A COARSE-GRAIN PARALLEL GENETIC ALGORI THM TO IMPROVE THE BOUNDS OF SOME RAMSEY NUMBERS

By

IKER GONDRA

Bachelor of Science

Oklahoma State University

Stillwater, Oklahoma

1998

Submitted to the Faculty of the
Graduate College of
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 2002

A COARSE-GRAIN PARALLEL GENETIC ALGORI THM TO

IMPROVE THE BOUNDS OF SOME RAMSEY NUMBERS

Thesis Approved:

Mansur H. Samadzadeh

Thesis Advisor

J Chandler

Blayne E. Mayfield

Timothy J. Pettibone

Dean of the Graduate College

PREFACE

Ramsey Theory studies the existence of highly regular patterns within a large object or set of randomly selected points or numbers. The role of Ramsey numbers is to quantify some of the general existential theorems in Ramsey theory. Attempting to find Ramsey numbers has been an arduous task that is too often unfruitful. Only a handful of specific numbers are known.

Genetic Algorithms (GA), which are based on the idea of optimizing by simulating the natural processes of evolution, have proven successful in solving complex problems that are not easily solved through conventional methods. However, premature convergence is an inherent characteristic of traditional GA's that makes them incapable of searching numerous points in a problem domain. Parallel GA (PGA) is an extension of the classical GA that takes advantage of a GA's inherent parallelism to improve its time performance and reduce the likelihood of premature convergence. A cgGA (Coarse-Grain GA) maintains a number of independent populations and allows for the occasional interchange of individuals. In this manner, a cgGA increases the diversity of search paths and helps to stop premature convergence to non-optimal solutions.

The objective of this thesis was to develop a simulated Coarse-Grain GA to verify and validate the superior performance of cgGA's over traditional GA's applied to the problem of improving the bounds of classical Ramsey Numbers. Threads were used to simulate the parallel evolution of multiple subpopulations. The tool developed is a JAVA applet called SIPAGAR (SImulated PArallel Genetic Algorithm for finding

Ramsey numbers). Significant differences between the simulated cgGA and the traditional GA were observed in both the premature convergence rate and the quality of the results. This leads us to the conclusion that future cgGA-based attempts to improve the bounds of Ramsey Numbers will probably be more promising that those based on traditional GA's.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my graduate advisor, Dr. Mansur H. Samadzadeh, the best advisor I could have wished for, for his supervision and guidance through the duration of this research, and for his keen interest in my progress. I thank him for introducing me to the topic of Ramsey Numbers and for his invaluable advice along the way.

I would also like to thank Drs. John P. Chandler and Blayne E. Mayfield for serving on my thesis committee and for their valuable input.

I am forever indebted to my parents, family, and friends for their endless encouragement and support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF TABLES

CHAPTER I

INTRODUCTION

According to Ramsey Theory, a sufficiently large system (no matter how random) will always contain higly organized subsystems. The role of Ramsey numbers is to quantify some of these existential theorems. Finding Ramsey numbers has proven to be a very difficult task that has led researchers to experiment with different methods of accomplishing this task.

Genetic Algorithms (GA's), originated by John Holland [Holland 75] in 1975, have been successfully applied to complex problems in a large number of different disciplines. Most research has been devoted to the original computational model developed by John Holland [Holland 75], which will be referred to in this thesis as the Standard Genetic Algorithm (SGA). Because of its implicit parallelism (discussed in Section 3.4) and the few assumptions it makes about the problem being solved, an SGA is able to find solutions to complex problems that are not easily solved through conventional methods. An SGA maintains a set of possible solutions to a specific problem. It imitates the natural processes of selection and recombination to evolve better solutions.

In order to decide on the "goodness" of each solution in solving a problem, a numeric fitness value is computed. The effectiveness of an SGA in finding an optimal solution is largely determined by the size of the solution set [Goldberg et al. 92]. A small solution set usually results in premature convergence to a suboptimal solution –

a common problem of any SGA. On the other hand, the computational cost also increases as a function of the size of the solution set. A parallel GA (PGA) takes advantage of the highly parallelizable nature of SGAs in order to overcome these problems. In particular, a Coarse-Grain GA (cgGA) maintains several solution subsets, each of them "evolving" independently on a separate processor, and occasionally interchanging solutions. This thesis implements a simulation of a cgGA to compare the performance of an SGA with that of a cgGA in attacking a GA-hard problem that is one of the most interesting combinatorial problems – Finding Ramsey Numbers.

The rest of this thesis is organized as follows. The next chapter presents a brief introduction to Ramsey Theory. Chapter III discusses GA's along with their problems and theoretical foundations. A survey of the different types of PGA's with emphasis on cgGA's is presented in Chapter IV. Next, Chapter V describes the overall design and implementation of the simulated cgGA "SIPAGAR". A description of the experiments that were conducted to compare the performance of an SGA with that of a cgGA is included in Chapter VI. Chapter VII presents the results achieved, makes concluding remarks based on the results, and identifies some directions for future work.

CHAPTER II

RAMSEY THEORY

"Complete disorder is impossible."
T. S. Motzkin

Frank Plumpton Ramsey, an English mathematician and economist, proved that complete disorder is an impossibility in his paper "On a Problem of Formal Logic" (1930).

Ramsey theory studies the existence of highly regular patterns within a large object or set of randomly selected points or numbers. The role of Ramsey numbers is to quantify some of the general existential theorems in Ramsey theory.

The party puzzle is a classical problem used to introduce the theory. What is the minimum number of guests that must be invited to a party so that either a group of at least three people will know one another or at least three guests will not mutually know each other? The answer to this problem, which equals 6, is called the Ramsey number R(3,3).

Stated in a mathematical way, given 6 points or vertices, we draw a line segment between every pair of vertices to obtain a complete graph of order 6 (denoted by K_6). If the symmetric relationship of knowing/not knowing between 2 points in the graph is represented by the color of the edge connecting the two vertices, then the claim is that every one of the possible 32,768 colorings will yield a monochromatic K_3 (i.e., a complete graph of order 3 in which every edge has the same color). A proof of this appears in Appendix E. The special notation $K_6 \rightarrow K_3$ is used to record this result. In general, $K_n \rightarrow K_m$ states that every 2-coloring of the edges of K_n yields a monochromatic K_m.

Generalizing these observations, suppose that a and b are integers with a, $b \geq 2$, then a possible integer N has the (a,b) Ramsey property if the following holds: Given any set S of N elements, if we divide the 2-element subsets of S into two classes A and B, then either

1. there is an a-element subset of S all of whose 2-element subsets are in A, or
2. there is a b-element subset of S all of whose 2-element subsets are in B.

The smallest integer N that has the (a,b) Ramsey property is called a Ramsey number and is denoted by $R(a,b)$ [Erickson 96]. Thus, number 6 has the (3,3) Ramsey property and $R(3,3) = 6$.

As the following theorem shows, Ramsey's theory is generalized to graphs with an arbitrary number of edge colors.

> For any integer $c \geq 2$, and integers $A_1, A_2, ..., A_c \geq 2$, there exists a least integer $R(A_1, A_2, ..., A_c)$ with the following property: If the edges of the complete graph on $R(A_1, A_2, ..., A_c)$ vertices are partitioned into color classes $A_1, A_2, ..., A_c$, then for some i there exists a complete graph on A_i vertices all of whose edges are color A_i [Erickson 96].

The only known value for a multicolor classical Ramsey number is $R(3,3,3) = 17$. The interpretation of this is the following: Every coloring of the edges of a complete graph with 17 vertices in 3 colors will give rise to a triangle that is monochromatic in one of the 3 colors. Ramsey's theorem is also extended to hypergraphs.

> Let integer $c \geq 2$ and integers $A_1, A_2, ..., A_c \geq t \geq 2$. There exists a least integer $R(A_1, ..., A_c;t)$ with the following property: Every c-coloring of the complete t-uniform hypergraph $[R(A_1, A_2, ..., A_c;t)]^t$ with colors $A_1, A_2, ..., A_c$ yields a complete t-uniform hypergraph on A_i vertices in color A_i, for some i [Erickson 96].

If in the notation $R(G_1, G_2, ..., G_m;s)$ s is not specified, a 2-uniform hypergraph (i.e., a conventional graph) is assumed. Thus $R(3,3) = R(3,3;2)$ and $R(3,3,3) =$

R(3,3,3;2).

In order to find a Ramsey number, say $R(G_1, G_2, ..., G_k)$, we need to find the largest number N such that a k-colored complete graph K_N does not contain a monochromatic subgraph G_i in color i for $1 \leq i \leq k$. Once such an N is found, then $(N + 1)$ is $R(G_1, G_2, ..., G_k)$. For example, to deduce that R(3,3) = 6, we would have to show that 5 is the largest N such that a complete graph on N vertices does not necessarily contain a monochromatic triangle of either of 2 colors.

Unfortunately, attempting to find Ramsey numbers is an arduous task that is too often unfruitful. Only a handful of specific numbers are known (a table of known Ramsey numbers is included in Appendix C). Erdo's anecdote captures the difficulty of finding even the comparatively simple diagonal Ramsey numbers (i.e., $R(a,a)$),

> Aliens invade the earth and threaten to obliterate it in a year's time unless human beings can find the Ramsey number for red five and blue five. We could marshal the world's best minds and fastest computers, and within a year we could probably calculate the value. If the aliens demanded the Ramsey number for red six and blue six, however, we would have no choice but to launch a preemptive attack [Graham and Spencer 90].

This state of limited knowledge is exasperating because Ramsey numbers are intimately connected with other numbers and functions such as the Stirling numbers. It is well known that any new Ramsey number would be very valuable [Erickson 96]. If complete disorder is an impossibility, what order is there in apparent disorder? This research effort investigated the performance of some methods to improve the bounds of Ramsey numbers which attempt to quantify this "order".

CHAPTER III

GENETIC ALGORITHMS

3.1 Introduction to Standard Genetic Algorithms

Genetic algorithms (GA's) are adaptive methods that can be used to solve search and optimization problems. They are based on the mechanics of natural selection and genetic processes of living organisms. From one generation to another, populations evolve according to the principles of natural selection and the survival of the fittest individuals [Darwin 59]. By imitation of the natural process, GA's are capable of developing solutions to real problems.

The basic principles of GA's were established by John Holland in 1975 [Holland 75]. Holland's insight was to be able to represent the fundamental biological mechanisms that permit system adaptation into an abstract form that could be simulated on a computer for a wide range of problems. He introduced bit strings to represent feasible solutions (or individuals) in some problem space. GA's are analogous to the natural behavior of living organisms. Individuals in a population compete for resources. Those individuals that are better adapted survive and have a higher probability of mating and generating descendants. Therefore, the genes of stronger individuals will increase in successive generations.

A GA works with a population of individuals, each representing a feasible solution to a given problem. During each iteration step, called a generation, the individuals in the current population are evaluated and given a fitness value, which is proportional to the "goodness" of the solution in solving the problem. Individuals are represented with strings of parameters or genes known as chromosomes.

The phenotype, the chromosome, contains the information that is required to construct an individual (a solution to the problem). The phenotype is used by the fitness function to determine the genotype, which denotes the level of adaptation of the chromosome to the particular problem. To form a new population, individuals are selected with a probability proportional to their relative fitness. This ensures that well adapted individuals (good solutions) have more chances of being reproduced. Once two parents have been selected, their chromosomes are combined and the traditional operators of crossover and mutation [Holland 75] are applied to generate new individuals (i.e., new search points). In its simplest form, crossover consists of selecting random points in a string and swapping the substrings of the parents (Figure 1).

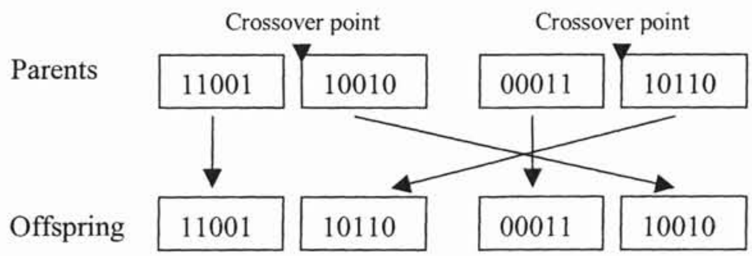


Figure 1. Simple crossover operator

The mutation operator is applied by changing at random the value of a bit in a string with a certain probability called the *mutation rate*. This operator is used to prevent premature convergence to local optima by introducing new genetic material (new points in the search space). Algorithm A below shows a standard or simple GA (SGA).

```
BEGIN SGA
  Randomly Create an initial population
  WHILE NOT termination criteria DO
    BEGIN
      Assign a fitness value to each individual
      Select individuals for reproduction
      Produce new individuals
      Mutate some individual(s)
      Generate new population by replacing bad
       individuals with some new good individuals
    END
END SGA
```

Algorithm A. Standard genetic algorithm (SGA) [Darwin 59]

In Algorithm A, the termination criteria may be triggered when either an acceptable solution has been found or when a problem-specific maximum number of generations has been reached.

GAs have been successful in solving complex problems that are not easily solved through conventional methods [Stracuzzi 98] for several reasons. They start with a population of points rather than a single point. Therefore, many portions of the domain are searched simultaneously and, as a result, they are less prone to settling at local optima during the search. GA's work with an encoding of the parameter set, not the parameters themselves. Because they do not depend on domain knowledge in performing the search, inconsistent or noisy domain data are less likely to affect them as is common with hill-climbing or domain specific heuristics [Stracuzzi 98].

The simulated parallel GA developed as part of this thesis is based on the islands model [Cohoon et al. 91]. The basic idea behind this model consists in dividing the population into several subpopulations (or islands). In each one of those islands, an SGA is run.

The next section outlines the representation and approach for the algorithm that runs in each one of the subpopulations. Parallel GA's and the islands model are further discussed in Chapter IV.

3.2 Approach and Representation

Given the problem of finding a Ramsey number, say $R(G_1, G_2, \ldots, G_k)$, we need to find the largest number N such that a k-colored complete graph K_N does not contain a monochromatic subgraph G_i in color i for $1 \leq i \leq k$. Once such an N is found, then (N + 1) is $R(G_1, G_2, \ldots, G_k)$. For example, it is known that $43 \leq R(5,5) \leq 49$. Therefore, to improve the lower bound of R(5,5), the first step would be to find a 2-colored graph of order 43 with no monochromatic subgraph of order 5. Then we could conclude that $44 \leq R(5,5) \leq 49$. We would then repeat the same process, each time increasing the lower bound by one, until the largest possible N can be found.

The first step in developing a GA that will solve a given problem is to define the following two mechanisms:

1) A way of encoding solutions to the problem in terms of chromosomes.
2) An evaluation function that returns a measurement of the fitness of a chromosome in solving the given problem.

These two steps are discussed in the following two subsections. The third subsection explains the need to use permutation-respecting crossover operators when using an order-based solution encoding.

3.2.1 Solution Encoding

A solution to the problem will be a complete graph of order N with a number X of monochromatic subgraphs of order K. In the optimal solution, X=0. There are several ways of representing a graph as a chromosome. An entry (i,j) in an NxN adjacency matrix can store the color of the edge (i,j). The lower or upper triangle of the adjacency matrix can then be mapped into a single dimensional array (a chromosome). A better approach is to use an order-based representation in which each chromosome is a permutation of edges, a decoder is then used to color the edges of a permutation [Eiben and van der Hauw 98]. The results of numerous experiments [Eiben and van der Hauw 98] conducted on a graph coloring problem have showed that other representations are inferior to the order-based representation.

3.2.2 Evaluation Function

As the decoder encounters the edges in the order that they occur in a certain chromosome, it assigns the smallest possible color from the set of k colors. If each of the k colors leads to a constraint violation (i.e., the formation of a monochromatic subgraph), the edge is left uncolored [Eiben and van der Hauw 98]. The fitness of a chromosome is then equal to the sum of the uncolored edges. Thus a chromosome with a fitness value of 5 is more fit than one with a fitness value of 10. The evaluation function to be minimized is defined as:

$$\mathrm{f}(x) = \sum_{i=1}^{n} W_i * \chi(x,i)$$

where n is the number of edges in the chromosome x, W_i is the local penalty (or weight) assigned to edge x_i, and

$$\chi(x,i) = \begin{cases} 1 \text{ if edge } x_i \text{ is left uncolored} \\ 0 \text{ otherwise} \end{cases}$$

If we simply count the uncolored edges, then $W_i \equiv 1$. However, not every edge is equally hard to color [Eiben and van der Hauw 98]. For example, coloring the first edge that appears in a chromosome is an easy task, the decoder may choose any of the k possible colors. On the other hand, coloring the edges at the end of the chromosome may be very difficult as the number of colors that do not result in a constraint violation may be heavily reduced. A better approach would then be to give "hard" edges (i.e., the edges that are colored last) a high weight, since this gives the evaluation function a high reward when satisfying them, thus concentrating on these edges [Eiben and var den Hauw 98].

In this thesis, we use a modified version of the evaluation function in which all edges are colored. The evaluation function to be maximized is defined as:

$$\mathrm{f}(x) = n - \sum_{i=1}^{n} W_i * \chi(x,i)$$

where n is the number of edges in the chromosome x, W_i is the local penalty (or weight) assigned to edge x_i, and

$$\chi(x,i) = \begin{cases} 1 \text{ if all } k \text{ colorings of edge } x_i \text{ create subgraph(s)} \\ 0 \text{ otherwise} \end{cases}$$

The local penalty W_i is equal to the number of monochromatic subgraphs that are created after coloring the edge with the color that minimizes the resulting number of monochromatic subgraphs. Because edges near the end of the chromosome result in more monochromatic subgraphs, this function gives a higher weight to those edges.

3.2.3 Crossover

Ordinary crossover and mutation operators cause problems for order-based representations. The reason for this is that offspring generated by means of ordinary operators may not be valid solutions for the problem being solved anymore. For example, suppose we have a complete graph of order 4 as shown in Figure 2.

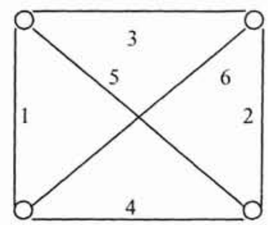


Figure 2. A complete graph of order 4

Also, suppose two chromosomes are selected for crossover (Figure 3).

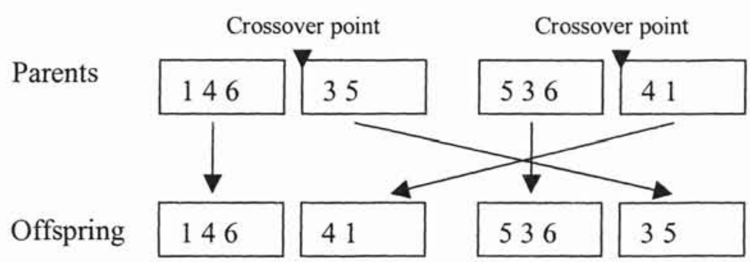


Figure 3. Creation of invalid permutations with traditional crossover operator

As can be observed, the offspring are not valid permutations anymore (i.e., for the first offspring, and analogously for the second offspring, the decoder would try to color edges 4 and 1 twice and never try to color edges 3 and 5). The way in which the ordinary mutation operator can produce invalid chromosomes is obvious. Several solutions have been suggested to deal with this problem [Poon and Carter 95]. An invalid chromosome could simply be disqualified, it could also be repaired. The approach that is followed in this thesis consists of using specialized permutation-respecting operators instead of creating invalid chromosomes [Ugoluk 97]. A list of the permutation-respecting operators is included in Appendix D.

3.3 Premature Convergence

Premature convergence is a common problem of any SGA. It occurs when the individuals in the population are selected proportionally according to their relative fitness. Some individuals may have a very high fitness value and, as the algorithm continues executing, they may dominate the entire population. Once a suboptimal solution dominates the population, selection will keep it there and prevent any further adaptation to the problem. When crossover occurs, no new patterns will be created, causing the search to stop. Previous research has focused on two general approaches to address this problem [Goodman et al. 94]. The first approach affects the selection phase and focuses on lowering the convergence speed so the algorithm can do a more thorough search before converging. The second approach attempts to keep a high population diversity by modifying traditional replacement and mating operators [Goodman et al. 94]. Some proposed methods for avoiding premature convergence are discussed next.

Goldberg and Richardson [Goldberg and Richardson 87] proposed a method to increase population diversity by modifying the fitness value of every individual. The basic idea is to lower the fitness value of individuals that are similar to one another and to increase the fitness value of solutions that are isolated or different from the rest of the population. In this manner, individuals that are close to one another (similar) will reduce their chances of being selected for crossover, thus increasing the probability of selecting isolated individuals.

For example, if $d(I_j, I_i)$ denotes the Hamming distance between individuals I_j and I_i, and k is a positive real parameter, we can define the following function h:

$$\mathrm{h}(\mathrm{d}(I_j, I_i)) = \begin{cases} k - \mathrm{d}(I_j, I_i) & \text{if } \mathrm{d}(I_j, I_i) < k \\ 0 & \text{if } \mathrm{d}(I_j, I_i) \geq k \end{cases}$$

Now, for each individual I_j, σ^j is defined as the summation of $\mathrm{h}(\mathrm{d}(I_j, I_i))$ for all individuals I_i where $i \neq j$ [Goldberg and Richardson 87]. The value of σ^j is then used to modify the fitness function of each individual I_j. If $\mathrm{g}(I_j)$ gives the fitness value of solution I_j, the new value would be $\mathrm{g}(I_j) / \sigma^j$ [Goldberg and Richardson 87]. In other words, we determine how similar each individual is to all the other solutions in the population and modify its fitness value accordingly.

Another possible improvement over the traditional method of proportional selection is to set a limit on the number of times that an individual can be selected for reproduction. For each individual i, we could use a counter initialized to $\mathrm{fv}_i / \mathrm{fv}_{average}$ where fv_i is the fitness value of solution i and $\mathrm{fv}_{average}$ is the average fitness value of the entire population. In this manner, we allow a good individual to be chosen more number of times but only up to a certain limit (i.e., until the value of the counter reaches 0).

Another commonly used method for dealing with premature convergence is tournament selection. It consists of randomly choosing k individuals out of the entire population to form a tournament. The best individual in the tournament is then selected for reproduction. In this way, the selection of individuals which are not necessarily the best solutions in the population is permitted.

De Jong introduced the concept of a crowding scheme [De Jong 75]. The approach consists of randomly choosing a subpopulation of CF (crowding factor) individuals.

Hamming's distance is used to determine a value for each individual according to its similarity with other individuals in the subpopulation. An offspring then replaces one of the individuals with a high "similarity value". Therefore, similar solutions in a subpopulation will compete with one another and the speed at which convergence occurs is reduced [De Jong 75]. Another approach for maintaining diversity is to allow the insertion of an offspring into the population only if it is different enough from all other individuals [Mouldin 84].

Even though much research has been devoted to avoiding premature convergence, this problem is still an inherent characteristic of traditional GA's. Therefore, these algorithms are incapable of maintaining different high-fitness individuals within a single population, thus they are not able to search numerous points of the problem domain. Chapter IV presents a GA based on a more realistic model of nature that avoids premature convergence in a much more efficient manner and holds other advantages as well.

3.4 Why They Work – The Schemata Theorem

In his book, *Adaptation in Natural and Artificial Systems*, John Holland presents the theoretical foundations explaining the robustness of GA's as a search technique [Holland 75]. The key to finding an optimal solution for a given problem is to be able to identify and exploit useful properties in a large search space S. Each chromosome (or solution) $C_i \in S$ is represented by a set of genes (attributes or bits) G_i. For example, if two colors (0 and 1) are used to draw the 15 edges of a K_6, the chromosome 011100101011101 represents a coloring (solution). In this particular problem, S is all the possible colorings (solutions) of a K_6 with two colors. The size of S is 32,768. If "*" is used as a "don't care" symbol, then this chromosome can also be represented by the string 011**********01. Strings containing one or more "don't care" symbols are referred to as schemata [Holland 75]. A string corresponds to a particular schemata if we can obtain the string by substituying the "don't care" symbols with the corresponding bit value. For example, the string 100110 corresponds to the schemata 10***0 but not to 00***0.

Holland makes the important observation [Holland 75] that every string (chromosome or solution) corresponds to $2^m - 1$ different schemata, where m is the length of the string. To show this, observe that there are m positions in a string of length m and each position can contain either a bit value or the "don't care" symbol "*". A one is subtracted because the string of all "*" symbols represents the search space S itself, not an schemata (or partition of S) [Holland 75]. As a result, each time a string (chromosome or solution) is evaluated, many ($2^m - 1$) different schemata (or partitions of S) are sampled. Consequently, every time a population is explicitly evaluated, a number of schemata much greater than the population size is implicitly sampled. This is what is meant when referring to a GA's implicit parallelism.

John Holland, at the end of Chapter Four on Schemata in his 1975 book [Holland 75] summarizes this important observation as follows:

> …the elements $A \in \mathsf{a}$ each have a representation $(\delta_1(A), \ldots, \delta_l(A))$ in terms of the ordered set of l attributes $\delta_i(A) \in V_i$, $i = 1, \ldots, l$. Each $\xi \in \Xi = \Pi_{i=1}^{l} \{V_i \cup \{\square\}\}$ [,where a is a search space, A is a point or solution in the search space, δ_i is the value of the ith attribute in the representation of A, V_i are the set of values that δ_i can have, Ξ is the set of all tuples involving combinations of "don't-care" symbols and attributes, ξ is a member of Ξ,] designates a particular subset of a, namely all elements of a for which the corresponding representations match all positions in ξ which are not "$\square$"s. Given a set of observations $\mathsf{a}(1), \mathsf{a}(2), \ldots, \mathsf{a}(t)$ from a, the average payoff μ_ξ of the observed instances $\mathsf{a}(t') \in \xi$ is apportioned to ξ as its credit for the performances of the $A \in \mathsf{a}$ possesing the corresponding set of attributes. Since each $A \in \mathsf{a}$ is an instance of 2^l schemata, it constitutes a valid sample point of 2^l distinct subsets of (or events on) a. This suggests the existence of algorithms which, by testing many possibilities with a single trial, are intrinsically parallel and which store

the relative rankings of μ_ξ for a great many schemata by selecting a small set $\beta \subset a$ [Holland 75].

