

Appendix B: BBN Topologies



BBN Topology Used for Data Set B