From one generation to the next, the representation of a particular schemata in the population will increase or decrease according to the relative fitnesses of the strings that correspond to that schemata [Holland 75]. For example, if a particular schemata is sampled by *N* strings at generation *g*, it will be sampled by *N* * (fv(*N*) / fv) strings at generation *g*+1, where fv(*N*) is the average fitness value of the *N* strings and fv is the average population fitness value.

Holland discusses many other important details and observations related to the Schemata Theorem [Holland 75] which are beyond the scope of this thesis. One of the most important observations he makes is that crossover disrupts schemata, so an offspring may not contribute to the representation of its parents' schemata. Therefore, after crossover is performed, a given schemata will both gain and lose strings in a way that is independent of the fitness of its current strings. After taking several factors into consideration, Holland establishes the Schemata Theorem [Holland 75] – Schemata sampled by a set of strings with an average fitness that is larger than the population's average fitness value will receive an exponential increase of sampling strings in successive generations.

CHAPTER IV

PARALLEL GENETIC ALGORITHMS

4.1 Introduction

Consider the problem of delaying premature convergence on an SGA (see Section 3.3). We can take either of two approaches. If we maintain a very large population of individuals on each generation, it will take longer for good individuals to dominate. However, the high computational cost associated with the evaluation of the fitness of each individual in a big population makes the algorithm very inefficient. Another approach is to use a small population and maintain diversity by using some of the methods discussed in Section 3.3. However, the similarity comparisons on which those methods are based are also computationally expensive.

The fact that GA's search numerous points in the problem domain simultaneously, makes them ideal candidates for parallelization. A parallel genetic algorithm (PGA) is an extension of the classical GA that takes advantage of this property to improve its time performance and reduce the likelihood of premature convergence.

Following nature's parallel model, these algorithms maintain multiple, independent populations that each focus on a different area of the problem. The occasional interchange of solutions between these populations introduces diversity and allows for combinations that often result in a global optimum. The following section presents a common classification of PGA's based on their level of parallelism.

4.2 Classification of Parallel Genetic Algorithms

We can distinguish four different models for implementing a PGA according to the desired level of parallelism: Micro-Grain GA, Fine-grain GA, Coarse-Grain GA, and Massively Distributed Parallel GA. These models are briefly described below.

4.2.1 Micro-Grain GA(mgGA)

This model is different from other parallel approaches in that a single population is maintained. Also known as a global GA, it is the most simple model and it is equivalent to an SGA. The parallelism of this model comes from the use of multiple processors for evaluating individual fitnesses [Goodman et al. 94]. A master process maintains a single population and performs classical genetic operators while assigning the task of fitness evaluations to the slave processes (Figure 4). Maximum speedup can be attained if every slave process receives an equal amount of work. This model is useful when the fitness evaluation is the most expensive operation. However, mgGA's do not address the problem of premature convergence [Stracuzzi 98].

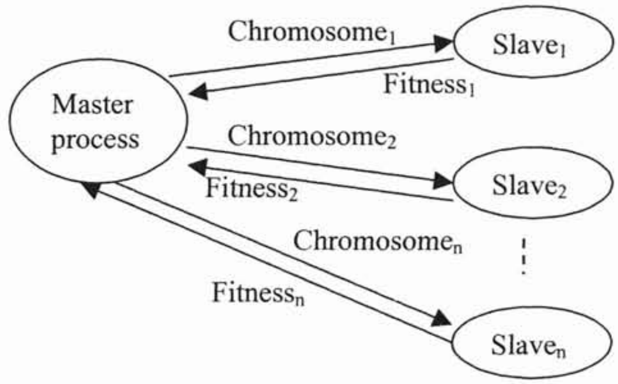
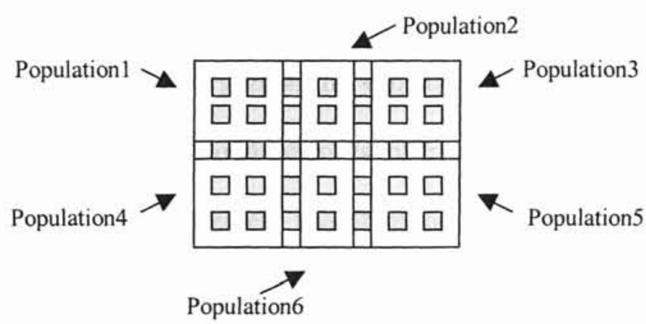


Figure 4. A Micro-grain GA (mgGA)

4.2.2 Fine-Grain GA (fgGA)

This model is a compromise between the micro-grain GA (mgGA) and models with fully separated individual populations [Stracuzzi 98]. The algorithm maintains a single population and allows two individuals to mate only if they are close to one another (neighbors). The entire population can be viewed as a set of small overlapping subpopulations (Figure 5). When selection is performed, only individuals within the same subpopulation may mate. Because some individuals are members of several subpopulations, genetic material is transferred from one population to another [Stracuzzi 98].



The individuals located on the boundaries between populations can mate. Thus, genetic material is transferred among populations.

Figure 5: A Fine-grain GA (fgGA)

The purpose of an fgGA is to delay the spread of genetic information among the subpopulations while still allowing some migration. The main issue affecting this model deals with the connectivity between neighbors. High connectivity makes subpopulations susceptible to premature convergence. On the other hand, low connectivity limits individual interactions and can result in a slowdown of the algorithm [Stracuzzi 98].

4.2.3 Coarse-Grain GA (cgGA)

A cgGA is based upon the theory of punctuated equilibria. In the paper "Distributed Genetic Algorithms for the Floorplan Design Problem", Cohoon et. al describe the theory of punctuated equilibria as follows:

> Punctuated equilibria is based on two principles: allopatric speciation and stasis. Allopatric speciation involves the rapid evolution of new species after a small set of members of a species, peripheral isolates, becomes segregated into a new environment. Stasis, or stability, of a species is simply the notion of lack of change. It implies that after equilibria is reached in an environment, there is very little drift away from the genetic composition of the species. Ideally, a species would persist until its environment changes (or the species would drift very little). Punctuated equilibria stresses that a powerful method for generating new species is to thrust an old species into a new environment, where change is beneficial and rewarded. For this reason, we should expect a genetic algorithm approach based on punctuated equilibria to perform better than the typical single environment scheme [Cohoon et al. 91].

The implication of this theory upon the structure of a GA is that given a single large population in which the environment is unchanging, equilibrium will be rapidly attained as the population converges. The offspring produced will be very similar to each other and to their parents, causing the population to stabilize on a local optimum. Allopatric speciation indicates that evolution can continue by the introduction of stabilized species into different subpopulations [Cohoon et al. 91].

Papadopoulos indicated the effectiveness of the cgGA in solving many "GA-hard" problems which other GA's are not able to solve [Papadopoulos 94]. He outlines a common implementation of a cgGA. A set of n individuals can be assigned to a dedicated processor. Given that N processors are available, the size of the total population is nxN. During a major iteration or epoch, every processor works in parallel, yet independently, evolving its individuals [Papadopoulos 94]. In theory, a processor should continue evolving its individuals until it reaches equilibrium. However, because there is no known adequate equilibrium stopping criteria, an epoch consists of a fixed number of generations, which greatly simplifies the task of synchronizing the processors [Cohoon et al. 91]. When the processors stop, chromosomes are interchanged between subpopulations. This migration of individuals has the effect of introducing new genetic material into populations that may have slowed down their evolution due to an equilibrium [Papadopoulos 94]. Algorithm B below is a generalized cgGA [Stracuzzi 98]

```
Global Data
graph          migration_topology;

Local Data
population     my_pop, my_new_pop, migrant_pop;
float          p_cross, p_mutation,
               migration_rate;     /* percent of pop moved during each migration */
int            N,                  /* population size */
               N_migrants;         /* n_migrants = N * migration_rate */

for all processing nodes
  my_pop = new random individual(s)
  evaluate(my_pop)
  while termination criteria not satisfied
    if migration criteria satisfied
      if using dynamic network connection
        update(migration_topology)
      migrant_pop = select(N_migrants, my_pop)
      send migrant_pop to another node according to migration_topology
      migrant_pop = receive migrants from another node
      add migrant_pop to my_pop and maintain population size N
      my_new_pop = select(N, my_pop)
      my_pop = crossover(p_cross, my_new_pop)
      my_pop = mutate(p_mutation, my_pop)
      evaluate(my_pop)
  end while
end forall
```

Algorithm B. A generalized cgGA [Stracuzzi 98]

The efficiency of a cgGA depends on the choices of several new parameters. The following are some of the strategies that were considered while developing a simulated cgGA as part of the thesis work:

- Migration Policy

The following parameters define the migration mechanism [Rebaudergo and Reorda 92]:

Migration Frequency determines the number of generations between two migrations (i.e., the size of an epoch). Frequent communications are useless because similar individuals are transmitted on each migration. Less frequent migrations increase the running time of the algorithm [Rebaudergo and Reorda 92].

Migration Size determines the number of individuals composing each migration. Sending too many individuals will result in a decrease of the average fitness [Rebaudergo and Reorda 92]. On the other hand, if only a few individuals are transmitted, they may be quickly eliminated if the receiving subpopulation has a much higher average fitness value.

Migrant Selection determines which immigrants are chosen within the source subpopulation. The individuals with the highest fitness could be chosen or they could be selected at random [Rebaudergo and Reorda 92]. The most common method is to choose an individual with probability proportional to its fitness value. In this manner, diversity increases as it is not only the good individuals that migrate.

Whether the communication between processing nodes is done in a synchronous or an asynchronous manner, is another issue to be considered.

- Connection Schemes

There are two widely used connection schemes: *static connection scheme* and *dynamic connection scheme*.

In a *static connection scheme*, the connections between processors are established at the beginning and not modified during execution. There are several different topologies: rings, lines, n-cubes, etc. [Goodman et al. 94]. This type of connection scheme is used in this thesis.

In a *dynamic connection scheme*, the network topology is allowed to change during run time.

- Node Structure

There are two different approaches depending on the similarity of the SGA's running on each processor: *homogeneous island GA*, and *heterogeneous island GA*.

In a *homogeneous island GA*, every processor uses the same parameters (crossover rate, mutation rate, population size, etc.) [Goodman et al. 94].

A *heterogeneous island GA* allows subpopulations with different parameters to evolve. This will increase the chance of finding an ideal set of parameters [Goodman et al. 94].

4.2.4 Massively Distributed Parallel GA (mdpGA)

In an mdpGA, every processor is assigned a small subpopulation (i.e., 10 individuals). Because of the small population size, selection must be done carefully [Stracuzzi 98].

CHAPTER V

DESIGN AND IMPLEMENTATION

SIPAGAR (SImulated PArallel Genetic Algorithm for finding Ramsey numbers) is implemented as a JAVA applet. JAVA applets provide a convenient way of displaying graphs and make the simulation very portable by being able to use it on the Web.

5.1 Classes and Methods

The main class of SIPAGAR is the *Ramsey* class. It inherits from the JAVA *Applet* class. The method *evolve* starts the threads of all the subpopulations and then starts the thread of the *Gamigration* class. The *permutation* class is the representation of individual solutions (i.e., graphs). The *group* class inherits from the JAVA *Thread* class. It represents a subpopulation of permutations that are evolved towards an optimal solution. The method *evolve* uses an object of class *Decoder* to assign fitness values to each permutation in the subpopulation. It also uses procedures in packages *crossover*, *mutation*, and *selection* to perform genetic operations. The method *decode* in class *Decoder* assigns a fitness value to a permutation according to the evaluation function. It uses the supporting functions of classes *table* and *triangle* for this purpose. Given a permutation of the edges of a complete graph, we need to color the edges in that order. Because the edges are numbered and mapped to a single dimensional array, we need two end vertices of an edge to check if a monochromatic subgraph is being formed as a result of a particular coloring. The class *table* provides supporting functions to obtain the (i,j) coordinates of a particular edge. Given the (i,j) coordinates of an edge, the supporting function *find_triangle* in class *triangle* checks if a triangle is being formed for all possible colorings of the edge and assigns to the edge the color that results in the fewest number of monochromatic triangles being formed.

The package *crossover* contains class *pmx*, which implements Partially Matched Crossover. The package *mutation* contains class *swap*, which implements swap mutation. The class *Roulette* in package *Selection* implements Roulette-Wheel-based selection. It takes a population of permutations as input, and returns a single permutation which is selected with a probability proportional to its fitness value relative to the average fitness value. This operation is implemented with an array of floating point numbers. Each array element corresponds to a permutation in the population and is initialized to the sum of fitness values of all permutations up to that particular permutation. A random floating point number between zero and the sum of all fitness values in the population is generated. The first permutation whose array value exceeds this value is chosen.

The *group_GUI* class uses the supporting functions in class *graph* to display the graph of a particular permutation in a subpopulation. Statistical data for each subpopulation is gathered in class *group_stats*. The class *global_stats* stores and graphically displays statistical information for all subpopulations. It displays the optimal fitness value for a particular run (the goal). As the subpopulations evolve, they provide information to a *global_stats* object about the best locally found permutation. The *global_stats* object displays a permutation with the best fitness value found so far among all subpopulations. An object of class *Gamigration* is a thread that once started, continuously checks the condition that triggers migration among the subpopulations. When the condition (migration frequency) is satisfied, the

Gamigration object performs the migration according to the migration criteria (topology, size, selection).

Migration is done synchronously. A subpopulation stops when the migration criteria has been locally satisfied. The *Gamigration* object triggers migration only when the migration criteria has been satisfied in all subpopulations. After migration is done, *Gamigration* resumes the evolution of all subpopulations. The class *GAException* handles exceptions that may occur when running the simulation. The class *Ramsey_GUI* implements the simulation's graphical user interface and connects events to listeners in class *Ramsey_Listener*. The classes *Ramsey_Action_Listener* and *Ramsey_Item_Listener* handle the events in the graphical user interface.

5.2 Graphical User Interface

The interface allows the user to input the problem, define the parameters, and run either a simple GA or the simulated parallel version and view the results. The main window (Figure 6) is divided into five parts: problem construction, control buttons, global statistics, log window, and local statistics.

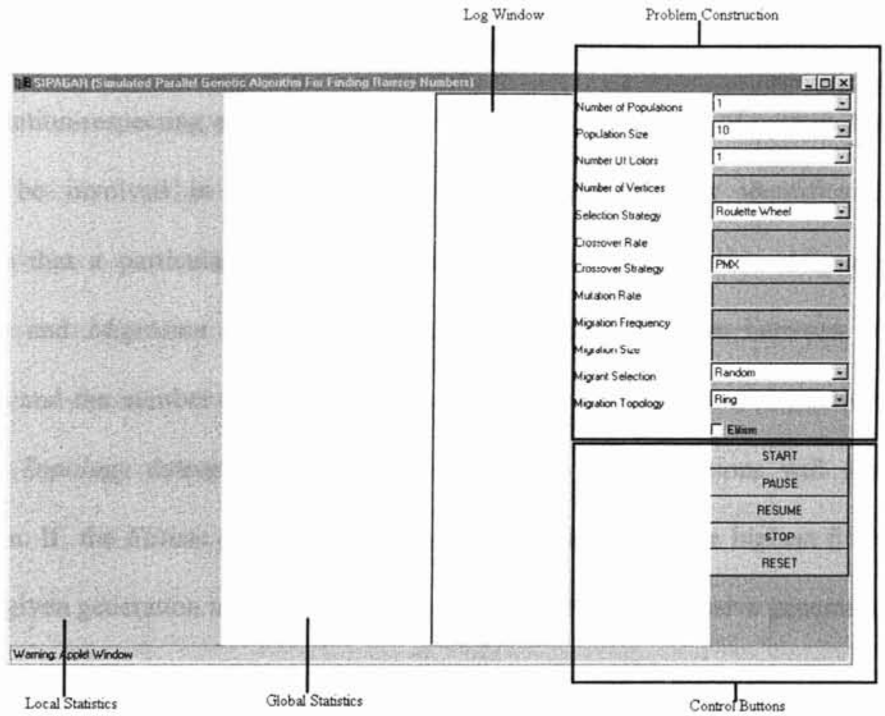


Figure 6. SIPAGAR's Graphical User Interface

In Figure 6, the problem construction part is used to enter the problem and define the GA parameters. The *Number of Populations* parameter determines whether a simple GA or the simulated cgGA is to be run. For a simple GA, the user only needs to assign the value 1 to *Number of Populations* and a value in the range 2 to 6 to run the simulated parallel version. *Population Size* determines the number of permutations that will evolve in each subpopulation. Choosing a larger value for this parameter does not necessarily lead to a better solution since it will slow down the execution. *Number Of Colors* specifies the number of colors that will be used to color the edges of the particular complete graph with *Number of Vertices* vertices. For instance, in order to test whether R(3,3,3,3) > 51, the user would set *Number of Colors* to 4 and *Number of Vertices* to 51. *Selection Strategy* and *Migrant Selection* identify the strategies that will be used to choose the permutations that will mate and migrate to different subpopulation respectively. *Crossover Strategy* and *Crossover Rate* indicate the permutation-respecting operator that will be used and the percent of permutations that will be involved in crossover respectively. *Mutation Rate* identifies the probability that a particular permutation will undergo swap mutation. *Migration Frequency* and *Migration Size* indicate the number of generation between two migrations and the number of permutations composing each migration, respectively. *Migration Topology* determines the way in which the subpopulations will share information. If the *Elitism* option is checked, a permutation with the highest fitness value in a given generation is guaranteed to be a member of the successive generation.

The control buttons are used to start, stop, pause, and resume execution. The *reset* button is used to stop execution and set parameters to their default values. The log window displays errors that may occur during execution or while setting the parameters. It also indicates when migration takes place and the migration pattern among subpopulations. The local statistics part displays a graphical representation of the best permutation as well as local statistical information of each subpopulation. The basic statistical information gathered for each subpopulation is the following:

gen #: the current generation number.

best f: the fitness value of the permutation with the best overall fitness.

av.f: average fitness value of all permutations in a subpopulation.

change: change in average fitness value from the previous generation.

The global statistics part displays an enlarged graphical representation of the best permutation among all subpopulations as well as global statistical information. The basic global statistical information gathered is the following:

Optimal Fitness: The optimal fitness value of any permutation for a particular problem (the goal).

Best Permutation: Actual edge permutation of the best permutation among all subpopulations (the solution).

Coloring: Colors assigned to the edges of the best permutation among all the subpopulations.

Best: Fitness value of the best permutation among all the subpopulations.

Figure 7 is a snaphot of SIPAGAR when executed with 6 subpopulations. Code Listings are included in Appendix F.

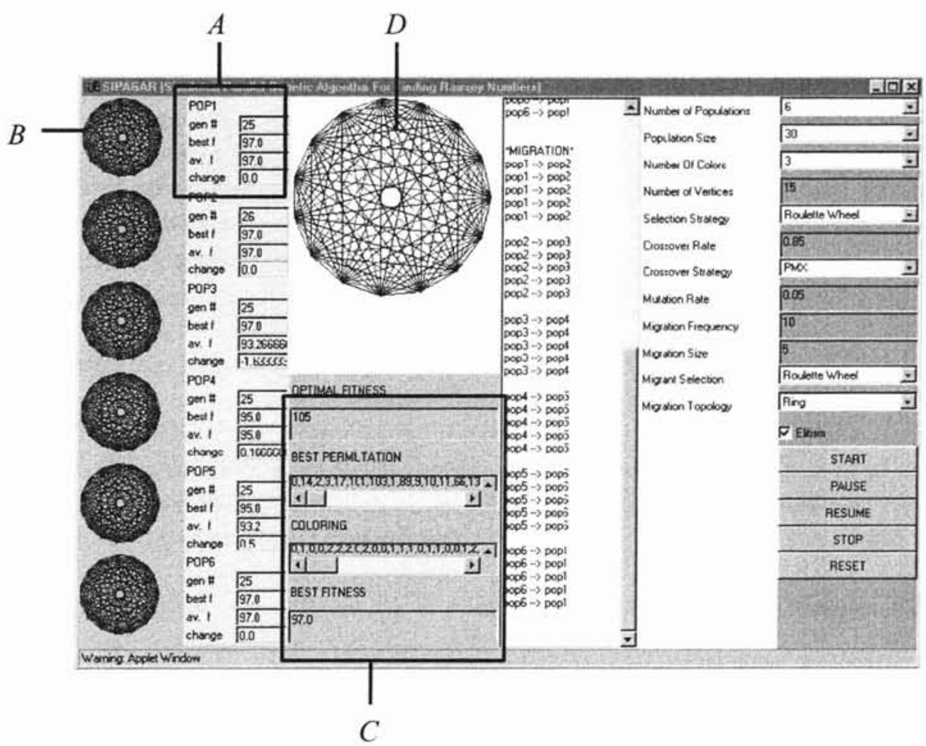


Figure 7. Snapshot of SIPAGAR

In Figure 7, the box labelled *A* contains the local statistics for population 1. At generation 25, the highest fitness value in population 1 was 97.0. The K_{15} labelled *B* is the Ramsey graph of a permutation in population 1 with fitness value 97.0. The global statistics are inside the box labelled *C*. The optimal fitness value is 105 and corresponds to a K_{15} all of whose 105 edges can be colored with no resulting monochromatic triangle in either of the 3 colors (the optimal permutation). The K_{15} labelled *D* is an enlarged display of the Ramsey graph of a permutation with highest fitness value among the 6 populations.

CHAPTER VI

EXPERIMENTS

In this chapter, we will compare the performance of a traditional GA with that of the simulated cgGA with several parameter values, when applied to the problem of finding R(3,3,3). R(3,3,3), which is equal to 17, is the only known multicolor Ramsey Number. It is used to observe and compare the rate of premature convergence as well as the performance of the traditional GA and the simulated cgGA. Because the value of R(3,3,3) is known, the global optimum for a particular run is also known, so we can detect when the algorithm converges to a local optimum.

6.1 Results of Runs for R(3,3,3)

As previously mentioned, R(3,3,3) is known to be equal to 17 and it is used in this experiment for performance comparisons only. Every coloring of the edges of a complete graph with 17 vertices in 3 colors will give rise to a triangle that is monochromatic in one of the 3 colors. To do the comparison of performance mentioned above, we use the problem of finding a complete graph with 16 vertices in 3 colors and containing no monochromatic triangle in either of the 3 colors (the optimal solution). The fitness value of the optimal solution is 120 (all the edges of the complete graph can be colored without any resulting monochromatic triangle). The problem was first run on the traditional GA option of SIPAGAR with the following parameter values:

Number of Populations: 1

Population Size: 20

Number of Colors: 3

Number of Vertices: 16

Selection Strategy: Roulette-Wheel

Crossover Rate: 0.85

Crossover Strategy: PMX

Mutation Rate: 0.05

Migration Frequency: N/A (Not Applicable)

Migration Size: N/A (Not Applicable)

Migrant Selection: N/A (Not Applicable)

Migration Topology: N/A (Not Applicable)

Elitism: True

The following table shows the statistics for the first 300 generations of this run.

Table I. Statistics for R(3,3,3) on traditional GA
(Number of Populations = 1, Population Size = 20)

Generation	Best Fitness	Average Fitness
0	104.00	98.25
10	104.00	99.85
20	104.00	100.50
30	104.00	100.75
40	104.00	101.05
50	104.00	101.50
60	104.00	101.85
70	104.00	101.45
80	104.00	101.85
90	104.00	102.35
100	104.00	102.75
110	104.00	102.50
120	104.00	102.70
130	104.00	102.85
140	104.00	103.30
150	104.00	103.40
160	104.00	102.80
170	104.00	102.90
180	104.00	103.00
190	104.00	102.80
200	104.00	102.80
210	104.00	102.00
220	104.00	102.20
230	104.00	102.60
240	104.00	103.20
250	104.00	103.00
260	104.00	103.60
270	104.00	103.80
280	104.00	103.80
290	104.00	103.80
300	104.00	104.00
> 300	104.00	104.00

From Table I we can observe that when the population size is small, the traditional GA converges very rapidly to a local optimum. Figure 8 shows a snaphot of this run when stopped at generation 107,199.

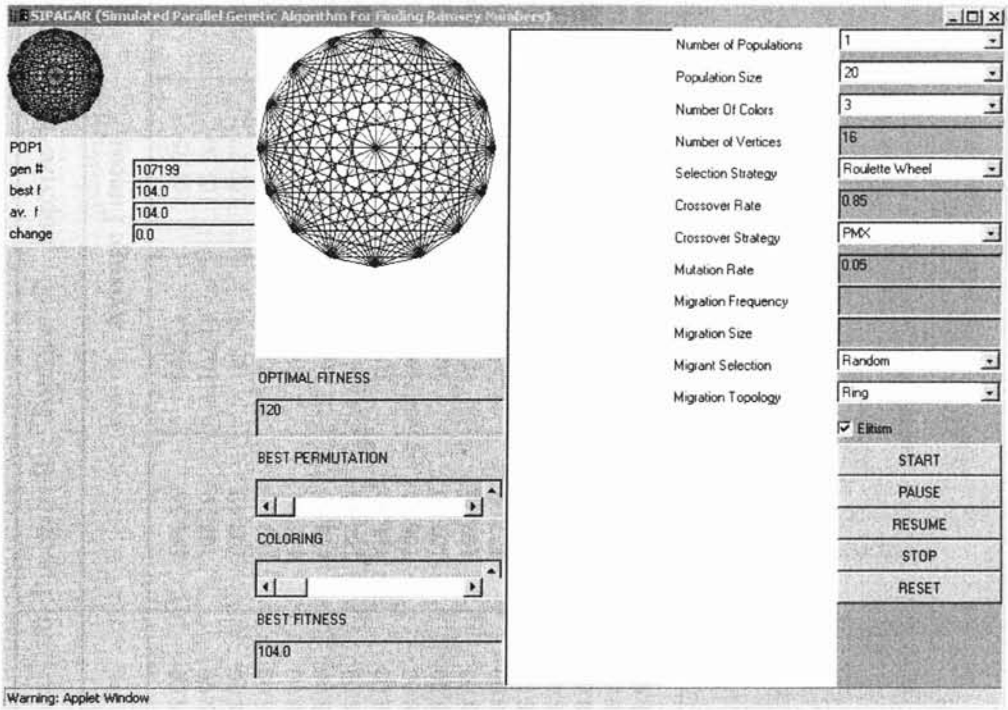


Figure 8. Snapshot of First Run

In order to observe the relationship between population size and the rate of premature convergence, the same problem was run on the traditional GA with population sizes of 40, 60, 80, and 100 with the values for all the other parameters kept unchanged. A summary of the results is shown in Table II (Appendix F contains statistics reported every 10 generations). Figure 9 shows the effect of population size on premature convergence.

Table II. Statistics For R(3,3,3) with different population sizes (Number of Populations = 1, Population Size = 40,60,80,100)

	PopSize=40	PopSize=60	PopSize=80	PopSize=100	PopSize=40	PopSize=60	PopSize=80	PopSize=100
Generation	Best Fitness				Average Fitness			
0	106.00	107.00	106.00	106.00	101.56	101.51	101.75	101.75
100	107.00	107.00	107.00	108.00	103.29	102.41	101.90	103.19
200	107.00	107.00	107.00	108.00	102.49	105.32	102.27	103.48
300	107.00	107.00	107.00	108.00	103.66	104.56	102.23	103.65
400	107.00	107.00	107.00	108.00	103.94	104.40	103.18	104.64
500	107.00	107.00	107.00	108.00	104.03	104.50	104.00	103.72
600	107.00	107.00	107.00	108.00	104.40	104.95	104.50	104.25
700	107.00	107.00	107.00	108.00	106.50	106.25	104.63	104.00
800	107.00	107.00	107.00	108.00	106.83	106.50	103.86	104.60
850	107.00	107.00	107.00	108.00	107.00	105.35	103.22	105.30
900	107.00	107.00	107.00	108.00	107.00	104.80	106.16	106.35
1000	107.00	107.00	107.00	108.00	107.00	105.85	106.81	107.15
1090	107.00	107.00	107.00	108.00	107.00	106.65	106.56	108.00
1100	107.00	107.00	107.00	108.00	107.00	106.90	106.75	108.00
1130	107.00	107.00	107.00	108.00	107.00	107.00	106.81	108.00
1160	107.00	107.00	107.00	108.00	107.00	107.00	107.00	108.00
1200	107.00	107.00	107.00	108.00	107.00	107.00	107.00	108.00

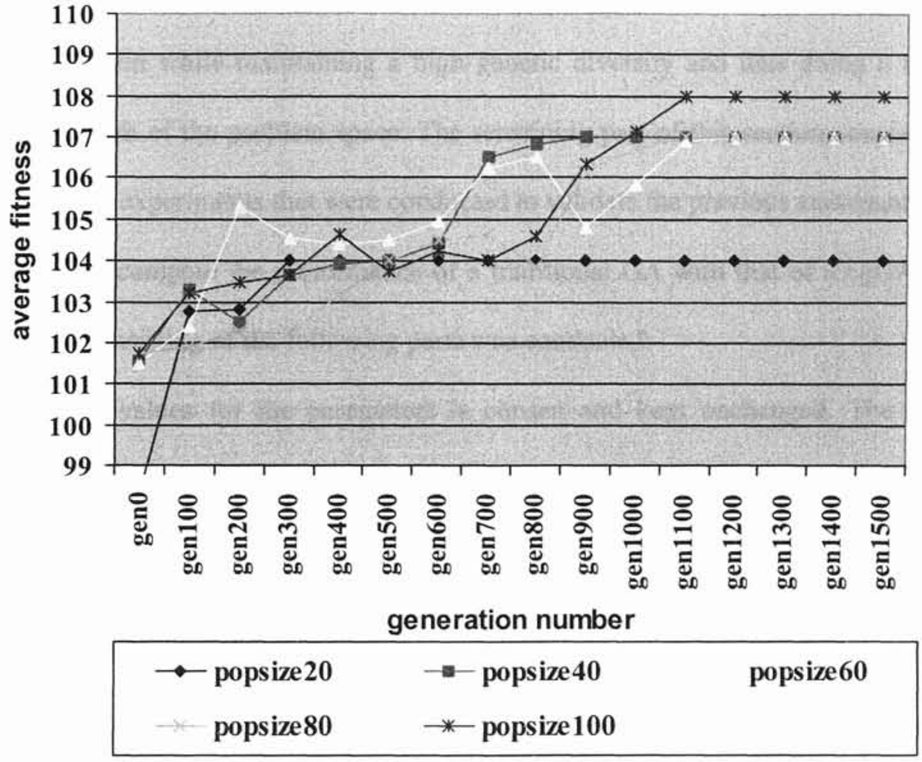


Figure 9. Effect of Population Size on Premature Convergence

As Figure 9 illustrates, increasing the population size results in a solution that is nearer to the optimal solution. It also delays premature convergence to local optima. In a small population, a permutation with a relatively high fitness value will be selected very often and its descendants will quickly dominate the population. This will result in reduced genetic diversity and the search will quickly stop after converging on a local optimum. On the other hand, as the population size increases, many permutations are evaluated at each generation and premature convergence is discouraged. This results in more paths being searched and thus an increase in the fitness of the solution. However, in our implementation, the time it takes the DECODER to evaluate the fitness of a permutation dominates the execution time. Therefore, a very large population can be very expensive in terms of time, so a smaller population is desirable. A cgGA is capable of maintaining the time performance of a small population while maintaining a high genetic diversity and thus doing a more complete search of the problem space. The remaining part of this section consists of outlines of the experiments that were conducted to validate the previous statement.

In order to compare the performance of a traditional GA with that of a cgGA, an experiment consisting of the following parts was conducted:

1) A set of values for the parameters is chosen and kept unchanged. The only parameter with a variable value is *Number of Populations*. The following parameter values were chosen:

Number of Populations: VARIABLE

Population Size: 20

Number of Colors: 3

Number of Vertices: 16

Selection Strategy: Roulette-Wheel

Crossover Rate: 0.85

Crossover Strategy: PMX

Mutation Rate: 0.05

Migration Frequency: 20 (if *Number Of Populations* > 1)

Migration Size: 3 (if *Number Of Populations* > 1)

Migrant Selection: Roulette-Wheel (if *Number Of Populations* > 1)

Migration Topology: Ring (if *Number Of Populations* > 1)

Elitism: True

2) A problem is chosen and kept unchanged. R(3,3,3) was chosen.

3) The problem is run on a traditional GA (*Number of Populations* = 1) 10 times. The

purpose of repeating the run 10 times is to obtain a better view of the average performance of the traditional GA.

4) The problem is run on the simulated cgGA with *Number Of Populations* equal to 2,3,4,5, and 6. The results of each one of these runs is compared with the performance of the traditional GA of Step 3 above.

A summary of the results of Step 3 is shown in Table III below. Figure 10 shows the change in the average fitness value of the populations in each one of the 10 runs.

Table III. Statistics for 10 runs of R(3,3,3) (Number of Populations = 1, Population Size = 20)

Generation	Best Fitness Run 1	Best Fitness Run 2	Best Fitness Run 3	Best Fitness Run 4	Best Fitness Run5	Average Fitness Run 1	Average Fitness Run 2	Average Fitness Run 3	Average Fitness Run 4	Average Fitness Run5
0	107.00	107.00	104.00	106.00	105.00	101.25	101.55	100.55	101.75	101.60
50	107.00	107.00	106.00	106.00	105.00	104.50	104.00	102.65	102.45	101.70
100	107.00	107.00	106.00	106.00	105.00	105.75	104.60	101.60	100.80	101.60
150	107.00	107.00	106.00	106.00	105.00	106.40	103.70	103.20	101.00	102.95
200	107.00	107.00	106.00	106.00	105.00	105.80	105.20	103.60	100.90	102.95
250	107.00	107.00	106.00	106.00	105.00	106.00	105.20	106.00	102.60	104.45
300	107.00	107.00	106.00	106.00	105.00	107.00	104.00	106.00	104.80	104.85
350	107.00	107.00	106.00	106.00	105.00	107.00	107.00	106.00	103.60	104.50
450	107.00	107.00	106.00	106.00	105.00	107.00	107.00	106.00	106.00	105.00
500	107.00	107.00	106.00	106.00	105.00	107.00	107.00	106.00	106.00	105.00
550	107.00	107.00	106.00	106.00	105.00	107.00	107.00	106.00	106.00	105.00
600	107.00	107.00	106.00	106.00	105.00	107.00	107.00	106.00	106.00	105.00

Generation	Best Fitness Run 6	Best Fitness Run 7	Best Fitness Run 8	Best Fitness Run 9	Best Fitness Run10	Average Fitness Run 6	Average Fitness Run 7	Average Fitness Run 8	Average Fitness Run 9	Average Fitness Run10
0	104.00	105.00	105.00	105.00	106.00	101.05	100.75	101.60	100.80	101.35
50	105.00	106.00	107.00	105.00	106.00	102.65	102.10	103.45	102.00	100.85
100	105.00	106.00	107.00	105.00	106.00	102.15	101.20	105.50	102.15	102.05
150	105.00	106.00	107.00	105.00	106.00	101.85	102.70	105.10	103.05	105.70
200	105.00	106.00	107.00	105.00	106.00	102.60	104.20	104.20	103.50	106.00
250	105.00	106.00	107.00	105.00	106.00	102.00	105.40	103.95	103.95	106.00
300	105.00	106.00	107.00	105.00	106.00	101.55	106.00	104.00	105.00	106.00
350	105.00	106.00	107.00	105.00	106.00	102.00	106.00	105.00	105.00	106.00
450	105.00	106.00	107.00	105.00	106.00	104.10	106.00	104.80	105.00	106.00
500	105.00	106.00	107.00	105.00	106.00	104.55	106.00	105.00	105.00	106.00
550	105.00	106.00	107.00	105.00	106.00	103.95	106.00	106.60	105.00	106.00
600	105.00	106.00	107.00	105.00	106.00	105.00	106.00	107.00	105.00	106.00

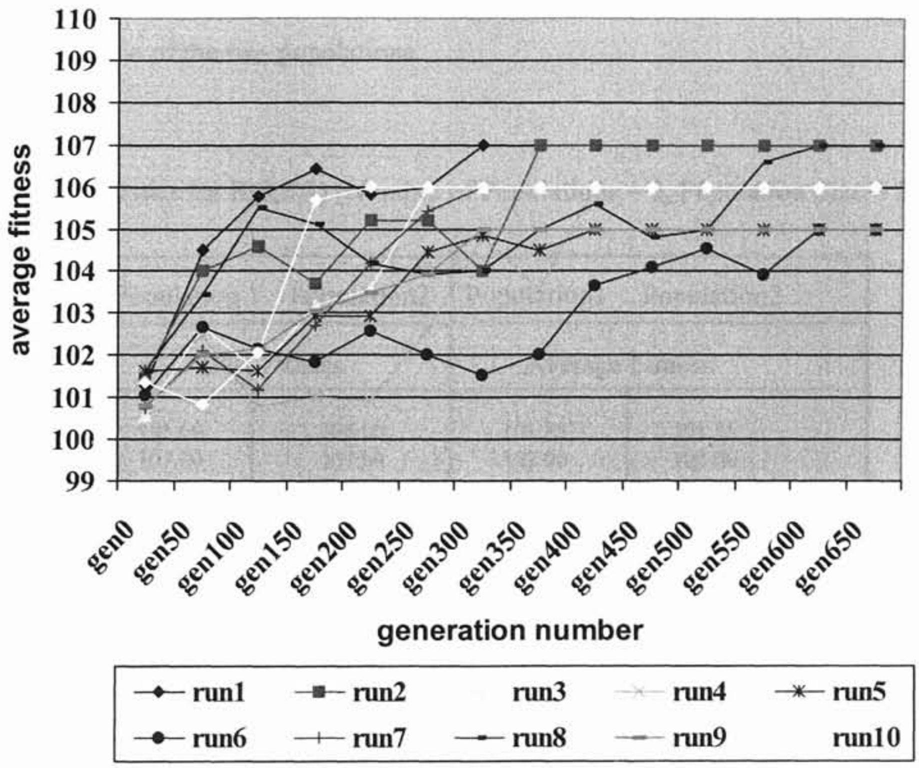


Figure 10. Average Fitness in 10 runs
Number of Populations = 1
Population Size = 20

As Figure 10 shows, there is no improvement after at most 600 generations and the search stagnates. Many runs were performed for each of the choices of *Population Size* and similar results were obtained. As shown in Table II and Figure 9, the larger the value of *Population Size*, the longer it took the population to converge to a local optimum and thus the search to stagnate. However, regardless of the value of *Population Size*, the traditional GA was never able to find a fitness value better than 108. After converging to a local optimum, the search stagnated even when allowed to run for hundreds of thousands of generations. Continuing with the experiment, the problem was run on the simulated cgGA with *Population Size* = 2.

Table IV is a summary of the results obtained (Appendix F contains statistics reported every 10 generations). Figure 11 shows the change in the average fitness value of each one of the two populations.

Table IV. Statistics for R(3,3,3) (Number of Populations = 2, Population Size = 20)

	Population 1	Population2	Population1	Population2
Generation	Best Fitness		Average Fitness	
0	105.00	106.00	101.85	101.55
50	107.00	107.00	102.90	102.80
100	107.00	107.00	103.75	102.80
150	107.00	107.00	103.95	100.25
200	107.00	107.00	105.80	102.85
250	107.00	107.00	101.05	105.80
300	107.00	107.00	103.80	106.80
350	107.00	107.00	106.60	106.30
400	107.00	107.00	106.90	107.00
450	107.00	107.00	106.60	106.60
500	107.00	107.00	106.60	106.80
550	107.00	107.00	105.90	105.60
600	107.00	107.00	105.60	106.30
650	107.00	107.00	106.80	106.70
700	107.00	107.00	106.80	107.00
750	107.00	107.00	106.10	106.80
800	107.00	107.00	106.40	106.40
850	107.00	107.00	106.60	106.20
900	107.00	107.00	106.80	105.60
950	107.00	107.00	106.70	105.40
1000	107.00	107.00	106.50	105.80
1050	107.00	107.00	106.20	106.30
1100	107.00	107.00	106.40	106.90
1150	107.00	107.00	107.00	106.90
1200	107.00	107.00	107.00	107.00

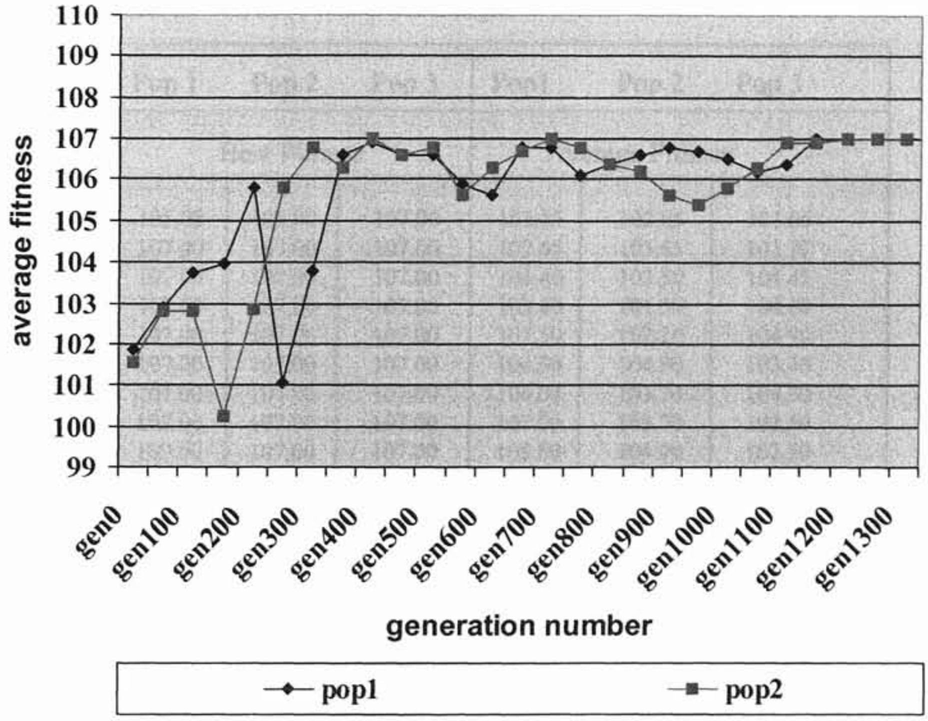


Figure 11. Average Fitness versus Generation Number
Number of Populations = 2
Population Size = 20

Comparing Figure 11 with Figure 10, it is seen that the average fitness values of the two populations take about twice as many generations to converge to a suboptimal solution.

Table V summarizes the results obtained when the problem was run with 3 populations. Figure 12 shows the change in the average fitness value of each one of the three populations.

Table V. Statistics for R(3,3,3) (Number of Populations = 3, Population Size = 20)

	Pop 1	Pop 2	Pop 3	Pop1	Pop 2	Pop 3
Generation	Best Fitness			Average Fitness		
0	105.00	106.00	107.00	101.65	102.65	101.05
50	107.00	107.00	107.00	102.65	103.85	101.95
100	107.00	107.00	107.00	104.40	103.50	104.45
150	107.00	107.00	107.00	103.40	101.50	104.20
200	107.00	107.00	107.00	105.50	102.20	104.90
250	107.00	107.00	107.00	104.30	104.90	103.40
300	107.00	107.00	107.00	104.00	103.70	104.30
350	107.00	107.00	107.00	107.00	103.70	102.50
400	107.00	107.00	107.00	105.50	104.90	102.80
450	107.00	107.00	107.00	103.10	107.00	104.00
500	107.00	107.00	107.00	105.80	104.30	106.40
550	107.00	107.00	107.00	106.70	106.10	106.70
600	107.00	107.00	107.00	106.40	106.40	107.00
650	107.00	107.00	107.00	107.00	106.40	107.00
700	107.00	107.00	107.00	107.00	107.00	107.00
750	107.00	107.00	107.00	107.00	107.00	107.00

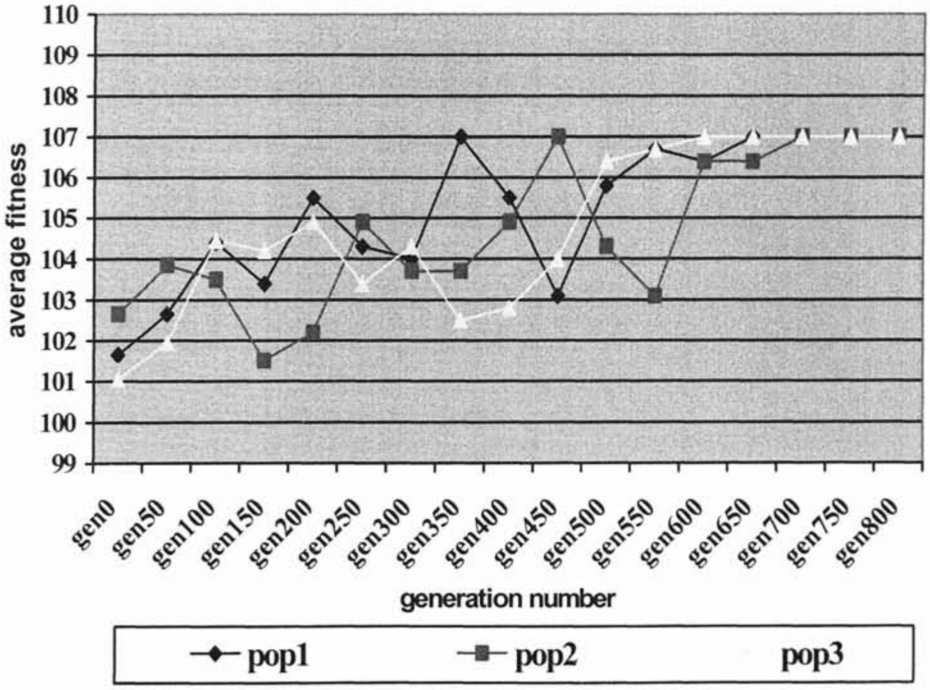


Figure 12. Average Fitness versus Generation Number
Number of Populations = 3
Population Size = 20

A comparison between Figures 11 and 12 shows a lack of improvement when increasing the number of populations from 2 to 3. The fitness values of the globally best permutations were identical in both cases. Furthermore, the populations in the run with 3 populations converged more rapidly than those in the run with 2 populations. This may be due to the fact that the local optimum (a permutation with fitness value equal to 107) was present in generation 0 of the run with 3 populations. The early appearance of a permutation with a relatively high fitness value may have triggered these results.

Table VI summarizes the results obtained when the problem was run with 4 populations. Figure 13 shows the change in the average fitness value of each one of the four populations.

Table VI. Statistics for R(3,3,3) (Number of Populations = 4, Population Size = 20)

Generation	Best Fitness				Average Fitness			
	Pop 1	Pop 2	Pop 3	Pop 4	Pop 1	Pop 2	Pop3	Pop4
0	105.00	106.00	106.00	106.00	100.85	101.60	102.40	101.70
50	105.00	106.00	106.00	106.00	101.50	102.45	102.70	102.35
100	106.00	106.00	106.00	106.00	102.60	103.45	103.25	102.35
150	106.00	106.00	106.00	106.00	102.90	103.30	103.95	105.60
200	108.00	106.00	108.00	108.00	103.25	105.35	103.55	105.60
250	108.00	108.00	108.00	108.00	106.00	104.05	105.80	103.30
300	108.00	108.00	108.00	108.00	104.40	104.30	104.10	107.30
350	108.00	108.00	108.00	108.00	105.00	104.85	105.35	106.50
400	108.00	108.00	108.00	108.00	104.00	105.40	105.00	107.85
450	108.00	108.00	108.00	108.00	106.00	103.20	105.40	107.50
500	108.00	108.00	108.00	108.00	105.40	103.85	105.80	106.55
550	108.00	108.00	108.00	108.00	104.65	104.00	106.00	105.80
600	108.00	108.00	108.00	108.00	106.60	106.00	107.20	107.40
650	108.00	108.00	108.00	108.00	108.00	104.80	107.60	107.60
700	108.00	108.00	108.00	108.00	108.00	105.80	107.80	107.80
750	108.00	108.00	108.00	108.00	107.20	106.00	107.80	108.00
800	108.00	108.00	108.00	108.00	108.00	107.80	108.00	108.00
850	108.00	108.00	108.00	108.00	108.00	108.00	108.00	108.00
900	108.00	108.00	108.00	108.00	108.00	108.00	108.00	108.00

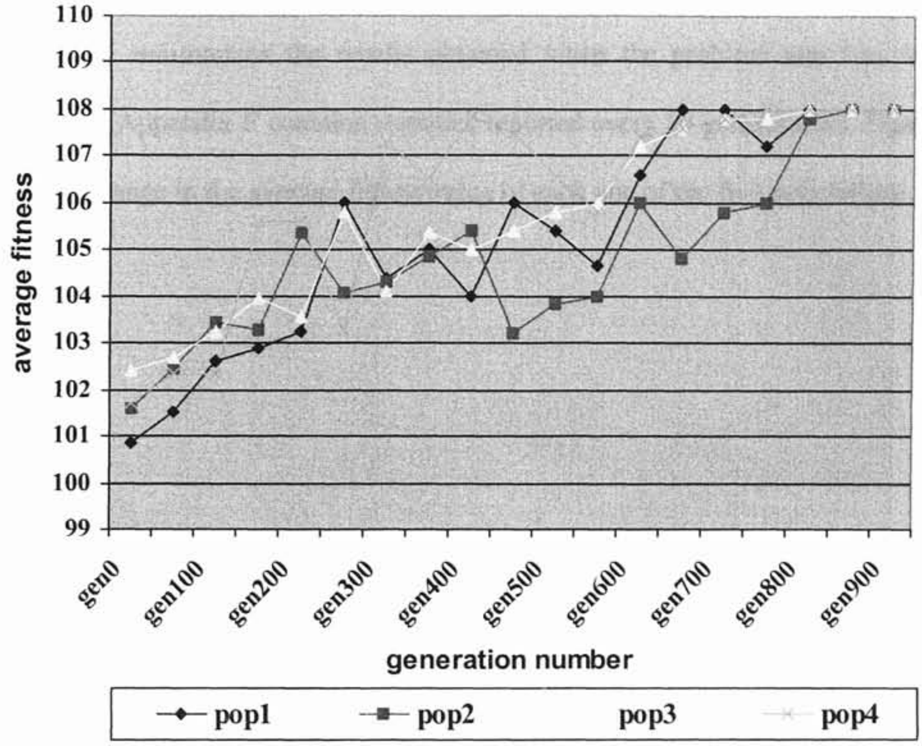


Figure 13. Average Fitness versus Generation Number
Number of Populations = 4
Population Size = 20

As can be observed, a permutation with fitness value equal to 108 is found when a run with 4 populations is performed. This is the highest fitness value that was achieved when multiple runs of the traditional GA with population sizes equal to 40, 60, 80, or 100 were done. This result is very significant from the point of view of computing time. As previously discussed, the time it takes the DECODER to evaluate the fitness of a permutation dominates the execution time thus a very large population can be very expensive in terms of time. However, the simulated cgGA with 4 populations and a small population size of 20 was able to find a permutation with a fitness value equal to the one found by a traditional GA with a large population size. This suggests that if several processors are available and a truly parallel cgGA is implemented, a great improvement in execution time would be obtained.

Table VII summarizes the results obtained when the problem was run with 5 populations (Appendix F contains statistics reported every 10 generations). Figure 14 shows the change in the average fitness value of each one of the five populations.

Table VII. Statistics for R(3,3,3) (Number of Populations = 5, Population Size = 20)

Generation	Best Fitness					Average Fitness				
	Pop 1	Pop 2	Pop 3	Pop 4	Pop 5	Pop 1	Pop2	Pop3	Pop4	Pop5
0	105.00	104.00	106.00	105.00	107.00	102.35	100.75	101.75	101.65	101.70
50	107.00	107.00	106.00	105.00	108.00	102.80	104.40	102.50	102.30	102.40
100	106.00	105.00	107.00	107.00	108.00	103.25	103.95	103.85	103.50	104.90
150	106.00	108.00	107.00	107.00	108.00	105.05	105.25	104.35	105.50	106.00
200	108.00	108.00	107.00	108.00	108.00	104.50	106.05	105.75	105.95	106.30
250	110.00	108.00	108.00	108.00	108.00	106.20	105.45	106.70	104.75	103.60
300	110.00	110.00	110.00	108.00	108.00	104.85	106.75	106.90	106.85	106.45
350	110.00	110.00	110.00	108.00	108.00	107.25	105.65	108.00	105.90	105.95
400	110.00	110.00	110.00	110.00	110.00	106.15	107.50	107.50	106.35	104.80
450	110.00	110.00	110.00	110.00	110.00	107.10	105.15	107.75	107.75	107.40
500	110.00	110.00	110.00	110.00	110.00	107.50	106.00	107.25	106.95	106.45
550	110.00	110.00	110.00	110.00	110.00	107.00	108.00	104.25	105.25	107.10
600	110.00	110.00	110.00	110.00	110.00	108.50	104.75	107.75	107.85	107.00
650	110.00	110.00	110.00	110.00	110.00	107.75	108.00	105.00	107.75	107.55
700	110.00	110.00	110.00	110.00	110.00	108.75	110.00	106.75	107.00	108.00
750	110.00	110.00	110.00	110.00	110.00	109.50	107.50	109.75	108.00	107.25
800	110.00	110.00	110.00	110.00	110.00	110.00	109.00	110.00	108.00	109.75
850	110.00	110.00	110.00	110.00	110.00	110.00	110.00	110.00	110.00	110.00
900	110.00	110.00	110.00	110.00	110.00	110.00	110.00	110.00	110.00	110.00

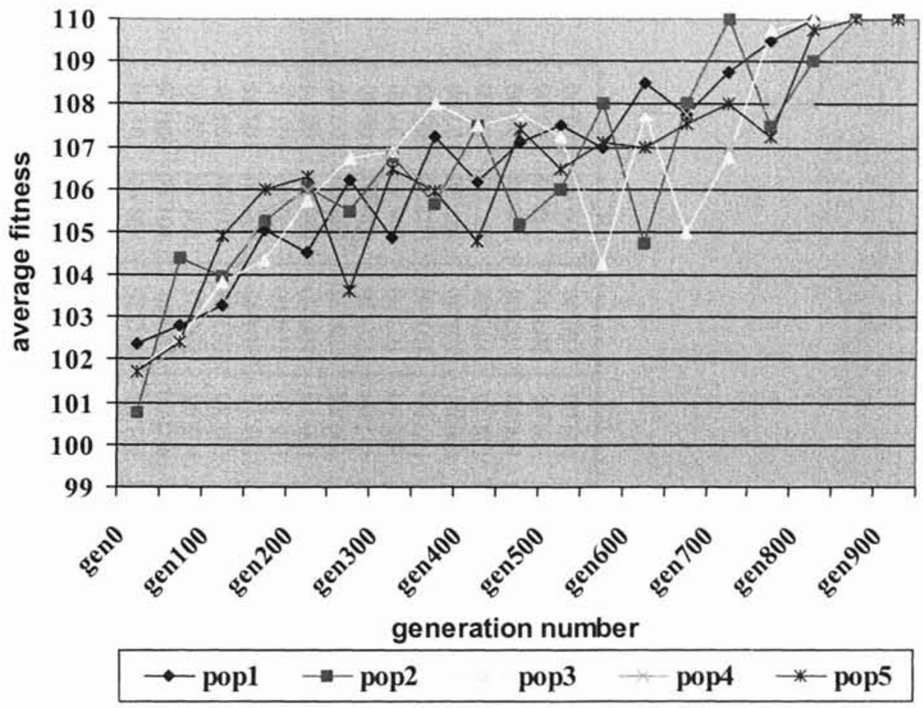


Figure 14. Average Fitness versus Generation Number
Number of Populations = 5
Population Size = 20

When several runs of the traditional GA with population size equal to 100 were performed, no permutation with fitness value higher than 108 was found. Thus, the results obtained indicate that a cgGA is capable of finding permutations with higher fitness values than those found with a traditional GA. In the case of a truly parallel cgGA, one would expect it to do this in a shorter time.

Table VIII summarizes the results obtained when the problem was run with 6 populations. Figure 15 shows the change in the average fitness value of each one of the five populations.

Table VIII. Statistics for R(3,3,3) (Number of Populations = 6, Population Size = 20)

	Pop 1	Pop 2	Pop 3	Pop 4	Pop 5	Pop 6	Pop1	Pop2	Pop3	Pop4	Pop5	Pop6
Generation	Best Fitness						Average Fitness					
0	104.00	103.00	106.00	105.00	106.00	107.00	101.35	101.80	101.70	102.25	101.90	101.50
50	103.00	106.00	107.00	106.00	105.00	105.00	102.10	102.45	103.60	106.00	101.90	100.20
100	107.00	109.00	107.00	105.00	106.00	106.00	104.15	103.40	102.95	103.25	103.50	103.50
150	107.00	109.00	109.00	109.00	107.00	106.00	100.35	102.45	103.80	102.85	104.60	104.50
200	107.00	109.00	100.00	109.00	109.00	107.00	104.70	103.50	103.10	104.95	104.00	103.35
250	107.00	109.00	100.00	109.00	107.00	109.00	103.70	105.95	105.20	104.60	106.10	103.85
300	109.00	109.00	109.00	109.00	108.00	109.00	104.70	105.65	106.10	105.30	106.00	106.75
350	109.00	109.00	109.00	109.00	109.00	109.00	106.90	106.10	106.85	104.55	106.90	107.00
400	109.00	109.00	109.00	109.00	109.00	109.00	105.30	106.10	107.00	105.20	106.80	106.80
450	109.00	109.00	109.00	109.00	109.00	109.00	106.20	105.80	106.50	107.00	106.80	107.00
500	109.00	109.00	109.00	109.00	109.00	109.00	106.80	106.70	107.00	107.00	107.00	106.20
550	109.00	109.00	109.00	109.00	109.00	109.00	106.80	107.00	106.60	106.80	107.00	107.00
600	109.00	109.00	109.00	109.00	109.00	109.00	105.80	107.60	106.50	107.00	107.00	106.60
650	109.00	109.00	109.00	109.00	109.00	109.00	109.00	107.80	108.00	109.00	106.90	108.90
700	109.00	109.00	109.00	109.00	109.00	109.00	108.80	109.00	109.00	109.00	109.00	109.00
750	109.00	109.00	109.00	109.00	109.00	109.00	109.00	109.00	109.00	109.00	109.00	109.00

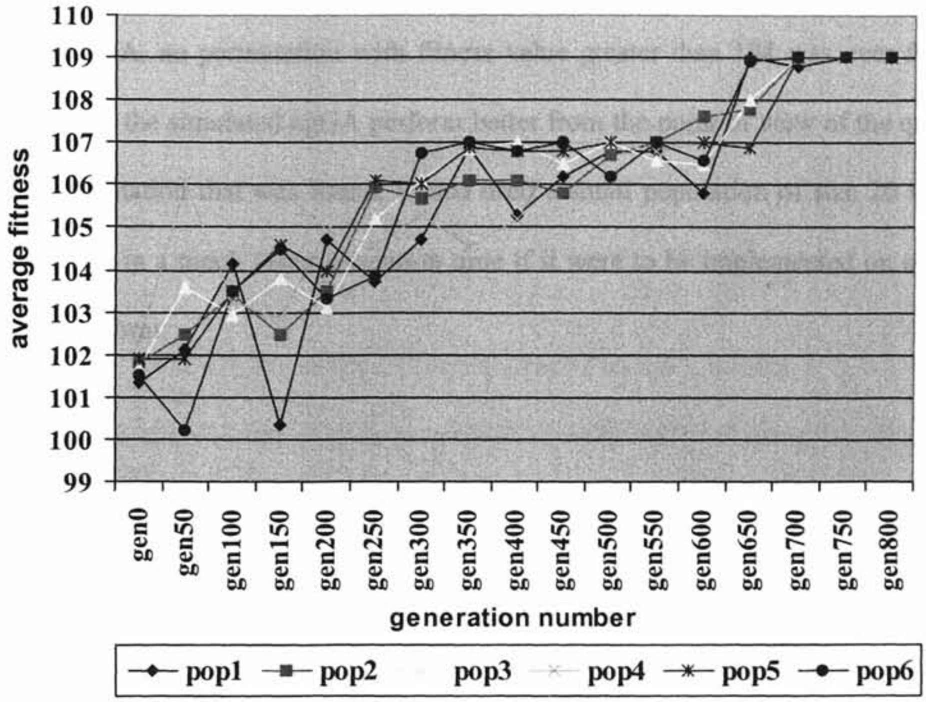


Figure 15. Average Fitness versus Generation Number
Number of Populations = 6
Population Size = 20

From the above results we can observe that increasing the number of populations in a cgGA does not necessarily result in better performance. What the optimal number of populations in a cgGA is, is a research question that is beyond the scope of this thesis.

6.2 Conclusions

According to the experimental results, increasing the number of populations in a cgGA does not help in reducing the rate of premature convergence. However, the difference between the fitness value of the permutations found with the traditional GA and with the simulated cgGA is evident. The reason for the superior performance of the cgGA is that it maintains multiple populations that evolve independently. In this manner, each population explores different parts of the search space and thus the chances of finding the global optimum increase. When R(3,3,3) was run on a traditional GA, no permutation with fitness value greater than 108 was ever found. Not only did the simulated cgGA perform better from the point of view of the quality of the permutation that was found, it also used a small population of size 20 which would result in a much faster execution time if it were to be implemented on a truly parallel platform.

CHAPTER VII

RESULTS, CONCLUSIONS, AND FUTURE WORK

The first section of this chapter presents the results achieved using the simulated cgGA for searching R(3,3,3,3) whose value is known to be between 51 and 64. Concluding remarks are made in the second section and directions for future research are presented at the end of the chapter.

7.1 Results of Run for R(3,3,3,3)

Because $51 \leq R(3,3,3,3) \leq 64$, in order to reduce the range of possible values for this Ramsey Number , we could try to increase the lower bound by one first. That is, finding a complete graph on 51 vertices with no monochromatic triangles on either of 4 different colors would prove that $52 \leq R(3,3,3,3) \leq 64$. The massive computation involved in assigning a fitness value to a complete graph on 51 vertices and the huge search space makes the goal of improving on this lower bound a very unrealistic goal for our simulated cgGA. The purpose of developing SIPAGAR was to compare the performance of a traditional GA with that of a cgGA in finding Ramsey Numbers. In this respect, this thesis has clearly shown that future attempts to find Ramsey Numbers based on a cgGA are more promising than those based on a traditional GA. Nevertheless, the simulated cgGA was run with the following parameter values:

Number of Populations: 6

Population Size: 50

Number of Colors: 4

Number of Vertices: 51

Selection Strategy: Roulette-Wheel

Crossover Rate: 0.85

Crossover Strategy: PMX

Mutation Rate: 0.05

Migration Frequency: 15

Migration Size: 5

Migrant Selection: Roulette-Wheel

Migration Topology: Ring

Elitism: True

The optimal permutation (a complete graph on 51 vertices with no monochromatic triangles in either of 4 colors) has a fitness value of 1275. A permutation with a fitness value equal to 946 was found as a result of running the simulated cgGA with the above parameter values. Figure 16 shows a snapshot of this run. No permutation with a fitness value greater than 924 was found when the traditional GA with a population size of 100 was run several times on this problem.

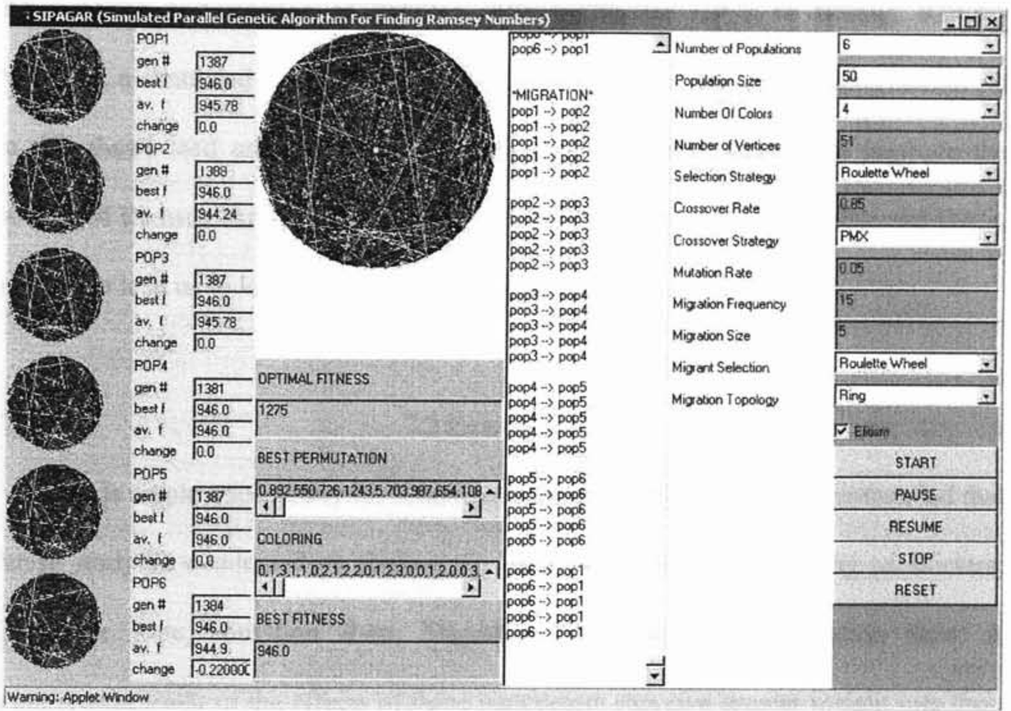


Figure 16. Snapshot of a Run on R(3,3,3,3)

7.2 Conclusions

This thesis has proposed a cgGA for solving one of the most interesting and difficult problems in combinatorics – finding Ramsey Numbers. We presented brief overviews of Ramsey Theory and Genetic Algorithms as a search technique. Parallel GA's were introduced as an extension of traditional GA's that are capable of improving the time performance and of reducing the likelihood of premature convergence. cgGA's were presented as a type of PGA that maintain a number of independent populations and allow for the occasional interchange of individuals. It was discussed how, in this manner, cgGA's increase the diversity of search paths and thus have a better chance of finding an optimal solution. In order to verify and validate the superior performance of cgGA's over traditional GA's in finding Ramsey Numbers, a simulated cgGA was developed. The results of the experiments conducted in this thesis lead us to the conclusion that cgGA-based attempts to improve the bounds of Ramsey Numbers are more promising than those based on traditional GA's, and hence lead us to look with increased confidence in these directions.

7.3 Future Work

There is ample opportunity for future work on this problem. It is recommended that future study be conducted on finding the ideal values for the following parameters: Crossover Rate, Mutation Rate, Migration Frequency, and Migration Size. A comparative study of the effects of these parameters can give greater insight into their optimal values. It would also be very interesting to experiment with different Crossover Selection, Crossover, and Migrant Selection strategies as well as with different Migration topologies. In the current implementation of SIPAGAR, there are only two choices for the Migrant Selection, Crossover, and Crossover Selection strategies. There is only a single choice for the Migration Topology. Probably the most important work that could be done in the future is to implement the cgGA proposed in this thesis on a truly parallel platform.

REFERENCES

[Cohoon et al. 91] J. P. Cohoon, S. U. Hedge, W. N. Martin, and D. S. Richards, "Distributed Genetic Algorithms for the Floorplan Design Problem," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 10, No. 4, pp. 483-492, April 1991.

[Darwin 59] Charles Darwin, *On the Origin of Species by Means of Natural Selection*, J. Murray Press, London, UK, 1859.

[De Jong 75] K. De Jong, *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*, University of Michigan Press, Ann Arbor, MI, 1975.

[Erickson 96] Martin J. Erickson, *Introduction to Combinatorics*, John Wiley & Sons, Inc., New York, NY, 1996.

[Eiben and van der Hauw 98] A. E. Eiben and J. K. van der Hauw, "Adaptive Penalties for Evolutionary Graph Coloring," *Lecture Notes in Computer Science*, pp. 95-106, Edited by: G. Goos, J. Hartmanis, and J. van Leeuwen, Springer-Verlag, Heidelberg, Germany, 1998.

[Goodman et al. 94] E. D. Goodman, Shyh-Chang Lin, and W. F. Punch III, "Coarse-Grain Parallel Genetic Algorithms: Categorization and New Approach," *Proceedings of the Sixth IEEE Symposium on Parallel and Distributed Processing*, pp. 28-37, Dallas, TX, October 1994.

[Goldberg and Richardson 87] D. E. Goldberg and J. T. Richardson, "Genetic Algorithms with Sharing for Multi-Modal Function Optimization," *Proceedings of the Second International Conference on Genetic Algorithms and Their Applications*, pp. 41-49, Cambridge, MA, July 1987.

[Goldberg et al. 92] D. E. Goldberg, K. Deb, and J. H. Clark, "Genetic Algorithms, Noise, and the Sizing of Populations," *Complex Systems*, Vol. 6, No. 4, pp. 333-362, February 1992.

[Graham and Spencer 90] Ronald L. Graham and Joel H. Spencer, "Ramsey Theory," *Scientific American*, Vol 263, No. 1, pp. 112-117, July 1990.

[Holland 75] John Holland, *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor, MI, 1975.

[Mouldin 84] M. Mouldin, "Maintaining Diversity in Genetic Search," *Proceedings of the Fourth National Conference on Artificial Intelligence*, pp. 247-250, Austin, TX, August 1984.

[Papadopoulos 94] Constantinos V. Papadopoulos, "On the Parallel Execution of Combinatorial Heuristics," *Proceedings of the First International Conference on Massively Parallel Computing Systems*, pp. 423-427, Los Alamitos, CA, May 1994.

[Poon and Carter 95] P. W. Poon and J. N. Carter, "Genetic Algorithm Crossover Operators for Ordering Applications." *Computers and Operations Research*, Vol. 22, No. 1, pp. 135-147, March 1995.

[Radziszowski 93] Stanislaw P. Radziszowski, "Small Ramsey Numbers," Technical Report RIT-TR-93-009, Department of Computer Science, Rochester Institute of Technology, Rochester, NY, 1993.

[Rebaudergo and Reorda 92] M. Rebaudergo and M. Sonza Reorda, "An Experimental Analysis of the Effects of Migration in Parallel Genetic Algorithms," *Proceedings of the Euromicro Workshop on Parallel and Distributed Processing*, pp. 232-238, Athens, Greece, January 1992.

[Stracuzzi 98] David J. Stracuzzi, "Some Methods for the Parallelization of Genetic Algorithms," http://ml-www.cs.umass.edu/~stracudj/genetic/dga.html, creation date: May 1998 (access date: January 2000).

[Ugoluk 97] Gokturk Ugoluk, "A Method for Chromosome Handling of r-Permutations of n-Element Sets in Genetic Algorithms," *Proceedings of the IEEE International Conference on Evolutionary Computation*, pp. 55-58, Indianapolis, IN, April 1997.

APPENDICES

APPENDIX A

GLOSSARY

Allopatric Speciation	The rapid evolution of new species after a small set of members of a species becomes segregated into a new environment.
Applet	A program written in JAVA that can be included in an HTML page.
Asynchronous Processes	Processes that do not block on an input/output operation waiting for the corresponding output/input reply from other process. A queue or buffer is used instead to store messages.
CF	Crowding Factor. Number of individuals in a subpopulation in De Jong's crowding scheme.
cgGA	Coarse-Grain Genetic Algorithm. A parallel Genetic Algorithm based on the theory of punctuated equilibria.
Chromosome	A sequence of genes (usually represented as a string of bits) determining an individual's genotype.
Complete Graph	A graph in which every two distinct vertices are joined by an edge.
Crossover	Sexual recombination. It is the genetic operation that allows new individuals to be created. It allows new points in the search space to be tested.
DECODER	A function used to decode elements of a permutation-based chromosome.
Dynamic Connection Scheme	A scheme in which the network topology may change during execution.

Evaluation Function	A function that evaluates and assigns a fitness value to an individual.
fgGA	Fine-Grain Genetic Algorithm. A Genetic Algorithm composed of small overlapping subpopulations. Individuals belonging to more than one subpopulation allow for the interchange of information between subpopulations.
Fitness Value	Value assigned to an individual according to its aptitude in solving a given problem.
GA	Genetic Algorithm.
GA-Hard Problem	Problems that are not easily solved by a Standard Genetic Algorithm.
Gene	Specific characteristic or attribute that is encoded in a chromosome.
Genetic Algorithm	A highly parallel mathematical algorithm that transforms a set of individual mathematical objects, each with an associated fitness value, into a new population using operations patterned after the Darwinian principle of reproduction and survival of the fittest.
Genotype	Observable characteristics of an individual (a solution).
Global Optimum	An optimal solution to a given problem.
GUI	Graphical User Interface.
Hamming's Distance	Minimum number of bit positions by which codewords for a particular code differ. Number of different genes between two chromosomes.
Heterogeneous Island GA	A cgGA in which processes may have different parameters.
Homogeneous Island GA	A cgGA in which every processor uses the same parameters.
Hypergraph	A graph whose hyperedges connect one or more vertices.

Islands Model	A parallel genetic algorithm in which the total population is divided into several subpopulations or islands and migration is performed at determined time intervals.
Local Optimum	A sub-optimal solution to a given problem.
Master Process	A process that maintains a population and performs classical genetic operations while assigning computational tasks to the slave processes.
mdpGA	Massively Distributed Parallel Genetic Algorithm. A Genetic Algorithm in which every subpopulation is assigned to a processors. Subpopulations are small.
mgGA	Micro-Grain Genetic Algorithm. A Genetic Algorithm that maintains a single population in a master process. The master process performs classical genetic operators while assigning the task of fitness evaluations to slave processes.
Migration	Process of interchanging individuals between different populations.
Migration Frequency	Determines the number of generations between two migrations.
Migrant Selection	Determines which immigrants are chosen within the source population.
Migration Size	Determines the number of individuals composing each migration.
Monochromatic K_n	A complete graph of order n in which all the edges have the same color.
Mutation	An operation that is usually used in a conventional genetic algorithm and consists of randomly changing the bits of a fixed-length string to introduce genetic diversity.
Mutation Rate	The frequency at which mutation is performed.
NP-Complete Problem	A problem B is NP-hard if solving it in polynomial time would make it possible to solve all problems in class NP in polynomial time. A problem is NP complete when it is both NP hard and it is in NP.

Order-Based Representation	A representation scheme in which each chromosome is a permutation of some given problem parameters.
Order of a Graph	Let G = (V, E) be a graph with vertex set V and edge set E, the number of vertices of G is called the order of G.
PGA	Parallel Genetic Algorithm. A Genetic Algorithm that maintains multiple, independent populations that each focus on a different area of a problem.
Phenotype	Genetic structure of an individual (a bit string).
Population	A set of individual mathematical objects (typically fixed-length character strings patterned after chromosome strings).
Reproduction	The creation of two children (chromosomes) by using two parents of the previous generation.
SIPAGAR	SImulated PArallel Genetic Algorithm for finding Ramsey numbers.
Slave Process	A process performing computational tasks for a master process.
Stasis	Stability or lack of change.
Static Connection Scheme	A scheme in which the connections between different processors are established at the beginning and not modified during execution.
Synchronous Processes	A process blocks on an input/output operation until the corresponding process replies with an output/input operation.
Termination Criteria	Criteria that cause a genetic algorithm to stop performing the operations on each generation of individuals to produce new generations.
Thread	A parallely executable sequence of instructions in a program
T-uniform Hypergraph	A hypergraph all of whose hyperedges connect t vertices.

APPENDIX B

TRADEMARK INFORMATION

JAVA	A registered trademark of Sun Microsystems, Inc.

APPENDIX C

KNOWN BOUNDS ON RAMSEY NUMBERS

Ramsey Numbers quantify some of the general existential theorems in Ramsey Theory. A Ramsey number is defined as the smallest integer N such that if the edges of the complete graph on R(a_1, a_2, ..., a_c) vertices are partitioned into c color classes, then there exists a complete graph on any one of a_i for 1 [i [c vertices all of whose edges are of color a_i. Ramsey numbers are very difficult to find, only a few are known.

The following bounds on classical and multicolor Ramsey numbers were found in a technical report by Radziszowsky [Radziszowsky 93].

Two-color classical Ramsey numbers

Table I. Known values and lower/upper bounds for two color Ramsey numbers R(k,l) = R(k,l;2) [Radziszowski 93].

l / k	3	4	5	6	7	8	9	10	11	12	13	14	15
3	6	9	14	18	23	28	36	40 43	46 51	52 59	59 69	66 78	73 88
4		18	25	35 41	49 61	55 84	69 115	80 149	96 191	128 238	131 291	136 349	145 417
5			43 49	58 87	80 143	95 216	121 316	141 442	153	181	193	221	242
6				102 165	109 298	122 495	153 780	167 1171	203	230	242	284	374
7					205 540	1031	1713	2826		312			
8						282 1870	3583	6090					
9							565 6588	12677					
10								798 23581					

Table II. Known lower bounds for higher two color Ramsey numbers R(k,l) [Radziszowski93]

l / k	15	16	17	18	19	20	21	22	23
3	73	79	92	98	106	109	122	125	136
4	145		164	182	198	230	242	282	
5	242		282		338		374	422	434
6	374	434	548	614	710	878			
7			578	618	758				
8	618		678	740	860	948			

What follows are some known bounds for Multicolor Ramsey Numbers:

The only known value for a multicolor Ramsey number is $R(3,3,3) = 17$

$$51 \leq R(3,3,3,3) \leq 64$$
$$162 \leq R(3,3,3,3,3) \leq 317$$
$$500 \leq R(3,3,3,3,3,3) \leq 1898$$

$$128 \leq R(4,4,4) \leq 236$$
$$458 \leq R(4,4,4,4)$$
$$942 \leq R(4,4,4,4,4)$$

$$385 \leq R(5,5,5)$$
$$1833 \leq R(5,5,5,5)$$
$$4711 \leq R(5,5,5,5,5)$$

$$1070 \leq R(6,6,6)$$
$$3433 \leq R(6,6,6,6)$$
$$3211 \leq R(7,7,7)$$
$$12841 \leq R(7,7,7,7)$$

$$30 \leq R(3,3,4) \leq 31$$
$$45 \leq R(3,3,5) \leq 57$$
$$60 \leq R(3,3,6)$$
$$72 \leq R(3,3,7)$$
$$110 \leq R(3,3,9)$$
$$141 \leq R(3,3,11)$$
$$55 \leq R(3,4,4) \leq 79$$
$$80 \leq R(3,4,5) \leq 161$$
$$91 \leq R(3,3,3,4) \leq 155$$
$$144 \leq R(3,3,4,4)$$

APPENDIX D

PERMUTATION-RESPECTING OPERATORS

Ordinary crossover and mutation operators cause problems for order-based representations. The reason for this is that offspring generated by means of ordinary operators may not be valid solutions for the problem being solved anymore. The following are some permutation-respecting operators that can be used with an order-based representation.

CROSSOVER

Uniform Order Crossover [Poon and Carter 95]

The offspring chromosome is initially empty. At each position, the first gene is selected at random from one of the two parent chromosomes and inserted into the offspring. The gene is then deleted from both parents.

Parent 1	3 4 5 1 7 6 2	Offspring	3 7 4 5 1 6 2
Parent 2	7 5 4 6 3 1 2	Random parental sequence: 1 2 1 2 1 1 2	

Partially Matched Crossover [Poon and Carter 95]

A matching section consisting of two crossover points is randomly chosen. Elements of the matching sections that occur in the other parent are deleted. The remaining part of each parent is combined with the matching section of the other parent. The matching section maintains its original position in the new chromosome.

Parent 1	3 4 5 1 7 6 2	_ _ 5 1 7 _ 2	Offspring1	5 1 4 6 3 7 2
Parent 2	7 5 4 6 3 1 2	3 4 _ _ _ 6 2	Offspring2	3 4 5 1 7 6 2

MUTATION

<u>Swap Mutation</u> [Poon and Carter 95]

Two randomly selected genes in a chromosome are swapped.

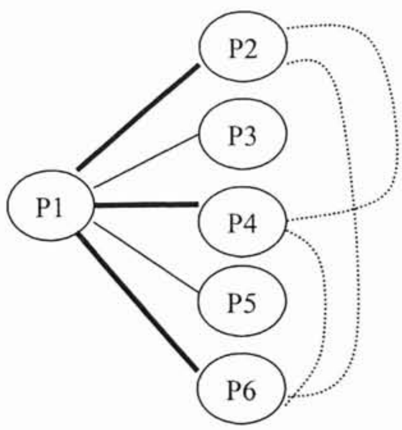
Individual	3 4 5 1 7 6 2	Mutated individual	3 6 5 1 7 4 2

APPENDIX E

PROOF OF R(3,3) = 6

The party puzzle is a classical problem used to introduce Ramsey Theory. What is the minimum number of guests that must be invited to a party so that either a group of at least three people will know one another or at least three guests will not mutually know each other? The answer to this problem, which equals 6, is called the Ramsey number R(3,3).

To show this, fix one person (or one point in the graph), say P1, and consider his or her relationship (or color of the edge) to P2, P3, P4, P5, and P6. By the pigeonhole principle, P1 must either know at least 3 of the other 5 people, or not know at least 3 of them. Suppose P1 knows P2, P4, and P6 as represented by the dark edges in the following figure.



If any pair of P2, P4, and P6 know each other, then at least one of the edges (P2, P4), (P2, P6), or (P4, P6) will be drawn with a dark edge, thus obtaining at least one monochromatic triangle (or 3 people who mutually know each other). If no pair of P2, P4, and P6 know each other, those 3 mutually do not know each other, thus P2, P4, and P6 are the vertices of a monochromatic triangle as well.

APPENDIX F

RESULTS OF EXPERIMENTS

This appendix contains statistics reported every 10 generations for all the runs performed as mentioned in Chapter VI. The statistics in this appendix are indexed by the corresponding Table number in Chapter VI (in the interest of brevity, only the statistics for three tables are listed here).

Statistics for Table 2

Population size = 40

```
generation #: 0   best: 106.0    average: 101.56   change: 101.56
generation #: 10  best: 107.0    average: 101.6    change: 0.03
generation #: 20  best: 107.0    average: 101.28   change: 0.01
generation #: 30  best: 107.0    average: 101.49   change: 0.14
generation #: 40  best: 107.0    average: 101.48   change: 0.03
generation #: 50  best: 107.0    average: 101.96   change: -0.05
generation #: 60  best: 107.0    average: 102.36   change: 0.31
generation #: 70  best: 107.0    average: 102.39   change: 0.03
generation #: 80  best: 107.0    average: 102.93   change: 0.0
generation #: 90  best: 107.0    average: 103.22   change: -0.01
generation #: 100 best: 107.0    average: 103.29   change: -0.04
generation #: 110 best: 107.0    average: 103.45   change: 0.01
generation #: 120 best: 107.0    average: 103.32   change: 0.05
generation #: 130 best: 107.0    average: 103.09   change: -0.04
generation #: 140 best: 107.0    average: 103.15   change: 0.01
generation #: 150 best: 107.0    average: 103.14   change: -0.15
generation #: 160 best: 107.0    average: 102.07   change: 0.02
generation #: 170 best: 107.0    average: 102.03   change: -0.10
generation #: 180 best: 107.0    average: 102.5    change: 0.01
generation #: 190 best: 107.0    average: 102.13   change: -0.04
generation #:  20 best: 107.0    average: 102.49   change: -0.10
generation #: 210 best: 107.0    average: 103.14   change: 0.03
generation #: 220 best: 107.0    average: 102.91   change: 0.010
generation #: 230 best: 107.0    average: 102.92   change: -0.17
generation #: 240 best: 107.0    average: 103.25   change: -0.04
generation #: 250 best: 107.0    average: 103.08   change: 0.14
generation #: 260 best: 107.0    average: 103.04   change: 0.05
generation #: 270 best: 107.0    average: 103.98   change: 0.07
generation #: 280 best: 107.0    average: 104.06   change: 0.0
generation #: 290 best: 107.0    average: 103.54   change: 0.09
```

```
generation #: 300 best: 107.0        average: 103.66     change: -0.07
generation #: 310 best: 107.0        average: 103.7      change: 0.07
generation #: 320 best: 107.0        average: 103.21     change: -0.18
generation #: 330 best: 107.0        average: 103.78     change: 0.12
generation #: 340 best: 107.0        average: 103.39     change: 0.03
generation #: 350 best: 107.0        average: 103.43     change: -0.09
generation #: 360 best: 107.0        average: 103.45     change: 0.06
generation #: 370 best: 107.0        average: 103.46     change: 0.04
generation #: 380 best: 107.0        average: 103.79     change: 0.13
generation #: 390 best: 107.0        average: 103.59     change: -0.01
generation #: 400 best: 107.0        average: 103.94     change: 0.04
generation #: 410 best: 107.0        average: 103.5      change: 0.0
generation #: 420 best: 107.0        average: 103.96     change: -0.08
generation #: 430 best: 107.0        average: 104.4      change: -0.10
generation #: 440 best: 107.0        average: 104.02     change: 0.0
generation #: 450 best: 107.0        average: 104.05     change: 0.03
generation #: 460 best: 107.0        average: 103.82     change: 0.03
generation #: 470 best: 107.0        average: 103.8      change: 0.00
generation #: 480 best: 107.0        average: 103.76     change: 0.06
generation #: 490 best: 107.0        average: 104.33     change: 0.01
generation #: 500 best: 107.0        average: 104.03     change: 0.10
generation #: 510 best: 107.0        average: 103.74     change: 0.23
generation #: 520 best: 107.0        average: 103.58     change: 0.04
generation #: 530 best: 107.0        average: 103.7      change: -0.11
generation #: 540 best: 107.0        average: 103.51     change: -0.03
generation #: 550 best: 107.0        average: 103.77     change: 0.03
generation #: 560 best: 107.0        average: 104.11     change: 0.0
generation #: 570 best: 107.0        average: 103.92     change: -0.15
generation #: 580 best: 107.0        average: 104.08     change: 0.01
generation #: 590 best: 107.0        average: 103.99     change: -0.02
generation #: 600 best: 107.0        average: 104.4      change: -0.03
generation #: 610 best: 107.0        average: 104.61     change: -0.20
generation #: 620 best: 107.0        average: 104.39     change: -0.07
generation #: 630 best: 107.0        average: 105.32     change: 0.07
generation #: 640 best: 107.0        average: 105.72     change: 0.23
generation #: 650 best: 107.0        average: 105.5      change: -0.15
generation #: 660 best: 107.0        average: 105.58     change: 0.0
generation #: 670 best: 107.0        average: 105.67     change: -0.03
generation #: 680 best: 107.0        average: 105.96     change: 0.0
generation #: 690 best: 107.0        average: 106.41     change: 0.03
generation #: 700 best: 107.0        average: 106.5      change: 0.03
generation #: 710 best: 107.0        average: 106.31     change: 0.0
generation #: 720 best: 107.0        average: 106.48     change: 0.03
generation #: 730 best: 107.0        average: 106.37     change: -0.01
generation #: 740 best: 107.0        average: 106.47     change: 0.04
generation #: 750 best: 107.0        average: 106.11     change: 0.10
generation #: 760 best: 107.0        average: 106.11     change: 0.03
generation #: 770 best: 107.0        average: 106.31     change: -0.04
generation #: 780 best: 107.0        average: 106.47     change: 0.0
generation #: 790 best: 107.0        average: 106.51     change: 0.03
generation #: 800 best: 107.0        average: 106.83     change: -0.03
generation #: 810 best: 107.0        average: 106.97     change: 0.03
generation #: 820 best: 107.0        average: 106.97     change: 0.0
generation #: 830 best: 107.0        average: 106.97     change: 0.03
generation #: 840 best: 107.0        average: 106.97     change: 0.0
generation #: 850 best: 107.0        average: 107.0      change: 0.0
generation #:>850 best: 107.0        average: 107.0      change: 0.0
```

Population Size = 60

```
generation#: 0    best: 107.0        average: 101.51     change: 101.51
generation#: 10   best: 107.0        average: 102.32     change: 0.13
generation#: 20   best: 107.0        average: 102.08     change: 0.11
generation#: 30   best: 107.0        average: 102.26     change: 0.10
generation#: 40   best: 107.0        average: 101.97     change: -0.06
```

```
generation#: 50  best: 107.0        average: 101.73   change: 0.13
generation#: 60  best: 107.0        average: 101.53   change: 0.11
generation#: 70  best: 107.0        average: 102.05   change: 0.29
generation#: 80  best: 107.0        average: 102.47   change: 0.17
generation#: 90  best: 107.0        average: 102.66   change: 0.18
generation#: 100 best: 107.0        average: 102.41   change: -0.23
generation#: 110 best: 107.0        average: 103.57   change: 0.08
generation#: 120 best: 107.0        average: 104.0    change: -0.03
generation#: 130 best: 107.0        average: 103.27   change: -0.08
generation#: 140 best: 107.0        average: 103.58   change: 0.23
generation#: 150 best: 107.0        average: 103.32   change: -0.02
generation#: 160 best: 107.0        average: 104.01   change: 0.08
generation#: 170 best: 107.0        average: 104.33   change: -0.09
generation#: 180 best: 107.0        average: 104.76   change: 0.03
generation#: 190 best: 107.0        average: 105.35   change: 0.08
generation#: 200 best: 107.0        average: 105.32   change: -0.12
generation#: 210 best: 107.0        average: 105.06   change: 0.08
generation#: 220 best: 107.0        average: 104.86   change: 0.02
generation#: 230 best: 107.0        average: 104.77   change: -0.01
generation#: 240 best: 107.0        average: 104.83   change: 0.11
generation#: 250 best: 107.0        average: 104.53   change: 0.12
generation#: 260 best: 107.0        average: 104.53   change: 0.12
generation#: 270 best: 107.0        average: 104.92   change: 0.03
generation#: 280 best: 107.0        average: 104.85   change: 0.08
generation#: 290 best: 107.0        average: 104.65   change: -0.07
generation#: 300 best: 107.0        average: 104.56   change: 0.16
generation#: 310 best: 107.0        average: 104.4    change: -0.08
generation#: 320 best: 107.0        average: 104.76   change: 0.02
generation#: 330 best: 107.0        average: 104.77   change: 0.01
generation#: 340 best: 107.0        average: 104.41   change: 0.01
generation#: 350 best: 107.0        average: 104.1    change: -0.03
generation#: 360 best: 107.0        average: 104.28   change: 0.04
generation#: 370 best: 107.0        average: 104.48   change: 0.11
generation#: 380 best: 107.0        average: 104.57   change: -0.04
generation#: 390 best: 107.0        average: 104.98   change: 0.08
generation#: 400 best: 107.0        average: 104.4    change: -0.14
generation#: 410 best: 107.0        average: 104.63   change: 0.0
generation#: 420 best: 107.0        average: 104.85   change: -0.05
generation#: 430 best: 107.0        average: 105.0    change: 0.09
generation#: 440 best: 107.0        average: 104.7    change: 0.0
generation#: 450 best: 107.0        average: 104.45   change: -0.04
generation#: 460 best: 107.0        average: 104.55   change: 0.0
generation#: 470 best: 107.0        average: 104.25   change: -0.15
generation#: 480 best: 107.0        average: 104.1    change: 0.04
generation#: 490 best: 107.0        average: 104.3    change: 0.14
generation#: 500 best: 107.0        average: 104.5    change: 0.04
generation#: 510 best: 107.0        average: 104.6    change: -0.05
generation#: 520 best: 107.0        average: 104.4    change: 0.10
generation#: 530 best: 107.0        average: 104.55   change: 0.09
generation#: 540 best: 107.0        average: 104.6    change: 0.25
generation#: 550 best: 107.0        average: 104.6    change: 0.09
generation#: 560 best: 107.0        average: 104.55   change: 0.04
generation#: 570 best: 107.0        average: 104.45   change: 0.0
generation#: 580 best: 107.0        average: 104.85   change: 0.0
generation#: 590 best: 107.0        average: 104.95   change: 0.0
generation#: 600 best: 107.0        average: 104.95   change: 0.0
generation#: 610 best: 107.0        average: 105.15   change: 0.0
generation#: 620 best: 107.0        average: 105.0    change: -0.09
generation#: 630 best: 107.0        average: 104.65   change: -0.04
generation#: 640 best: 107.0        average: 104.6    change: 0.04
generation#: 650 best: 107.0        average: 104.95   change: 0.20
generation#: 660 best: 107.0        average: 104.7    change: 0.04
generation#: 670 best: 107.0        average: 105.05   change: 0.14
generation#: 680 best: 107.0        average: 105.15   change: -0.04
generation#: 690 best: 107.0        average: 105.85   change: 0.09
generation#: 700 best: 107.0        average: 106.25   change: 0.0
```

```
generation#: 710 best: 107.0        average: 106.35    change: 0.0
generation#: 720 best: 107.0        average: 106.55    change: 0.0
generation#: 730 best: 107.0        average: 106.75    change: 0.0
generation#: 740 best: 107.0        average: 106.75    change: 0.04
generation#: 750 best: 107.0        average: 106.8     change: 0.0
generation#: 760 best: 107.0        average: 106.65    change: 0.0
generation#: 770 best: 107.0        average: 106.75    change: 0.15
generation#: 780 best: 107.0        average: 106.45    change: 0.0
generation#: 790 best: 107.0        average: 106.45    change: -0.04
generation#: 800 best: 107.0        average: 106.5     change: 0.04
generation#: 810 best: 107.0        average: 106.5     change: 0.0
generation#: 820 best: 107.0        average: 106.3     change: 0.0
generation#: 830 best: 107.0        average: 105.9     change: -0.14
generation#: 840 best: 107.0        average: 105.6     change: -0.10
generation#: 850 best: 107.0        average: 105.35    change: 0.09
generation#: 860 best: 107.0        average: 105.65    change: 0.0
generation#: 870 best: 107.0        average: 105.55    change: 0.14
generation#: 880 best: 107.0        average: 105.6     change: 0.04
generation#: 890 best: 107.0        average: 105.35    change: -0.05
generation#: 900 best: 107.0        average: 104.8     change: -0.15
generation#: 910 best: 107.0        average: 105.15    change: -0.04
generation#: 920 best: 107.0        average: 104.85    change: 0.0
generation#: 930 best: 107.0        average: 105.0     change: 0.0
generation#: 940 best: 107.0        average: 105.4     change: 0.0
generation#: 950 best: 107.0        average: 105.85    change: 0.04
generation#: 960 best: 107.0        average: 105.8     change: 0.09
generation#: 970 best: 107.0        average: 105.95    change: 0.15
generation#: 980 best: 107.0        average: 105.75    change: -0.09
generation#: 990 best: 107.0        average: 106.0     change: 0.09
generation#: 1000 best: 107.0       average: 105.85    change: 0.0
generation#: 1010 best: 107.0       average: 106.35    change: 0.0
generation#: 1020 best: 107.0       average: 106.05    change: -0.10
generation#: 1030 best: 107.0       average: 106.35    change: -0.05
generation#: 1040 best: 107.0       average: 106.35    change: 0.04
generation#: 1050 best: 107.0       average: 106.4     change: -0.04
generation#: 1060 best: 107.0       average: 106.7     change: -0.04
generation#: 1070 best: 107.0       average: 106.65    change: -0.04
generation#: 1080 best: 107.0       average: 106.5     change: 0.0
generation#: 1090 best: 107.0       average: 106.65    change: -0.04
generation#: 1100 best: 107.0       average: 106.9     change: 0.10
generation#: 1110 best: 107.0       average: 106.95    change: 0.04
generation#: 1120 best: 107.0       average: 106.95    change: 0.0
generation#: 1130 best: 107.0       average: 107.0     change: 0.0
generation#:>1130 best: 107.0       average: 107.0     change: 0.0
```

Population Size = 80

```
generation#:: 0   best: 106.0       average: 101.75 change: 101.75
generation#:: 10  best: 107.0       average: 102.11 change: 0.16
generation#:: 20  best: 107.0       average: 102.41 change: 0.12
generation#:: 30  best: 107.0       average: 102.55 change: 0.01
generation#:: 40  best: 107.0       average: 102.4  change: -0.04
generation#:: 50  best: 107.0       average: 102.25 change: 0.02
generation#:: 60  best: 107.0       average: 102.13 change: -0.01
generation#:: 70  best: 107.0       average: 102.27 change: 0.07
generation#:: 80  best: 107.0       average: 102.31 change: 0.04
generation#:: 90  best: 107.0       average: 102.2  change: 0.06
generation#:: 100 best: 107.0       average: 101.9  change: -0.06
generation#:: 110 best: 107.0       average: 101.75 change: -0.05
generation#:: 120 best: 107.0       average: 102.06 change: -0.02
generation#:: 130 best: 107.0       average: 101.96 change: 0.0
generation#:: 140 best: 107.0       average: 101.71 change: 0.06
generation#:: 150 best: 107.0       average: 101.9  change: -0.16
generation#:: 160 best: 107.0       average: 101.46 change: -0.02
generation#:: 170 best: 107.0       average: 101.4  change: 0.02
```

```
generation#:: 180 best: 107.0        average: 101.7  change: -0.06
generation#:: 190 best: 107.0        average: 101.97 change: 0.04
generation#:: 200 best: 107.0        average: 102.27 change: 0.07
generation#:: 210 best: 107.0        average: 102.95 change: 0.16
generation#:: 220 best: 107.0        average: 103.17 change: 0.04
generation#:: 230 best: 107.0        average: 103.11 change: 0.02
generation#:: 240 best: 107.0        average: 102.93 change: -0.01
generation#:: 250 best: 107.0        average: 102.70 change: -0.12
generation#:: 260 best: 107.0        average: 102.92 change: 0.09
generation#:: 270 best: 107.0        average: 103.18 change: 0.06
generation#:: 280 best: 107.0        average: 103.08 change: 0.06
generation#:: 290 best: 107.0        average: 102.77 change: 0.06
generation#:: 300 best: 107.0        average: 102.23 change: -0.06
generation#:: 310 best: 107.0        average: 101.42 change: -0.07
generation#:: 320 best: 107.0        average: 101.27 change: 0.06
generation#:: 330 best: 107.0        average: 101.23 change: 0.02
generation#:: 340 best: 107.0        average: 101.47 change: 0.0
generation#:: 350 best: 107.0        average: 101.93 change: 0.02
generation#:: 360 best: 107.0        average: 102.25 change: 0.03
generation#:: 370 best: 107.0        average: 102.33 change: -0.09
generation#:: 380 best: 107.0        average: 102.65 change: -0.03
generation#:: 390 best: 107.0        average: 102.92 change: 0.02
generation#:: 400 best: 107.0        average: 103.18 change: 0.06
generation#:: 410 best: 107.0        average: 103.90 change: 0.0
generation#:: 420 best: 107.0        average: 104.17 change: -0.08
generation#:: 430 best: 107.0        average: 104.06 change: -0.13
generation#:: 440 best: 107.0        average: 104.68 change: 0.03
generation#:: 450 best: 107.0        average: 104.48 change: 0.11
generation#:: 460 best: 107.0        average: 104.23 change: 0.14
generation#:: 470 best: 107.0        average: 104.31 change: -0.02
generation#:: 480 best: 107.0        average: 104.01 change: -0.17
generation#:: 490 best: 107.0        average: 104.23 change: 0.08
generation#:: 500 best: 107.0        average: 104.00 change: 0.12
generation#:: 510 best: 107.0        average: 104.48 change: -0.17
generation#:: 520 best: 107.0        average: 104.95 change: 0.17
generation#:: 530 best: 107.0        average: 104.98 change: 0.0
generation#:: 540 best: 107.0        average: 104.40 change: -0.06
generation#:: 550 best: 107.0        average: 104.30 change: -0.36
generation#:: 560 best: 107.0        average: 104.71 change: 0.15
generation#:: 570 best: 107.0        average: 105.16 change: 0.0
generation#:: 580 best: 107.0        average: 104.70 change: -0.27
generation#:: 590 best: 107.0        average: 104.52 change: -0.14
generation#:: 600 best: 107.0        average: 104.50 change: 0.12
generation#:: 610 best: 107.0        average: 104.55 change: 0.18
generation#:: 620 best: 107.0        average: 104.66 change: 0.41
generation#:: 630 best: 107.0        average: 104.51 change: 0.04
generation#:: 640 best: 107.0        average: 104.72 change: 0.0
generation#:: 650 best: 107.0        average: 104.60 change: -0.12
generation#:: 660 best: 107.0        average: 104.13 change: 0.08
generation#:: 670 best: 107.0        average: 103.83 change: 0.02
generation#:: 680 best: 107.0        average: 104.10 change: -0.06
generation#:: 690 best: 107.0        average: 104.52 change: 0.0
generation#:: 700 best: 107.0        average: 104.63 change: 0.06
generation#:: 710 best: 107.0        average: 105.02 change: 0.15
generation#:: 720 best: 107.0        average: 104.88 change: 0.03
generation#:: 730 best: 107.0        average: 104.71 change: 0.01
generation#:: 740 best: 107.0        average: 104.37 change: 0.06
generation#:: 750 best: 107.0        average: 104.72 change: 0.31
generation#:: 760 best: 107.0        average: 104.83 change: -0.12
generation#:: 770 best: 107.0        average: 104.37 change: 0.0
generation#:: 780 best: 107.0        average: 104.03 change: -0.06
generation#:: 790 best: 107.0        average: 103.96 change: -0.06
generation#:: 800 best: 107.0        average: 103.86 change: -0.06
generation#:: 810 best: 107.0        average: 103.83 change: -0.12
generation#:: 820 best: 107.0        average: 104.20 change: -0.06
generation#:: 830 best: 107.0        average: 103.76 change: -0.02
```

```
generation#:: 840 best: 107.0       average: 103.70 change: 0.01
generation#:: 850 best: 107.0       average: 103.22 change: 0.09
generation#:: 860 best: 107.0       average: 103.43 change: -0.02
generation#:: 870 best: 107.0       average: 104.26 change: 0.02
generation#:: 880 best: 107.0       average: 105.35 change: 0.12
generation#:: 890 best: 107.0       average: 105.66 change: 0.0
generation#:: 900 best: 107.0       average: 106.16 change: 0.0
generation#:: 910 best: 107.0       average: 106.35 change: 0.0
generation#:: 920 best: 107.0       average: 106.66 change: 0.0
generation#:: 930 best: 107.0       average: 106.48 change: -0.06
generation#:: 940 best: 107.0       average: 106.25 change: 0.12
generation#:: 950 best: 107.0       average: 106.66 change: 0.08
generation#:: 960 best: 107.0       average: 106.81 change: 0.06
generation#:: 970 best: 107.0       average: 106.81 change: 0.0
generation#:: 980 best: 107.0       average: 106.87 change: 0.06
generation#:: 990 best: 107.0       average: 106.75 change: 0.0
generation#:: 1000 best: 107.0      average: 106.81 change: 0.0
generation#:: 1010 best: 107.0      average: 106.81 change: 0.0
generation#:: 1020 best: 107.0      average: 106.87 change: 0.0
generation#:: 1030 best: 107.0      average: 106.81 change: -0.06
generation#:: 1040 best: 107.0      average: 106.62 change: -0.06
generation#:: 1050 best: 107.0      average: 106.62 change: 0.0
generation#:: 1060 best: 107.0      average: 106.75 change: 0.0
generation#:: 1070 best: 107.0      average: 106.75 change: 0.06
generation#:: 1080 best: 107.0      average: 106.81 change: 0.0
generation#:: 1090 best: 107.0      average: 106.56 change: -0.06
generation#:: 1100 best: 107.0      average: 106.75 change: 0.12
generation#:: 1110 best: 107.0      average: 106.37 change: 0.0
generation#:: 1120 best: 107.0      average: 106.62 change: 0.0
generation#:: 1130 best: 107.0      average: 106.81 change: 0.0
generation#:: 1140 best: 107.0      average: 106.87 change: 0.0
generation#:: 1150 best: 107.0      average: 106.93 change: 0.0
generation#:: 1160 best: 107.0      average: 107.00 change: 0.0
generation#::>1160 best: 107.0      average: 107.00 change: 0.0
```

Population Size = 100

```
generation#: 0     best: 106.0      average: 101.75 change: 101.75
generation#: 10    best: 108.0      average: 101.46 change: -0.19
generation#: 20    best: 108.0      average: 102.06 change: 0.15
generation#: 30    best: 108.0      average: 102.34 change: 0.24
generation#: 40    best: 108.0      average: 102.46 change: 0.03
generation#: 50    best: 108.0      average: 102.57 change: 0.12
generation#: 60    best: 108.0      average: 102.54 change: -0.04
generation#: 70    best: 108.0      average: 102.86 change: -0.18
generation#: 80    best: 108.0      average: 103.04 change: 0.06
generation#: 90    best: 108.0      average: 103.53 change: 0.06
generation#: 100   best: 108.0      average: 103.19 change: -0.10
generation#: 110   best: 108.0      average: 103.12 change: -0.12
generation#: 120   best: 108.0      average: 103.17 change: 0.07
generation#: 130   best: 108.0      average: 103.16 change: -0.04
generation#: 140   best: 108.0      average: 102.84 change: -0.14
generation#: 150   best: 108.0      average: 102.05 change: -0.17
generation#: 160   best: 108.0      average: 102.38 change: 0.06
generation#: 170   best: 108.0      average: 102.42 change: 0.18
generation#: 180   best: 108.0      average: 102.68 change: 0.09
generation#: 190   best: 108.0      average: 103.19 change: 0.14
generation#: 200   best: 108.0      average: 103.48 change: -0.06
generation#: 210   best: 108.0      average: 103.57 change: 0.10
generation#: 220   best: 108.0      average: 103.26 change: -0.04
generation#: 230   best: 108.0      average: 103.62 change: 0.04
generation#: 240   best: 108.0      average: 104.15 change: 0.23
generation#: 250   best: 108.0      average: 103.81 change: -0.06
generation#: 260   best: 108.0      average: 103.33 change: -0.18
generation#: 270   best: 108.0      average: 103.30 change: 0.03
```

```
generation#: 280  best: 108.0      average: 103.62 change: -0.03
generation#: 290  best: 108.0      average: 103.25 change: 0.0
generation#: 300  best: 108.0      average: 103.65 change: -0.039
generation#: 310  best: 108.0      average: 103.79 change: 0.03
generation#: 320  best: 108.0      average: 103.16 change: 0.04
generation#: 330  best: 108.0      average: 103.62 change: -0.08
generation#: 340  best: 108.0      average: 104.05 change: -0.03
generation#: 350  best: 108.0      average: 104.32 change: -0.06
generation#: 360  best: 108.0      average: 105.04 change: 0.05
generation#: 370  best: 108.0      average: 104.83 change: -0.29
generation#: 380  best: 108.0      average: 104.55 change: 0.14
generation#: 390  best: 108.0      average: 104.30 change: -0.04
generation#: 400  best: 108.0      average: 104.64 change: 0.23
generation#: 410  best: 108.0      average: 104.94 change: -0.04
generation#: 420  best: 108.0      average: 105.24 change: 0.04
generation#: 430  best: 108.0      average: 105.21 change: 0.09
generation#: 440  best: 108.0      average: 105.01 change: -0.11
generation#: 450  best: 108.0      average: 104.65 change: 0.02
generation#: 460  best: 108.0      average: 104.21 change: 0.04
generation#: 470  best: 108.0      average: 103.87 change: -0.14
generation#: 480  best: 108.0      average: 103.98 change: -0.03
generation#: 490  best: 108.0      average: 104.01 change: -0.09
generation#: 500  best: 108.0      average: 103.72 change: -0.01
generation#: 510  best: 108.0      average: 103.69 change: -0.06
generation#: 520  best: 108.0      average: 103.71 change: 0.00
generation#: 530  best: 108.0      average: 103.72 change: 0.06
generation#: 540  best: 108.0      average: 103.66 change: -0.04
generation#: 550  best: 108.0      average: 104.00 change: 0.09
generation#: 560  best: 108.0      average: 103.80 change: 0.04
generation#: 570  best: 108.0      average: 103.84 change: 0.0
generation#: 580  best: 108.0      average: 104.05 change: 0.04
generation#: 590  best: 108.0      average: 104.45 change: -0.04
generation#: 600  best: 108.0      average: 104.25 change: 0.0
generation#: 610  best: 108.0      average: 104.25 change: 0.0
generation#: 620  best: 108.0      average: 104.40 change: 0.0
generation#: 630  best: 108.0      average: 104.60 change: 0.09
generation#: 640  best: 108.0      average: 104.80 change: 0.09
generation#: 650  best: 108.0      average: 104.95 change: 0.20
generation#: 660  best: 108.0      average: 104.90 change: 0.0
generation#: 670  best: 108.0      average: 104.55 change: 0.09
generation#: 680  best: 108.0      average: 104.70 change: 0.04
generation#: 690  best: 108.0      average: 104.80 change: -0.10
generation#: 700  best: 108.0      average: 104.00 change: -0.04
generation#: 710  best: 108.0      average: 103.90 change: 0.05
generation#: 720  best: 108.0      average: 103.65 change: -0.04
generation#: 730  best: 108.0      average: 103.90 change: -0.04
generation#: 740  best: 108.0      average: 104.30 change: 0.04
generation#: 750  best: 108.0      average: 104.30 change: -0.10
generation#: 760  best: 108.0      average: 104.30 change: -0.10
generation#: 770  best: 108.0      average: 104.55 change: 0.0
generation#: 780  best: 108.0      average: 104.85 change: 0.04
generation#: 790  best: 108.0      average: 104.85 change: 0.0
generation#: 800  best: 108.0      average: 104.60 change: 0.0
generation#: 810  best: 108.0      average: 105.00 change: 0.0
generation#: 820  best: 108.0      average: 104.95 change: 0.04
generation#: 830  best: 108.0      average: 104.75 change: -0.09
generation#: 840  best: 108.0      average: 105.05 change: 0.09
generation#: 850  best: 108.0      average: 105.30 change: 0.09
generation#: 860  best: 108.0      average: 105.55 change: 0.09
generation#: 870  best: 108.0      average: 105.45 change: 0.04
generation#: 880  best: 108.0      average: 105.65 change: 0.0
generation#: 890  best: 108.0      average: 106.20 change: -0.09
generation#: 900  best: 108.0      average: 106.35 change: 0.04
generation#: 910  best: 108.0      average: 106.50 change: 0.0
generation#: 920  best: 108.0      average: 106.75 change: 0.09
generation#: 930  best: 108.0      average: 107.20 change: 0.0
```

```
generation#: 940  best: 108.0        average: 107.05 change: 0.04
generation#: 950  best: 108.0        average: 107.05 change: 0.0
generation#: 960  best: 108.0        average: 107.00 change: 0.0
generation#: 970  best: 108.0        average: 107.20 change: 0.10
generation#: 980  best: 108.0        average: 107.20 change: 0.04
generation#: 990  best: 108.0        average: 107.30 change: -0.04
generation#: 1000 best: 108.0        average: 107.15 change: -0.04
generation#: 1010 best: 108.0        average: 107.40 change: 0.0
generation#: 1020 best: 108.0        average: 107.15 change: 0.0
generation#: 1030 best: 108.0        average: 107.60 change: 0.0
generation#: 1040 best: 108.0        average: 107.50 change: -0.09
generation#: 1050 best: 108.0        average: 107.70 change: 0.0
generation#: 1060 best: 108.0        average: 107.60 change: 0.09
generation#: 1070 best: 108.0        average: 107.50 change: 0.0
generation#: 1080 best: 108.0        average: 107.95 change: 0.04
generation#: 1090 best: 108.0        average: 108.00 change: 0.0
generation#:>1090 best: 108.0        average: 108.00 change: 0.0
```

Statistics for Table 4

Population 1

```
generation#: 0    best: 105.0        average: 101.85 change: 101.85
generation#: 10   best: 106.0        average: 101.95 change: -0.20
generation#: 20   best: 106.0        average: 101.90 change: 0.05
generation#: 30   best: 106.0        average: 102.40 change: 0.25
generation#: 40   best: 106.0        average: 102.25 change: 0.0
generation#: 50   best: 107.0        average: 102.90 change: 0.45
generation#: 60   best: 107.0        average: 102.90 change: 0.0
generation#: 70   best: 107.0        average: 103.25 change: 0.04
generation#: 80   best: 107.0        average: 102.10 change: 0.0
generation#: 90   best: 107.0        average: 102.90 change: 0.0
generation#: 100  best: 107.0        average: 103.75 change: 0.70
generation#: 110  best: 107.0        average: 104.80 change: -0.75
generation#: 120  best: 107.0        average: 102.50 change: 0.0
generation#: 130  best: 107.0        average: 104.80 change: -0.10
generation#: 140  best: 107.0        average: 103.65 change: 0.0
generation#: 150  best: 107.0        average: 103.95 change: -0.39
generation#: 160  best: 107.0        average: 104.55 change: 0.0
generation#: 170  best: 107.0        average: 103.55 change: 0.0
generation#: 180  best: 107.0        average: 101.75 change: -0.34
generation#: 190  best: 107.0        average: 101.65 change: 0.0
generation#: 200  best: 107.0        average: 101.05 change: -0.65
generation#: 210  best: 107.0        average: 103.55 change: 0.0
generation#: 220  best: 107.0        average: 103.75 change: 0.0
generation#: 230  best: 107.0        average: 104.00 change: 0.0
generation#: 240  best: 107.0        average: 104.45 change: 0.0
generation#: 250  best: 107.0        average: 103.80 change: 0.39
generation#: 260  best: 107.0        average: 104.50 change: 0.0
generation#: 270  best: 107.0        average: 104.80 change: 0.0
generation#: 280  best: 107.0        average: 106.20 change: 0.40
generation#: 290  best: 107.0        average: 106.55 change: 0.0
generation#: 300  best: 107.0        average: 106.60 change: 0.09
generation#: 310  best: 107.0        average: 106.70 change: 0.0
generation#: 320  best: 107.0        average: 105.90 change: -0.09
generation#: 330  best: 107.0        average: 106.10 change: -0.10
generation#: 340  best: 107.0        average: 105.90 change: 0.10
generation#: 350  best: 107.0        average: 106.60 change: -0.10
generation#: 360  best: 107.0        average: 106.50 change: 0.09
generation#: 370  best: 107.0        average: 106.80 change: 0.0
generation#: 380  best: 107.0        average: 106.90 change: 0.0
```

```
generation#: 390  best: 107.0      average: 106.90 change: 0.0
generation#: 400  best: 107.0      average: 106.90 change: 0.0
generation#: 410  best: 107.0      average: 106.90 change: 0.0
generation#: 420  best: 107.0      average: 106.60 change: -0.20
generation#: 430  best: 107.0      average: 106.60 change: 0.0
generation#: 440  best: 107.0      average: 106.80 change: 0.0
generation#: 450  best: 107.0      average: 106.60 change: 0.09
generation#: 460  best: 107.0      average: 106.50 change: 0.09
generation#: 470  best: 107.0      average: 106.40 change: 0.0
generation#: 480  best: 107.0      average: 106.90 change: 0.0
generation#: 490  best: 107.0      average: 106.90 change: 0.0
generation#: 500  best: 107.0      average: 106.60 change: 0.0
generation#: 510  best: 107.0      average: 106.20 change: 0.0
generation#: 520  best: 107.0      average: 105.60 change: -0.10
generation#: 530  best: 107.0      average: 106.20 change: 0.10
generation#: 540  best: 107.0      average: 106.20 change: 0.10
generation#: 550  best: 107.0      average: 105.90 change: 0.0
generation#: 560  best: 107.0      average: 105.70 change: 0.10
generation#: 570  best: 107.0      average: 105.40 change: 0.0
generation#: 580  best: 107.0      average: 106.20 change: -0.09
generation#: 590  best: 107.0      average: 106.20 change: -0.09
generation#: 600  best: 107.0      average: 105.60 change: 0.0
generation#: 610  best: 107.0      average: 106.00 change: 0.09
generation#: 620  best: 107.0      average: 106.60 change: 0.0
generation#: 630  best: 107.0      average: 106.60 change: 0.0
generation#: 640  best: 107.0      average: 106.60 change: 0.0
generation#: 650  best: 107.0      average: 106.80 change: 0.0
generation#: 660  best: 107.0      average: 106.80 change: 0.0
generation#: 670  best: 107.0      average: 106.70 change: 0.0
generation#: 680  best: 107.0      average: 106.50 change: 0.0
generation#: 690  best: 107.0      average: 106.60 change: 0.0
generation#: 700  best: 107.0      average: 106.80 change: 0.0
generation#: 710  best: 107.0      average: 106.50 change: 0.0
generation#: 720  best: 107.0      average: 106.50 change: -0.09
generation#: 730  best: 107.0      average: 106.40 change: 0.0
generation#: 740  best: 107.0      average: 106.60 change: 0.0
generation#: 750  best: 107.0      average: 106.10 change: -0.30
generation#: 760  best: 107.0      average: 105.80 change: 0.0
generation#: 770  best: 107.0      average: 106.40 change: 0.0
generation#: 780  best: 107.0      average: 106.70 change: 0.0
generation#: 790  best: 107.0      average: 106.40 change: -0.09
generation#: 800  best: 107.0      average: 106.40 change: -0.09
generation#: 810  best: 107.0      average: 106.40 change: 0.0
generation#: 820  best: 107.0      average: 106.50 change: -0.09
generation#: 830  best: 107.0      average: 106.40 change: 0.0
generation#: 840  best: 107.0      average: 106.50 change: 0.0
generation#: 850  best: 107.0      average: 106.60 change: 0.0
generation#: 860  best: 107.0      average: 106.50 change: 0.0
generation#: 870  best: 107.0      average: 106.60 change: 0.19
generation#: 880  best: 107.0      average: 106.70 change: 0.10
generation#: 890  best: 107.0      average: 105.50 change: 0.0
generation#: 900  best: 107.0      average: 106.80 change: -0.20
generation#: 910  best: 107.0      average: 106.70 change: -0.09
generation#: 920  best: 107.0      average: 106.90 change: 0.0
generation#: 930  best: 107.0      average: 106.90 change: 0.0
generation#: 940  best: 107.0      average: 107.00 change: 0.09
generation#: 950  best: 107.0      average: 106.70 change: -0.29
generation#: 960  best: 107.0      average: 106.70 change: 0.0
generation#: 970  best: 107.0      average: 106.60 change: 0.0
generation#: 980  best: 107.0      average: 105.30 change: 0.0
generation#: 990  best: 107.0      average: 105.30 change: 0.0
generation#: 1000 best: 107.0      average: 106.50 change: -0.20
generation#: 1010 best: 107.0      average: 106.40 change: 0.0
generation#: 1020 best: 107.0      average: 106.40 change: 0.0
generation#: 1030 best: 107.0      average: 106.10 change: 0.0
generation#: 1040 best: 107.0      average: 105.80 change: 0.0
```

```
generation#: 1050 best: 107.0      average: 106.20 change: -0.09
generation#: 1060 best: 107.0      average: 106.50 change: 0.09
generation#: 1070 best: 107.0      average: 106.40 change: -0.09
generation#: 1080 best: 107.0      average: 106.40 change: 0.10
generation#: 1090 best: 107.0      average: 106.60 change: 0.0
generation#: 1100 best: 107.0      average: 106.40 change: 0.10
generation#: 1110 best: 107.0      average: 106.30 change: 0.09
generation#: 1120 best: 107.0      average: 107.00 change: 0.0
generation#: 1130 best: 107.0      average: 106.90 change: 0.0
generation#: 1140 best: 107.0      average: 106.90 change: 0.0
generation#: 1150 best: 107.0      average: 107.00 change: 0.09
generation#: 1160 best: 107.0      average: 107.00 change: 0.0
generation#:>1160 best: 107.0      average: 107.00 change: 0.0
```

Population 2

```
generation#: 0    best: 106.0      average: 101.55 change: 101.55
generation#: 10   best: 107.0      average: 101.00 change: 0.0
generation#: 20   best: 107.0      average: 100.90 change: 0.0
generation#: 30   best: 107.0      average: 100.10 change: -0.20
generation#: 40   best: 107.0      average: 102.45 change: 0.04
generation#: 50   best: 107.0      average: 102.80 change: -0.29
generation#: 60   best: 107.0      average: 101.95 change: -0.39
generation#: 70   best: 107.0      average: 102.30 change: -0.60
generation#: 80   best: 107.0      average: 102.60 change: 0.09
generation#: 90   best: 107.0      average: 103.90 change: 0.30
generation#: 100  best: 107.0      average: 102.80 change: 0.04
generation#: 110  best: 107.0      average: 102.95 change: 0.0
generation#: 120  best: 107.0      average: 102.85 change: 0.44
generation#: 130  best: 107.0      average: 100.95 change: 0.0
generation#: 140  best: 107.0      average: 100.25 change: 0.0
generation#: 150  best: 107.0      average: 100.25 change: 0.40
generation#: 160  best: 107.0      average: 101.20 change: 0.40
generation#: 170  best: 107.0      average: 101.20 change: 0.0
generation#: 180  best: 107.0      average: 101.85 change: 0.04
generation#: 190  best: 107.0      average: 101.70 change: 0.25
generation#: 200  best: 107.0      average: 102.85 change: 0.54
generation#: 210  best: 107.0      average: 102.90 change: -0.09
generation#: 220  best: 107.0      average: 103.05 change: 0.0
generation#: 230  best: 107.0      average: 103.75 change: 0.0
generation#: 240  best: 107.0      average: 104.40 change: 0.0
generation#: 250  best: 107.0      average: 105.80 change: -0.40
generation#: 260  best: 107.0      average: 106.50 change: 0.09
generation#: 270  best: 107.0      average: 106.50 change: 0.0
generation#: 280  best: 107.0      average: 106.20 change: 0.0
generation#: 290  best: 107.0      average: 106.50 change: 0.40
generation#: 300  best: 107.0      average: 106.80 change: -0.10
generation#: 310  best: 107.0      average: 107.00 change: 0.09
generation#: 320  best: 107.0      average: 107.00 change: 0.0
generation#: 330  best: 107.0      average: 106.90 change: 0.0
generation#: 340  best: 107.0      average: 106.90 change: 0.0
generation#: 350  best: 107.0      average: 106.30 change: 0.20
generation#: 360  best: 107.0      average: 106.80 change: 0.0
generation#: 370  best: 107.0      average: 106.90 change: 0.0
generation#: 380  best: 107.0      average: 107.00 change: 0.0
generation#: 390  best: 107.0      average: 107.00 change: 0.0
generation#: 400  best: 107.0      average: 107.00 change: 0.0
generation#: 410  best: 107.0      average: 107.00 change: 0.0
generation#: 420  best: 107.0      average: 107.00 change: 0.0
generation#: 430  best: 107.0      average: 106.80 change: 0.0
generation#: 440  best: 107.0      average: 106.50 change: 0.0
generation#: 450  best: 107.0      average: 106.60 change: -0.10
generation#: 460  best: 107.0      average: 106.60 change: 0.0
generation#: 470  best: 107.0      average: 107.00 change: 0.0
generation#: 480  best: 107.0      average: 106.20 change: -0.09
generation#: 490  best: 107.0      average: 106.50 change: 0.0
```

```
generation#: 500  best: 107.0     average: 106.80 change: -0.10
generation#: 510  best: 107.0     average: 106.10 change: 0.0
generation#: 520  best: 107.0     average: 106.10 change: 0.0
generation#: 530  best: 107.0     average: 105.80 change: -0.10
generation#: 540  best: 107.0     average: 105.60 change: 0.0
generation#: 550  best: 107.0     average: 105.60 change: 0.0
generation#: 560  best: 107.0     average: 105.90 change: 0.0
generation#: 570  best: 107.0     average: 106.10 change: -0.10
generation#: 580  best: 107.0     average: 105.10 change: -0.10
generation#: 590  best: 107.0     average: 105.40 change: 0.0
generation#: 600  best: 107.0     average: 106.30 change: 0.20
generation#: 610  best: 107.0     average: 106.60 change: 0.09
generation#: 620  best: 107.0     average: 106.60 change: 0.0
generation#: 630  best: 107.0     average: 106.90 change: 0.10
generation#: 640  best: 107.0     average: 106.90 change: 0.0
generation#: 650  best: 107.0     average: 106.70 change: 0.20
generation#: 660  best: 107.0     average: 106.90 change: 0.0
generation#: 670  best: 107.0     average: 106.90 change: 0.10
generation#: 680  best: 107.0     average: 107.00 change: 0.0
generation#: 690  best: 107.0     average: 107.00 change: 0.0
generation#: 700  best: 107.0     average: 107.00 change: 0.0
generation#: 710  best: 107.0     average: 107.00 change: 0.0
generation#: 720  best: 107.0     average: 107.00 change: 0.0
generation#: 730  best: 107.0     average: 106.80 change: 0.0
generation#: 740  best: 107.0     average: 106.80 change: 0.0
generation#: 750  best: 107.0     average: 106.80 change: 0.09
generation#: 760  best: 107.0     average: 106.60 change: -0.10
generation#: 770  best: 107.0     average: 106.70 change: 0.0
generation#: 780  best: 107.0     average: 105.90 change: -0.09
generation#: 790  best: 107.0     average: 106.20 change: 0.10
generation#: 800  best: 107.0     average: 106.40 change: 0.10
generation#: 810  best: 107.0     average: 106.50 change: 0.09
generation#: 820  best: 107.0     average: 106.90 change: 0.0
generation#: 830  best: 107.0     average: 106.30 change: 0.0
generation#: 840  best: 107.0     average: 106.40 change: 0.10
generation#: 850  best: 107.0     average: 106.20 change: -0.09
generation#: 860  best: 107.0     average: 105.60 change: 0.0
generation#: 870  best: 107.0     average: 105.50 change: 0.0
generation#: 880  best: 107.0     average: 106.70 change: 0.10
generation#: 890  best: 107.0     average: 107.00 change: 0.0
generation#: 900  best: 107.0     average: 105.60 change: 0.19
generation#: 910  best: 107.0     average: 105.80 change: 0.0
generation#: 920  best: 107.0     average: 105.80 change: 0.0
generation#: 930  best: 107.0     average: 105.90 change: 0.0
generation#: 940  best: 107.0     average: 105.50 change: 0.0
generation#: 950  best: 107.0     average: 105.40 change: 0.30
generation#: 960  best: 107.0     average: 105.50 change: 0.0
generation#: 970  best: 107.0     average: 105.30 change: -0.10
generation#: 980  best: 107.0     average: 106.60 change: 0.0
generation#: 990  best: 107.0     average: 106.70 change: 0.0
generation#: 1000 best: 107.0     average: 105.80 change: 0.29
generation#: 1010 best: 107.0     average: 106.40 change: 0.0
generation#: 1020 best: 107.0     average: 105.80 change: 0.0
generation#: 1030 best: 107.0     average: 106.30 change: 0.09
generation#: 1040 best: 107.0     average: 106.30 change: 0.0
generation#: 1050 best: 107.0     average: 106.30 change: 0.29
generation#: 1060 best: 107.0     average: 106.00 change: -0.09
generation#: 1070 best: 107.0     average: 106.10 change: 0.0
generation#: 1080 best: 107.0     average: 106.80 change: 0.09
generation#: 1090 best: 107.0     average: 106.90 change: 0.0
generation#: 1100 best: 107.0     average: 106.90 change: -0.09
generation#: 1110 best: 107.0     average: 106.90 change: 0.0
generation#: 1120 best: 107.0     average: 106.60 change: -0.20
generation#: 1130 best: 107.0     average: 107.00 change: 0.0
generation#: 1140 best: 107.0     average: 107.00 change: 0.0
generation#: 1150 best: 107.0     average: 106.90 change: -0.09
```

```
generation#: 1160 best: 107.0        average: 107.00 change: 0.0
generation#: 1170 best: 107.0        average: 107.00 change: 0.0
generation#: 1180 best: 107.0        average: 107.00 change: 0.0
generation#: 1190 best: 107.0        average: 107.00 change: 0.0
generation#: 1200 best: 107.0        average: 107.00 change: 0.0
generation#: 1210 best: 107.0        average: 107.00 change: 0.0
generation#: 1220 best: 107.0        average: 107.00 change: 0.0
generation#: 1230 best: 107.0        average: 107.00 change: 0.0
generation#: 1240 best: 107.0        average: 107.00 change: 0.0
generation#: 1250 best: 107.0        average: 107.00 change: 0.0
```

Statistics for Table 7

Population 1

```
generation#: 0    best: 105.0        average: 102.35 change: 102.35
generation#: 10   best: 106.0        average: 101.15 change: -0.19
generation#: 20   best: 107.0        average: 103.50 change: -0.34
generation#: 30   best: 107.0        average: 103.05 change: 0.0
generation#: 40   best: 107.0        average: 102.95 change: -0.09
generation#: 50   best: 107.0        average: 102.80 change: 0.0
generation#: 60   best: 107.0        average: 104.50 change: -0.45
generation#: 70   best: 107.0        average: 104.45 change: -0.04
generation#: 80   best: 105.0        average: 102.70 change: -0.20
generation#: 90   best: 105.0        average: 102.95 change: -0.09
generation#: 100  best: 106.0        average: 103.25 change: -0.29
generation#: 110  best: 105.0        average: 104.50 change: 0.15
generation#: 120  best: 108.0        average: 104.40 change: 0.45
generation#: 130  best: 108.0        average: 104.10 change: -0.30
generation#: 140  best: 106.0        average: 104.90 change: -0.19
generation#: 150  best: 106.0        average: 105.05 change: 0.0
generation#: 160  best: 107.0        average: 104.80 change: -0.15
generation#: 170  best: 107.0        average: 105.30 change: 0.04
generation#: 180  best: 108.0        average: 104.50 change: -0.04
generation#: 190  best: 108.0        average: 103.95 change: -0.45
generation#: 200  best: 108.0        average: 104.50 change: 0.29
generation#: 210  best: 108.0        average: 104.30 change: 0.0
generation#: 220  best: 110.0        average: 105.05 change: -0.65
generation#: 230  best: 110.0        average: 105.55 change: -0.25
generation#: 240  best: 110.0        average: 106.10 change: 0.34
generation#: 250  best: 110.0        average: 106.20 change: 0.0
generation#: 260  best: 110.0        average: 104.85 change: -0.15
generation#: 270  best: 110.0        average: 104.80 change: 0.0
generation#: 280  best: 110.0        average: 104.80 change: 0.39
generation#: 290  best: 110.0        average: 105.95 change: -0.29
generation#: 300  best: 110.0        average: 104.85 change: 0.25
generation#: 310  best: 110.0        average: 105.45 change: 0.25
generation#: 320  best: 110.0        average: 106.35 change: 0.5
generation#: 330  best: 110.0        average: 107.00 change: 0.0
generation#: 340  best: 110.0        average: 107.15 change: 0.15
generation#: 350  best: 110.0        average: 107.25 change: 0.0
generation#: 360  best: 110.0        average: 107.40 change: -0.34
generation#: 370  best: 110.0        average: 106.80 change: -0.10
generation#: 380  best: 110.0        average: 106.00 change: -0.75
generation#: 390  best: 110.0        average: 105.75 change: -0.25
generation#: 400  best: 110.0        average: 106.15 change: -0.34
generation#: 410  best: 110.0        average: 106.00 change: 0.0
generation#: 420  best: 110.0        average: 107.15 change: 1.15
generation#: 430  best: 110.0        average: 107.25 change: 0.0
generation#: 440  best: 110.0        average: 107.10 change: 0.34
generation#: 450  best: 110.0        average: 107.10 change: 0.0
generation#: 460  best: 110.0        average: 106.05 change: -0.40
generation#: 470  best: 110.0        average: 106.00 change: 0.0
```

```
generation#: 480   best: 110.0        average: 107.75 change: -0.25
generation#: 490   best: 110.0        average: 108.00 change: 0.0
generation#: 500   best: 110.0        average: 107.50 change: -0.5
generation#: 510   best: 110.0        average: 108.00 change: 0.0
generation#: 520   best: 110.0        average: 106.25 change: 0.0
generation#: 530   best: 110.0        average: 106.25 change: 0.0
generation#: 540   best: 110.0        average: 106.50 change: 0.25
generation#: 550   best: 110.0        average: 107.00 change: 0.0
generation#: 560   best: 110.0        average: 105.50 change: 0.0
generation#: 570   best: 110.0        average: 107.25 change: 0.5
generation#: 580   best: 110.0        average: 107.00 change: 0.75
generation#: 590   best: 110.0        average: 108.25 change: 0.5
generation#: 600   best: 110.0        average: 108.50 change: 0.0
generation#: 610   best: 110.0        average: 108.25 change: -0.25
generation#: 620   best: 110.0        average: 107.50 change: 0.0
generation#: 630   best: 110.0        average: 107.25 change: 0.0
generation#: 640   best: 110.0        average: 107.25 change: 0.25
generation#: 650   best: 110.0        average: 107.75 change: 0.0
generation#: 660   best: 110.0        average: 107.75 change: 0.25
generation#: 670   best: 110.0        average: 107.25 change: -0.25
generation#: 680   best: 110.0        average: 107.25 change: 0.25
generation#: 690   best: 110.0        average: 108.50 change: 0.25
generation#: 700   best: 110.0        average: 108.75 change: 0.25
generation#: 710   best: 110.0        average: 109.25 change: 0.0
generation#: 720   best: 110.0        average: 109.50 change: 0.25
generation#: 730   best: 110.0        average: 110.00 change: 0.0
generation#: 740   best: 110.0        average: 109.75 change: 0.25
generation#: 750   best: 110.0        average: 109.50 change: -0.25
generation#: 760   best: 110.0        average: 109.25 change: -0.5
generation#: 770   best: 110.0        average: 110.00 change: 0.0
generation#:>770   best: 110.0        average: 110.00 change: 0.0
```

Population 2

```
generation#: 0     best: 104.0        average: 100.75 change: 100.75
generation#: 10    best: 105.0        average: 101.85 change: 0.44
generation#: 20    best: 105.0        average: 102.50 change: 0.29
generation#: 30    best: 105.0        average: 102.75 change: 0.0
generation#: 40    best: 107.0        average: 104.35 change: -0.35
generation#: 50    best: 107.0        average: 104.40 change: 0.40
generation#: 60    best: 107.0        average: 103.55 change: 0.14
generation#: 70    best: 106.0        average: 103.40 change: 0.30
generation#: 80    best: 107.0        average: 104.55 change: -0.35
generation#: 90    best: 107.0        average: 104.30 change: 0.0
generation#: 100   best: 105.0        average: 103.95 change: -0.09
generation#: 110   best: 106.0        average: 103.45 change: 0.0
generation#: 120   best: 106.0        average: 104.45 change: -0.04
generation#: 130   best: 106.0        average: 104.90 change: 0.10
generation#: 140   best: 108.0        average: 105.20 change: 0.95
generation#: 150   best: 108.0        average: 105.25 change: -0.04
generation#: 160   best: 107.0        average: 104.50 change: 0.20
generation#: 170   best: 107.0        average: 104.80 change: 0.14
generation#: 180   best: 108.0        average: 104.80 change: 0.14
generation#: 190   best: 108.0        average: 105.15 change: 0.15
generation#: 200   best: 108.0        average: 106.05 change: -0.20
generation#: 210   best: 108.0        average: 105.70 change: 0.0
generation#: 220   best: 108.0        average: 104.75 change: -0.20
generation#: 230   best: 108.0        average: 103.85 change: -0.20
generation#: 240   best: 108.0        average: 107.05 change: -0.40
generation#: 250   best: 108.0        average: 105.45 change: 0.0
generation#: 260   best: 110.0        average: 107.15 change: -0.44
generation#: 270   best: 110.0        average: 107.20 change: 0.0
generation#: 280   best: 110.0        average: 106.00 change: 0.15
generation#: 290   best: 110.0        average: 104.60 change: 0.25
generation#: 300   best: 110.0        average: 106.75 change: 0.0
generation#: 310   best: 110.0        average: 106.40 change: 0.0
```

```
generation#: 320  best: 110.0    average: 105.75 change: -0.59
generation#: 330  best: 110.0    average: 107.25 change: 0.25
generation#: 340  best: 110.0    average: 106.45 change: 0.25
generation#: 350  best: 110.0    average: 105.65 change: -0.34
generation#: 360  best: 110.0    average: 104.95 change: 0.35
generation#: 370  best: 110.0    average: 104.45 change: -0.14
generation#: 380  best: 110.0    average: 107.50 change: -0.5
generation#: 390  best: 110.0    average: 106.55 change: 0.14
generation#: 400  best: 110.0    average: 107.50 change: 0.0
generation#: 410  best: 110.0    average: 107.75 change: 0.25
generation#: 420  best: 110.0    average: 107.00 change: -0.5
generation#: 430  best: 110.0    average: 108.00 change: 0.0
generation#: 440  best: 110.0    average: 105.10 change: 0.14
generation#: 450  best: 110.0    average: 105.15 change: 0.0
generation#: 460  best: 110.0    average: 108.00 change: 0.0
generation#: 470  best: 110.0    average: 106.00 change: 0.0
generation#: 480  best: 110.0    average: 106.00 change: 0.0
generation#: 490  best: 110.0    average: 105.50 change: -0.5
generation#: 500  best: 110.0    average: 106.00 change: 0.0
generation#: 510  best: 110.0    average: 106.00 change: 0.0
generation#: 520  best: 110.0    average: 107.75 change: 0.0
generation#: 530  best: 110.0    average: 108.00 change: 0.0
generation#: 540  best: 110.0    average: 105.25 change: -0.25
generation#: 550  best: 110.0    average: 108.00 change: 0.0
generation#: 560  best: 110.0    average: 104.50 change: -0.25
generation#: 570  best: 110.0    average: 104.50 change: -0.25
generation#: 580  best: 110.0    average: 105.75 change: 0.0
generation#: 590  best: 110.0    average: 105.50 change: 0.25
generation#: 600  best: 110.0    average: 104.75 change: -0.75
generation#: 610  best: 110.0    average: 105.25 change: -0.25
generation#: 620  best: 110.0    average: 105.25 change: -0.25
generation#: 630  best: 110.0    average: 105.00 change: 0.0
generation#: 640  best: 110.0    average: 108.00 change: 0.0
generation#: 650  best: 110.0    average: 108.00 change: 0.0
generation#: 660  best: 110.0    average: 105.00 change: 0.0
generation#: 670  best: 110.0    average: 106.75 change: 0.0
generation#: 680  best: 110.0    average: 110.00 change: 0.0
generation#: 690  best: 110.0    average: 110.00 change: 0.0
generation#: 700  best: 110.0    average: 110.00 change: 0.0
generation#: 710  best: 110.0    average: 110.00 change: 0.0
generation#: 720  best: 110.0    average: 110.00 change: 0.0
generation#: 730  best: 110.0    average: 110.00 change: 0.0
generation#: 740  best: 110.0    average: 107.75 change: 1.0
generation#: 750  best: 110.0    average: 107.50 change: -0.25
generation#: 760  best: 110.0    average: 109.00 change: 0.0
generation#: 770  best: 110.0    average: 109.00 change: 0.0
generation#: 780  best: 110.0    average: 109.00 change: 0.0
generation#: 790  best: 110.0    average: 109.00 change: 0.0
generation#: 800  best: 110.0    average: 109.00 change: 0.0
generation#: 810  best: 110.0    average: 109.00 change: 0.0
generation#: 820  best: 110.0    average: 109.50 change: -0.5
generation#: 830  best: 110.0    average: 109.00 change: 0.0
generation#: 840  best: 110.0    average: 109.50 change: -0.5
generation#: 850  best: 110.0    average: 110.00 change: 0.0
generation#: 860  best: 110.0    average: 110.00 change: 0.0
generation#: 870  best: 110.0    average: 110.00 change: 0.0
generation#:>870  best: 110.0    average: 110.00 change: 0.0
```

Population 3

```
generation#: 0    best: 106.0    average: 101.75 change: 101.75
generation#: 10   best: 105.0    average: 102.00 change: -0.09
generation#: 20   best: 107.0    average: 103.80 change: 0.70
generation#: 30   best: 107.0    average: 103.65 change: -0.04
generation#: 40   best: 106.0    average: 101.35 change: 0.79
generation#: 50   best: 106.0    average: 102.50 change: 0.15
```

```
generation#: 60   best: 106.0    average: 103.35 change: 0.34
generation#: 70   best: 107.0    average: 103.55 change: 0.0
generation#: 80   best: 108.0    average: 104.10 change: 0.44
generation#: 90   best: 108.0    average: 104.60 change: 0.14
generation#: 100  best: 107.0    average: 103.85 change: 0.14
generation#: 110  best: 107.0    average: 104.75 change: -0.25
generation#: 120  best: 107.0    average: 104.10 change: 0.1
generation#: 130  best: 107.0    average: 103.70 change: 0.25
generation#: 140  best: 107.0    average: 104.50 change: -0.25
generation#: 150  best: 107.0    average: 104.35 change: -0.10
generation#: 160  best: 107.0    average: 104.60 change: 0.049
generation#: 170  best: 107.0    average: 105.50 change: 0.049
generation#: 180  best: 107.0    average: 105.15 change: -0.39
generation#: 190  best: 107.0    average: 105.80 change: 0.0
generation#: 200  best: 107.0    average: 105.75 change: 0.049
generation#: 210  best: 107.0    average: 105.70 change: 0.0
generation#: 220  best: 108.0    average: 107.30 change: 0.20
generation#: 230  best: 108.0    average: 107.15 change: 0.0
generation#: 240  best: 108.0    average: 105.45 change: 0.04
generation#: 250  best: 108.0    average: 106.70 change: 0.15
generation#: 260  best: 108.0    average: 106.30 change: 0.25
generation#: 270  best: 108.0    average: 106.00 change: 0.0
generation#: 280  best: 110.0    average: 106.50 change: 0.25
generation#: 290  best: 110.0    average: 106.50 change: 0.09
generation#: 300  best: 110.0    average: 106.90 change: -0.14
generation#: 310  best: 110.0    average: 106.65 change: 0.10
generation#: 320  best: 110.0    average: 106.95 change: 0.15
generation#: 330  best: 110.0    average: 106.35 change: -0.15
generation#: 340  best: 110.0    average: 107.75 change: 0.25
generation#: 350  best: 110.0    average: 108.00 change: 0.0
generation#: 360  best: 110.0    average: 108.00 change: 0.0
generation#: 370  best: 110.0    average: 106.00 change: -0.15
generation#: 380  best: 110.0    average: 107.00 change: 0.0
generation#: 390  best: 110.0    average: 107.25 change: 0.0
generation#: 400  best: 110.0    average: 107.50 change: 0.0
generation#: 410  best: 110.0    average: 106.50 change: 0.15
generation#: 420  best: 110.0    average: 107.50 change: 0.25
generation#: 430  best: 110.0    average: 107.25 change: 0.0
generation#: 440  best: 110.0    average: 107.50 change: -0.5
generation#: 450  best: 110.0    average: 107.75 change: 0.0
generation#: 460  best: 110.0    average: 107.75 change: 0.0
generation#: 470  best: 110.0    average: 108.00 change: 0.0
generation#: 480  best: 110.0    average: 107.25 change: 0.0
generation#: 490  best: 110.0    average: 107.50 change: 0.0
generation#: 500  best: 110.0    average: 107.25 change: -0.25
generation#: 510  best: 110.0    average: 105.60 change: -0.75
generation#: 520  best: 110.0    average: 107.85 change: 0.0
generation#: 530  best: 110.0    average: 107.85 change: 0.0
generation#: 540  best: 110.0    average: 106.50 change: -0.5
generation#: 550  best: 110.0    average: 104.25 change: 0.5
generation#: 560  best: 110.0    average: 107.75 change: -0.25
generation#: 570  best: 110.0    average: 108.00 change: 0.0
generation#: 580  best: 110.0    average: 107.50 change: -0.5
generation#: 590  best: 110.0    average: 107.00 change: 0.25
generation#: 600  best: 110.0    average: 107.75 change: 0.0
generation#: 610  best: 110.0    average: 108.00 change: 0.25
generation#: 620  best: 110.0    average: 105.25 change: -0.25
generation#: 630  best: 110.0    average: 108.00 change: 0.0
generation#: 640  best: 110.0    average: 104.50 change: 0.25
generation#: 650  best: 110.0    average: 105.00 change: 0.25
generation#: 660  best: 110.0    average: 106.75 change: -0.25
generation#: 670  best: 110.0    average: 106.50 change: 0.5
generation#: 680  best: 110.0    average: 106.25 change: -0.25
generation#: 690  best: 110.0    average: 106.75 change: 0.25
generation#: 700  best: 110.0    average: 106.75 change: 0.25
generation#: 710  best: 110.0    average: 108.00 change: 0.0
```

```
generation#: 720  best: 110.0     average: 106.25 change: -0.5
generation#: 730  best: 110.0     average: 109.25 change: 0.25
generation#: 740  best: 110.0     average: 109.25 change: -0.75
generation#: 750  best: 110.0     average: 109.75 change: 0.5
generation#: 760  best: 110.0     average: 109.75 change: 0.0
generation#: 770  best: 110.0     average: 109.75 change: 0.0
generation#: 780  best: 110.0     average: 110.00 change: 0.0
generation#: 790  best: 110.0     average: 110.00 change: 0.0
generation#: 800  best: 110.0     average: 110.00 change: 0.0
generation#: 810  best: 110.0     average: 110.00 change: 0.0
generation#: 820  best: 110.0     average: 108.00 change: 0.75
generation#: 830  best: 110.0     average: 108.00 change: 0.0
generation#: 840  best: 110.0     average: 110.00 change: 0.0
generation#:>840  best: 110.0     average: 110.00 change: 0.0
```

Population 4

```
generation#: 0    best: 105.0     average: 101.65 change: 101.65
generation#: 10   best: 107.0     average: 103.65 change: 0.0
generation#: 20   best: 106.0     average: 100.65 change: -0.59
generation#: 30   best: 106.0     average: 101.30 change: -0.60
generation#: 40   best: 108.0     average: 101.40 change: 0.0
generation#: 50   best: 105.0     average: 102.30 change: -0.40
generation#: 60   best: 105.0     average: 102.35 change: -0.05
generation#: 70   best: 108.0     average: 103.30 change: 0.29
generation#: 80   best: 107.0     average: 103.50 change: 0.04
generation#: 90   best: 105.0     average: 103.15 change: 0.0
generation#: 100  best: 107.0     average: 103.50 change: 0.0
generation#: 110  best: 107.0     average: 103.95 change: 0.15
generation#: 120  best: 107.0     average: 103.95 change: -0.39
generation#: 130  best: 107.0     average: 105.10 change: 0.0
generation#: 140  best: 107.0     average: 105.70 change: -0.09
generation#: 150  best: 107.0     average: 105.50 change: 0.20
generation#: 160  best: 108.0     average: 105.15 change: 0.60
generation#: 170  best: 108.0     average: 105.30 change: 0.25
generation#: 180  best: 107.0     average: 105.65 change: 0.20
generation#: 190  best: 107.0     average: 105.50 change: 0.0
generation#: 200  best: 108.0     average: 105.95 change: 0.10
generation#: 210  best: 108.0     average: 106.85 change: 0.0
generation#: 220  best: 107.0     average: 105.55 change: -0.10
generation#: 230  best: 107.0     average: 105.45 change: 0.04
generation#: 240  best: 108.0     average: 104.55 change: -0.40
generation#: 250  best: 108.0     average: 104.75 change: 0.0
generation#: 260  best: 108.0     average: 105.75 change: 0.09
generation#: 270  best: 108.0     average: 106.30 change: 0.0
generation#: 280  best: 108.0     average: 106.15 change: 0.15
generation#: 290  best: 108.0     average: 106.80 change: -0.10
generation#: 300  best: 108.0     average: 106.85 change: 0.25
generation#: 310  best: 108.0     average: 106.60 change: 0.0
generation#: 320  best: 108.0     average: 106.55 change: 0.14
generation#: 330  best: 108.0     average: 106.20 change: -0.34
generation#: 340  best: 110.0     average: 106.30 change: 0.39
generation#: 350  best: 110.0     average: 105.90 change: 0.0
generation#: 360  best: 110.0     average: 106.70 change: 0.10
generation#: 370  best: 110.0     average: 108.00 change: 0.0
generation#: 380  best: 110.0     average: 106.40 change: 0.0
generation#: 390  best: 110.0     average: 107.25 change: 0.0
generation#: 400  best: 110.0     average: 106.35 change: 0.29
generation#: 410  best: 110.0     average: 107.75 change: 0.0
generation#: 420  best: 110.0     average: 106.85 change: 0.04
generation#: 430  best: 110.0     average: 105.15 change: 0.0
generation#: 440  best: 110.0     average: 107.50 change: 0.0
generation#: 450  best: 110.0     average: 107.75 change: 0.0
generation#: 460  best: 110.0     average: 107.40 change: 0.0
generation#: 470  best: 110.0     average: 106.25 change: -0.25
```

```
generation#: 480   best: 110.0     average: 106.50 change: 0.29
generation#: 490   best: 110.0     average: 106.35 change: -0.15
generation#: 500   best: 110.0     average: 106.95 change: 0.60
generation#: 510   best: 110.0     average: 107.75 change: 0.0
generation#: 520   best: 110.0     average: 105.75 change: 0.0
generation#: 530   best: 110.0     average: 105.50 change: 0.0
generation#: 540   best: 110.0     average: 108.00 change: 0.0
generation#: 550   best: 110.0     average: 105.25 change: -0.25
generation#: 560   best: 110.0     average: 107.40 change: 0.15
generation#: 570   best: 110.0     average: 105.50 change: 0.0
generation#: 580   best: 110.0     average: 107.55 change: 0.0
generation#: 590   best: 110.0     average: 107.70 change: 0.0
generation#: 600   best: 110.0     average: 107.85 change: 0.14
generation#: 610   best: 110.0     average: 107.70 change: 0.0
generation#: 620   best: 110.0     average: 107.55 change: -0.15
generation#: 630   best: 110.0     average: 107.85 change: 0.0
generation#: 640   best: 110.0     average: 107.50 change: -0.5
generation#: 650   best: 110.0     average: 107.75 change: 0.0
generation#: 660   best: 110.0     average: 107.85 change: 0.0
generation#: 670   best: 110.0     average: 108.00 change: 0.0
generation#: 680   best: 110.0     average: 106.00 change: 0.0
generation#: 690   best: 110.0     average: 106.75 change: 0.0
generation#: 700   best: 110.0     average: 107.00 change: -0.5
generation#: 710   best: 110.0     average: 108.00 change: 0.0
generation#: 720   best: 110.0     average: 108.00 change: 0.0
generation#: 730   best: 110.0     average: 105.75 change: 0.25
generation#: 740   best: 110.0     average: 108.00 change: 0.0
generation#: 750   best: 110.0     average: 108.00 change: 0.0
generation#: 760   best: 110.0     average: 106.75 change: 0.0
generation#: 770   best: 110.0     average: 108.00 change: 0.0
generation#: 780   best: 110.0     average: 109.50 change: 0.0
generation#: 790   best: 110.0     average: 109.00 change: 0.0
generation#: 800   best: 110.0     average: 108.00 change: 0.25
generation#: 810   best: 110.0     average: 110.00 change: 0.0
generation#: 820   best: 110.0     average: 110.00 change: 0.0
generation#: 830   best: 110.0     average: 110.00 change: 0.0
generation#: 840   best: 110.0     average: 110.00 change: 0.5
generation#: 850   best: 110.0     average: 110.00 change: 0.0
generation#: 860   best: 110.0     average: 110.00 change: 0.0
generation#: 870   best: 110.0     average: 110.00 change: 0.0
generation#: 880   best: 110.0     average: 110.00 change: 0.0
```

Population 5

```
generation#: 0     best: 107.0     average: 101.70 change: 101.7
generation#: 10    best: 108.0     average: 101.55 change: 0.0
generation#: 20    best: 107.0     average: 103.50 change: -0.34
generation#: 30    best: 108.0     average: 102.55 change: 0.0
generation#: 40    best: 105.0     average: 102.20 change: -0.25
generation#: 50    best: 108.0     average: 102.40 change: -0.39
generation#: 60    best: 108.0     average: 102.20 change: -0.04
generation#: 70    best: 105.0     average: 102.60 change: 0.0
generation#: 80    best: 105.0     average: 102.45 change: -0.20
generation#: 90    best: 107.0     average: 103.80 change: 0.0
generation#: 100   best: 108.0     average: 104.90 change: -0.04
generation#: 110   best: 108.0     average: 104.85 change: -0.15
generation#: 120   best: 108.0     average: 105.40 change: -0.04
generation#: 130   best: 108.0     average: 105.70 change: 0.20
generation#: 140   best: 108.0     average: 106.05 change: -0.5
generation#: 150   best: 108.0     average: 106.00 change: 0.04
generation#: 160   best: 108.0     average: 106.00 change: -0.5
generation#: 170   best: 108.0     average: 106.15 change: 0.0
generation#: 180   best: 108.0     average: 106.35 change: -0.30
generation#: 190   best: 108.0     average: 106.15 change: -0.25
generation#: 200   best: 108.0     average: 106.30 change: -0.15
```

```
generation#: 210   best: 108.0        average: 106.25 change:  0.0
generation#: 220   best: 108.0        average: 106.30 change: -0.04
generation#: 230   best: 108.0        average: 105.85 change:  0.0
generation#: 240   best: 108.0        average: 103.60 change:  0.34
generation#: 250   best: 108.0        average: 103.60 change:  0.0
generation#: 260   best: 108.0        average: 104.80 change:  0.29
generation#: 270   best: 108.0        average: 106.45 change:  0.15
generation#: 280   best: 108.0        average: 106.90 change: -0.5
generation#: 290   best: 108.0        average: 106.45 change:  0.0
generation#: 300   best: 108.0        average: 106.45 change: -0.09
generation#: 310   best: 108.0        average: 106.40 change:  0.15
generation#: 320   best: 108.0        average: 106.55 change: -0.20
generation#: 330   best: 108.0        average: 106.50 change: -0.25
generation#: 340   best: 108.0        average: 106.00 change: -0.79
generation#: 350   best: 108.0        average: 105.95 change:  0.0
generation#: 360   best: 110.0        average: 106.50 change:  0.09
generation#: 370   best: 110.0        average: 106.90 change:  0.0
generation#: 380   best: 110.0        average: 104.05 change:  0.34
generation#: 390   best: 110.0        average: 104.60 change:  0.14
generation#: 400   best: 110.0        average: 104.80 change:  0.34
generation#: 410   best: 110.0        average: 105.15 change:  0.15
generation#: 420   best: 110.0        average: 105.55 change: -0.20
generation#: 430   best: 110.0        average: 107.00 change:  0.15
generation#: 440   best: 110.0        average: 107.40 change:  0.0
generation#: 450   best: 110.0        average: 107.40 change:  0.0
generation#: 460   best: 110.0        average: 106.60 change:  0.39
generation#: 470   best: 110.0        average: 106.80 change: -0.15
generation#: 480   best: 110.0        average: 105.05 change:  0.04
generation#: 490   best: 110.0        average: 105.80 change:  0.0
generation#: 500   best: 110.0        average: 106.45 change:  0.70
generation#: 510   best: 110.0        average: 107.40 change:  0.0
generation#: 520   best: 110.0        average: 107.75 change: -0.25
generation#: 530   best: 110.0        average: 107.50 change:  0.0
generation#: 540   best: 110.0        average: 107.70 change:  0.0
generation#: 550   best: 110.0        average: 107.10 change:  0.0
generation#: 560   best: 110.0        average: 107.10 change:  0.09
generation#: 570   best: 110.0        average: 107.40 change:  0.0
generation#: 580   best: 110.0        average: 107.50 change:  0.0
generation#: 590   best: 110.0        average: 107.50 change:  0.0
generation#: 600   best: 110.0        average: 107.00 change:  0.0
generation#: 610   best: 110.0        average: 107.25 change:  0.0
generation#: 620   best: 110.0        average: 107.50 change:  0.25
generation#: 630   best: 110.0        average: 107.50 change:  0.0
generation#: 640   best: 110.0        average: 107.70 change:  0.0
generation#: 650   best: 110.0        average: 107.55 change:  0.0
generation#: 660   best: 110.0        average: 108.00 change:  0.0
generation#: 670   best: 110.0        average: 108.00 change:  0.0
generation#: 680   best: 110.0        average: 108.00 change:  0.0
generation#: 690   best: 110.0        average: 108.00 change:  0.0
generation#: 700   best: 110.0        average: 108.00 change:  0.0
generation#: 710   best: 110.0        average: 106.50 change: -0.25
generation#: 720   best: 110.0        average: 106.50 change:  0.0
generation#: 730   best: 110.0        average: 106.00 change: -0.25
generation#: 740   best: 110.0        average: 106.75 change:  0.0
generation#: 750   best: 110.0        average: 107.25 change:  0.0
generation#: 760   best: 110.0        average: 108.00 change:  0.0
generation#: 770   best: 110.0        average: 107.50 change:  0.0
generation#: 780   best: 110.0        average: 110.00 change:  0.0
generation#: 790   best: 110.0        average: 109.00 change:  0.0
generation#: 800   best: 110.0        average: 109.75 change: -0.25
generation#: 810   best: 110.0        average: 107.75 change:  0.25
generation#: 820   best: 110.0        average: 110.00 change:  0.0
generation#:>820   best: 110.0        average: 110.00 change:  0.0
```

APPENDIX G

CODE LISTINGS

This appendix contains the code listings of the classes and packages that comprise SIPAGAR in alphabetical order.

class Decoder

```
public class decoder
{
   private int subgraph;       // Number of vertices in the
                               // monochromatic subgraph that we are
                               // trying to avoid
   private int num_colors;     // Number of colors that will be used to
                               // color the edges of the complete
                               // graph
   private int best_color;     // Color that results in the fewest
                               // number of monochromatic subgraphs
                               // being formed
   private int fewest_sub;     // Number of monochromatic subgraphs
                               // that result with best_color
   private triangle tri;       // Helper function to find monochromatic
                               // triangles

   // ** Constructor ** //

   public decoder(int sub_vertices, int colors)
   {
        subgraph = sub_vertices;
        num_colors = colors;
        best_color = 0;
        fewest_sub = 1000;
        tri = new triangle();
   }


   // ** getnumsubvertices ** //
   //
   // Returns number of vertices in monochromatic subgraph
   //

   public synchronized int getnumsubvertices() { return subgraph; }
```

```
// ** getnumcolors ** //
//
// Returns number of colors used to color the edges of the
// complete graph
//

public synchronized int getnumcolors() { return num_colors; }

// ** getbestcolor ** //
//
// Returns color that results in the fewest number of
// monochromatic subgraphs
//

public synchronized int getbestcolor() { return best_color; }

// ** setbestcolor ** //
//
// Sets the color that results in the fewest number of
// monochromatic subgraphs
//

public synchronized void setbestcolor(int bestcol) { best_color =
bestcol; }

// ** getfewestsub ** //
//
// Returns the number of monochromatic subgraphs formed with
// best_color
//

public synchronized int getfewestsub() { return fewest_sub; }

// ** setfewestsub ** //
//
// Sets the number of monochromatic subgraphs that result with
// best_color
//

public synchronized void setfewestsub(int numsub) { fewest_sub =
numsub; }

// ** decode ** //
//
// Assigns a fitness value to a permutation according to the
// evaluation function.It uses supporting functions of class
//  "table" and "triangle" for this purpose
//

public synchronized void decode(permutation p, table t)
{
     int penalty = 0;

   for(int e=0; e < p.getnumedge(); e++)
   {
     if(subgraph == 3)
        {
           tri.reset();

tri.find_triangle(p,t,this,t.i(p.getedge(e)),t.j(p.getedge(e)));

           p.setcolor(p.getedge(e), best_color);
           penalty = fewest_sub;
           p.setfitval(p.getfitval() - penalty);
        }
```

```
        fewest_sub = 1000;
      }
    } // End of decode

} // End of decoder
```

class GAException

```
import java.awt.TextArea;

// ** GAException ** //
//
// Exception class that handles problems that arise from
// running the genetic algorithm
//

public class GAException extends Exception
{
  // ** Constructor ** //

  public GAException() {}


  // ** report ** //
  //
  // Displays this exception's error message on a TextArea object
  //

  public void report(TextArea log, String message)
  {
    log.append("Error - GAException\n");
    log.append(message + "\n");
  }

} // End of GAException
```

class Gamigration

```
import java.awt.TextArea;
import selection.*;


// ** GAmigration ** //
//
// A GAmigration object is a thread that once started, continuously
// checks the conditions that trigger migration. When these
// conditions (migration frequency) are satisfied, the GAmigration
// object implements the migration among the populations according to
// the migration criteria(migration topology, migration size, migrant
// selection). Migration is done synchronously. A subpopulation stops
// when the migration criteria has been locally satisfied. The
// Gamigration object triggers migration when the migration criteria
// has been satisfied in all subpopulations. After migration is done
```

```
// according to the migration topology, GAmigration resumes the
// evolution of all the subpopulations.
class GAmigration extends Thread
{
  private int num_populations;      // Number of populations (between
                                    // 2 and 6
  private int topology;             // Migration topology indentifier
  private int size;                 // Number of individuals composing
                                    // each migration
  private int mig_select;           // Migrant selection policy
                                    // identifier

  private group pop1;               // Access to the first population
  private group pop2;               // Access to the second population
  private group pop3;               // Access to the third population
  private group pop4;               // Access to the fourth population
  private group pop5;               // Access to the fifth population
  private group pop6;               // Access to the sixth population
  private group_stats pop1_stats;   // Access to the first
                                    // population's statistics
  private group_stats pop2_stats;   // Access to the second
                                    // population's statistics
  private group_stats pop3_stats;   // Access to the third
                                    // population's statistics
  private group_stats pop4_stats;   // Access to the fourth
                                    // population's statistics
  private group_stats pop5_stats;   // Access to the fifth
                                    // population's statistics
  private group_stats pop6_stats;   // Access to the sixth
                                    // population's statistics

  private int[] pop1_migrants;      // Indexes of selected migrants in
                                    // population 1
  private int[] pop2_migrants;      // Indexes of selected migrants in
                                    // population 2

  private int[] pop3_migrants;      // Indexes of selected migrants in
                                    // population 3
  private int[] pop4_migrants;      // Indexes of selected migrants in
                                    // population 4
  private int[] pop5_migrants;      // Indexes of selected migrants in
                                    // population 5
  private int[] pop6_migrants;      // Indexes of selected migrants in
                                    // population 6

  private TextArea log;             // Access to the graphical user
                                    // interface

  // ** Constructor ** //

  public GAmigration(int numpopulations, int topo, int nummigrants,
                     int select, TextArea guilog, group island1,
                     group island2, group island3, group island4,
                     group island5, group island6, group_stats
                     island1_stats, group_stats island2_stats,
                     group_stats island3_stats, group_stats
                     island4_stats, group_stats island5_stats,
                     group_stats island6_stats)
  {
    num_populations = numpopulations;
    topology = topo;
    size = nummigrants;
    mig_select = select;
    log = guilog;
```

```
// Obtain access to all possible populations (even if not all
// populations exist). The populations that actually exist and
// thus the populations that will be operated on is determined by
// *num_populations*. This simplifies the construction of the
// GAmigration object.

  pop1 = island1;
  pop1_stats = island1_stats;
  pop2 = island2;
  pop2_stats = island2_stats;
  pop3 = island3;
  pop3_stats = island3_stats;
  pop4 = island4;
  pop4_stats = island4_stats;
  pop5 = island5;
  pop5_stats = island5_stats;
  pop6 = island6;
  pop6_stats = island6_stats;

  pop1_migrants = new int[size];
  pop2_migrants = new int[size];
  pop3_migrants = new int[size];
  pop4_migrants = new int[size];
  pop5_migrants = new int[size];
  pop6_migrants = new int[size];

} // End of constructor



// ** trigger_migration ** //
//
// This method runs indefinitely until the number of generations
// that have occurred in all populations since the last migration
// took place is equal to the migration frequency. When this
// occurs, trigger_migration returns the value *true*, which
// indicates to GAmigration that migration should take place.


private boolean trigger_migration()
{
  switch(num_populations)
  {
    case 2:
      return(pop1.ready() && pop2.ready());
    case 3:
      return(pop1.ready() && pop2.ready() && pop3.ready());
    case 4:
      return(pop1.ready() && pop2.ready() && pop3.ready() &&
pop4.ready());
    case 5:
      return(pop1.ready() && pop2.ready() && pop3.ready() &&
             pop4.ready() && pop5.ready());
    case 6:
      return(pop1.ready() && pop2.ready() && pop3.ready() &&
             pop4.ready() && pop5.ready() && pop6.ready());
    default:
      return false;
  }

} // End of trigger_migration


// ** select_migrants ** //
//
// This method selects the permutation that will be chosen as
```

```
// migrants in each population according to the Migrant Selection
// Strategy.
//

private void select_migrants()
{
  // Select *size* migrants in each population according to the
  // selection strategy

  for(int i=0; i < size; i++) // For the number of migrants
  {
    if(mig_select == 1) // Random
      pop1_migrants[i] = (int) (Math.random() * (pop1.get_popsize()
- 1));
    else // Roulette
      pop1_migrants[i] = selection.Roulette.select(pop1.get_pop(),
pop1.get_popsize());

    if(num_populations >= 2)
      if(mig_select == 1) // Random
        pop2_migrants[i] = (int) (Math.random()*(pop2.get_popsize()
- 1));
      else // Roulette
        pop2_migrants[i]= selection.Roulette.select(pop2.get_pop(),
pop2.get_popsize());

    if(num_populations >= 3)
      if(mig_select == 1) // Random
        pop3_migrants[i] = (int) (Math.random()*(pop3.get_popsize()
- 1));
      else // Roulette
        pop3_migrants[i]= selection.Roulette.select(pop3.get_pop(),
pop3.get_popsize());

    if(num_populations >= 4)
      if(mig_select == 1) // Random
        pop4_migrants[i] = (int) (Math.random()*(pop4.get_popsize()
- 1));
      else // Roulette
        pop4_migrants[i]= selection.Roulette.select(pop4.get_pop(),
pop4.get_popsize());

    if(num_populations >= 5)
      if(mig_select == 1) // Random
        pop5_migrants[i] = (int) (Math.random()*(pop5.get_popsize()
- 1));
      else // Roulette
        pop5_migrants[i]= selection.Roulette.select(pop5.get_pop(),
pop5.get_popsize());

    if(num_populations == 6)
      if(mig_select == 1) // Random
        pop6_migrants[i] = (int) (Math.random()*(pop6.get_popsize()
- 1));
      else // Roulette
        pop6_migrants[i]= selection.Roulette.select(pop6.get_pop(),
pop6.get_popsize());
  }

} // End of select_migrants


// ** do_migration ** //
//
// This method transfers the selected migrants in each population
// to another population according to the migration topology.
```

```
//

private void do_migration()
{
  permutation[] temp = new permutation[size];
  permutation[] temp2 = new permutation[size];

  if(topology == 1) // Ring
  {

    // Make a temporary copy of the permutations in pop2 that were
    // selected as migrants

    for(int k=0; k < size; k++)
      temp[k] = pop2.get_permutation(pop2_migrants[k]);

    // Transfer migrants in population 1 to population 2

    for(int k=0; k < size; k++)
      {pop2.set_permutation(pop1.get_permutation(pop1_migrants[k]),
pop2_migrants[k]);
       log.append("pop1 --> pop2\n");}
       log.append("\n");

    if(num_populations >= 2)
      if(num_populations > 2)
       {

 // Make a temporary copy of the permutations in pop3 that
 // were selected as migrants

         for(int k=0; k < size; k++)
         temp2[k] = pop3.get_permutation(pop3_migrants[k]);

       // Transfer migrants in population 2 to population 3

         for(int k=0; k < size; k++)
           {pop3.set_permutation(temp[k], pop3_migrants[k]);
            log.append("pop2 --> pop3\n");}
           log.append("\n");
       }
      else // number of populations is 2
       {
          // Transfer migrants in population 2 to population 1

          for(int k=0; k < size; k++)
            {pop1.set_permutation(temp[k], pop1_migrants[k]);
             log.append("pop2 --> pop1\n");}
            log.append("\n");
       }

    if(num_populations >= 3)
     if(num_populations > 3)
       {
         // Make a temporary copy of the permutations in pop4 that
// were selected as migrants

         for(int k=0; k < size; k++)
         temp[k] = pop4.get_permutation(pop4_migrants[k]);

         // Transfer migrants in population 3 to population 4

         for(int k=0; k < size; k++)
           {pop4.set_permutation(temp2[k], pop4_migrants[k]);
            log.append("pop3 --> pop4\n");}
           log.append("\n");
```

```
    }
   else // number of populations is 3
    {

      // Transfer migrants in population 3 to population 1

      for(int k=0; k < size; k++)
        {pop1.set_permutation(temp2[k], pop1_migrants[k]);
         log.append("pop3 --> pop1\n");}
        log.append("\n");
    }

  if(num_populations >= 4)
    if(num_populations > 4)
    {
      // Make a temporary copy of the permutations in pop5 that
// were selected as migrants

      for(int k=0; k < size; k++)
      temp2[k] = pop5.get_permutation(pop3_migrants[k]);

      // Transfer migrants in population 4 to population 5

      for(int k=0; k < size; k++)
        {pop5.set_permutation(temp[k], pop5_migrants[k]);
         log.append("pop4 --> pop5\n");}
        log.append("\n");
    }
   else // number of populations is 4
    {
      // Transfer migrants in population 4 to population 1

      for(int k=0; k < size; k++)
        {pop1.set_permutation(temp[k], pop1_migrants[k]);
         log.append("pop4 --> pop1\n");}
        log.append("\n");
    }

  if(num_populations >= 5)
    if(num_populations > 5)
    {
      // Make a temporary copy of the permutations in pop6 that
// were selected as migrants

      for(int k=0; k < size; k++)
      temp[k] = pop6.get_permutation(pop6_migrants[k]);

      // Transfer migrants in population 5 to population 6

      for(int k=0; k < size; k++)
        {pop6.set_permutation(temp2[k], pop6_migrants[k]);
         log.append("pop5 --> pop6\n");}
        log.append("\n");
    }
   else // number of populations is 5
    {
      // Transfer migrants in population 5 to population 1

      for(int k=0; k < size; k++)
        {pop1.set_permutation(temp2[k], pop1_migrants[k]);
         log.append("pop5 --> pop1\n");}
        log.append("\n");
    }
  if(num_populations == 6)
    {
      // Transfer migrants in population 6 to population 1
```

```
          for(int k=0; k < size; k++)
            {pop1.set_permutation(temp[k], pop1_migrants[k]);
             log.append("pop6 --> pop1\n");}
            log.append("\n");
        }
    }// Ring topology
} // End of do_migration

// ** run ** //
//
// The GAmigration thread runs indefinitely. When the migration
// frequency is satisfied (as signaled by *trigger_migration*),
// GAmigration proceeds to do the migration according to the
// migration parameters.
//

public void run()
{
  while(true)
  {
    if (trigger_migration())
    {
      log.append("*MIGRATION*\n");

      // Select the permutations that will migrate in each
// population according to the selection strategy

      select_migrants();
      do_migration();
      log.append("\n");

      // Resume the evolution of the subpopulations

      pop1.resume();
      pop1.set_ready(false);

      if(num_populations >= 2)
        {
          pop2.resume();
          pop2.set_ready(false);
        }

      if(num_populations >= 3)
        {
          pop3.resume();
          pop3.set_ready(false);
        }

      if(num_populations >= 4)
        {
          pop4.resume();
          pop4.set_ready(false);
        }

      if(num_populations >= 5)
        {
          pop5.resume();
          pop5.set_ready(false);
        }

      if(num_populations == 6)
        {
          pop6.resume();
          pop6.set_ready(false);
        }
```

```
      }
    }
  }  // End of run
} // End of GAmigration
```

class global_stats

```
import java.awt.*;

// ** global_stats ** //
//
// Stores and graphically displays statistical information for all
// populations. A global_stats object displays the optimal fitness
// value for a particular run of the genetic algorithm (the goal).
// As the populations evolve, they provide information to the
// global_stats object about the best locally found permutation.
// The global_stats object graphically displays a permutation with
// the best fitness value found so far among all subpopulations.
//

class global_stats extends Panel
{
  private permutation best;            // A permutation with best
   fitness value
  private double best_fitness;         // Fitness value of best
  private double optimal_fitness;      // Optimal fitness value for a
   particular run
  private int num_populations;         // Number of populations
  private int num_vertices;            // Number of vertices
  private group_GUI graph;             // Graph of permutation best

  private TextField optimal;           // Displays optimal_fitness
  private TextArea best_permutation;   // Displays edges of best
  private TextArea best_coloring;      // Coloring of best_permutation
  private TextField bestfitness;       // Displays best_fitness

  private Label label1;
   // Label for optimal
  private Label label2;
   // Label for best_permutation
  private Label label3;                // Label for best_coloring
  private Label label4;                // Label for bestfitness
  // ** Constructor ** //
  public global_stats(int numpop, int numvert, group_GUI g)
  {
    optimal_fitness = 0;
    best_fitness = 0;
    num_populations = numpop;
    num_vertices = numvert;
    graph = g;

    label1 = new Label("OPTIMAL FITNESS");
    label2 = new Label("BEST PERMUTATION");
    label3 = new Label("COLORING");
    label4 = new Label("BEST FITNESS");

    optimal = new TextField(10);
    optimal.setEditable(false);
```

```
    best_permutation = new TextArea();
    best_permutation.setEditable(false);
    best_coloring = new TextArea();
    best_coloring.setEditable(false);
    bestfitness = new TextField(10);
    bestfitness.setEditable(false);

    setLayout(new GridLayout(8,1));
    setBackground(Color.lightGray);

    add(label1);
    add(optimal);
    add(label2);
    add(best_permutation);
    add(label3);
    add(best_coloring);
    add(label4);
    add(bestfitness);

    // Calculate and display the optimal fitness value

    optimal.setText(Integer.toString((num_vertices * (num_vertices -
  1))/2));

  } // End of Constructor


  // ** get_best_fitness ** //
  //
  // Returns the fitness value of the globally best permutation
  //

  public double get_best_fitness() { return best_fitness; }

  // ** set_best ** //
  //
  // Records the fitness value of the globally best permutation.
  // Displays the edges and coloring of the globally best permutation
  // and its graphical representation.
  //

  public synchronized void set_best(permutation p, double value,
  table T)
  {
    best = p;
    best_fitness = value;

    // Display graph of best permutation

    graph.set_graph(best, T);

    // Display edges of best permutation

    best.print_edges(best_permutation);

    // Display coloring of the edges of best permutation

    best.print_colors(best_coloring);

    // Display fitness of best permutation

    bestfitness.setText(Double.toString(best_fitness));

  } // End of set_best
} // End of global_stats
```

class group

```
import java.awt.TextArea;
import selection.*;
import crossover.*;
import mutation.*;


// ** group ** //
//
// A group contains permutations that are evolved towards an
// optimal solution.
//

class group extends Thread
{
   private int num_permutations;    // Number of permutations in
                                       subpopulation
   private int num_populations;     // Number of subpopulations
   private permutation[] pop;       // permutations in this
                                       subpopulation
   private permutation[] newpop;    // permutations in next generation
   permutation father;              // Permutation chosen for
                                       crossover
   permutation mother;              // Permutation chosen for
                                       crossover
   private decoder Decoder;         // Decoder
   private table Table;             // Table
   private group_GUI display;       // GUI for this subpopulation
   private group_stats stats;       // Statistics for this
                                       subpopulation
   private global_stats global;     // Global Statistics
   private TextArea log;            // Area for displaying of
                                       statistics
   private int select_strategy;     // Selection Strategy for this
                                       subpopulation
   private int cross_strategy;      // Crossover Strategy for this
                                       subpopulation
   private double crossover_rate;   // Value of Crossover Rate
   private double mutation_rate;    // Value of Mutation Rate
   private boolean elitism;         // Elitism option
   private int frequency;           // generations between successive
                                       migrations
   private int num_migrations;      // Number of migrations performed
                                       so far
   private boolean migrate_ready;   // True when ready to perform
                                       migration
   private int index;
  // Temporary variable


   // ** Constructor ** //
   //
   // Builds a new group of n_permutations random permutations
   //

   public group(int n_permutations, int num_vertices, decoder d,
  table t, group_GUI gui, group_stats my_stats, TextArea guilog, int
```

class group

```
import java.awt.TextArea;
import selection.*;
import crossover.*;
import mutation.*;


// ** group ** //
//
// A group contains permutations that are evolved towards an
// optimal solution.
//

class group extends Thread
{
   private int num_permutations;    // Number of permutations in
                                       subpopulation
   private int num_populations;     // Number of subpopulations
   private permutation[] pop;       // permutations in this
                                       subpopulation
   private permutation[] newpop;    // permutations in next generation
   permutation father;              // Permutation chosen for
                                       crossover
   permutation mother;              // Permutation chosen for
                                       crossover
   private decoder Decoder;         // Decoder
   private table Table;             // Table
   private group_GUI display;       // GUI for this subpopulation
   private group_stats stats;       // Statistics for this
                                       subpopulation
   private global_stats global;     // Global Statistics
   private TextArea log;            // Area for displaying of
                                       statistics
   private int select_strategy;     // Selection Strategy for this
                                       subpopulation
   private int cross_strategy;      // Crossover Strategy for this
                                       subpopulation
   private double crossover_rate;   // Value of Crossover Rate
   private double mutation_rate;    // Value of Mutation Rate
   private boolean elitism;         // Elitism option
   private int frequency;           // generations between successive
                                       migrations
   private int num_migrations;      // Number of migrations performed
                                       so far
   private boolean migrate_ready;   // True when ready to perform
                                       migration
   private int index;
  // Temporary variable


   // ** Constructor ** //
   //
   // Builds a new group of n_permutations random permutations
   //

   public group(int n_permutations, int num_vertices, decoder d,
  table t, group_GUI gui, group_stats my_stats, TextArea guilog, int
```

```
select, double crossrate, double mutrate, boolean elite, int
cross, int freq, int numpopulations, global_stats gs)
  {
    num_permutations = n_permutations;
    pop = new permutation[num_permutations];
    newpop = new permutation[num_permutations];
    Decoder = d;
    Table = t;
    display = gui;
    stats = my_stats;
    log = guilog;
    select_strategy = select;
    cross_strategy = cross;
    crossover_rate = crossrate;
    mutation_rate = mutrate;
    elitism = elite;
    frequency = freq;
    num_migrations = 0;
    migrate_ready = false;
    num_populations = numpopulations;
    global = gs;
    index = 0;

    // Create random permutations

    for(int i=0; i < num_permutations; i++)
     {
      pop[i] = new permutation(num_vertices);
      newpop[i] = new permutation(num_vertices);
     }
  }



 // ** evolve ** //
 //
 // Uses an object of class Decoder to assign fitness values to
 // each permutation in this subpopulation. It also uses procedures
 // in packages "crossover", "mutation", and "selection" to perform
 // genetic operations on the permutations.
 //

 public synchronized void evolve()
 {
   while(true)
   {

     // Use the decoder to assign a fitness value to each
// permutation in the current generation

     for(int i=0; i < num_permutations; i++)
     {
       if (!pop[i].isdecoded())
         Decoder.decode(pop[i],Table);
       pop[i].set_decoded(true);
     }

     // Display statistics for this generation

     stats.do_stats(pop, num_permutations);

     if(stats.get_prev_best_fitness() < stats.get_best_fitness())
       display.set_graph(stats.get_best(),Table);

     // Update global statistics if necessary
```

```
      if(stats.get_best_fitness() > global.get_best_fitness())
        global.set_best(stats.get_best(),
stats.get_best_fitness(),Table);

      // If elitism is enabled, transfer the best permutation to the
      // new population

      if(elitism)
      {
        newpop[0] = stats.get_best();
        index = 1;
      }
      else
        index = 0;

      // Create the new population

      while(index < num_permutations)
      {
        switch(select_strategy)
        {
          case 1: // Roulette Wheel
            father = pop[selection.Roulette.select(pop,
num_permutations)];
            break;

        } // Selection of first parent is done

        // If crossover needs to be performed

        if (Math.random() <= crossover_rate)
        {
          switch(select_strategy)
          {
            case 1: // Roulette Wheel
              mother = pop[selection.Roulette.select(pop,
num_permutations)];
              break;
          }  // Selection of second parent is done


          // Perform crossover according to the crossover strategy

          switch(cross_strategy)
          {
            case 1:  // PMX
              if (index == (num_permutations - 1))
               {

crossover.pmx.mate(father,mother,newpop[index],newpop[index]);

                // Perform mutation according to the mutation
                // strategy

                if (Math.random() <= mutation_rate)
                  mutation.swap.mutate(newpop[index]);
               }
              else
               {

crossover.pmx.mate(father,mother,newpop[index],newpop[index+1]);

                // Perform mutation according to the mutation
// strategy
```

```
                if (Math.random() <= mutation_rate)
                 {
                   mutation.swap.mutate(newpop[index]);
                   mutation.swap.mutate(newpop[index+1]);
                 }
               }

               index++;
               index++;
               break;
           }
        } // Crossover is done
        else
         {
           // Directly transfer the parent without doing crossover

           newpop[index] = father;
           index++;
         }

      } // New population is created

      // Exchange the old population with the new population

      pop = newpop;

      // Increment the generation number

      stats.increment_generation();

      // Check if the migration condition is satisfied

   if(num_populations > 1)
      if (stats.get_num_generations()
==(num_migrations*frequency+frequency))
        {
          // Signal to GAmigration

          migrate_ready = true;

          // Stop the evolution of this population

          this.suspend();

          // Increment the number of migrations and reset the
// migration signal

          num_migrations++;
          migrate_ready = false;
        }

   } // while

  } // End of evolve

 // ** ready ** //
 //
 // Indicates if this subpopulation is ready to perform migration
 //

 public boolean ready() { return migrate_ready; }

 // ** set_ready ** //
 //
 // Sets "migrate_ready" to indicate this subpopulation is ready to
 // perform migration
```

```
    //

    public void set_ready(boolean value) { migrate_ready = value; }


    // ** get_popsize ** //
    //
    // Returns number of permutations in this subpopulation
    //

    public int get_popsize() { return num_permutations; }

    // ** get_pop ** //
    //
    // Returns permutations in this subpopulation
    //

    public permutation[] get_pop() { return pop; }

    // ** get_permutation ** //
    //
    // Returns a permutation in this subpopulation
    //

    public permutation get_permutation(int index) { return pop[index];
   }

    // ** set_permutation ** //
    //
    // Inserts a new permutation in this subpopulation
    //

    public void set_permutation(permutation p, int index) { pop[index]
   = p; }

    // ** run ** //
    //
    // Invokes the method "evolve"
    //

    public void run()
    {
      evolve();
    }

} // End of class group
```

class group GUI

```
import java.awt.*;
import util.*;

// ** group_GUI ** //
//
// Component to display the permutations of a group in a graphical
  // manner
//
```

```
class group_GUI extends Canvas
{
  private Dimension size;
  private permutation p;  // Permutation of graph to be displayed
  private int x;          // x-coordinate of graph's position
  private int y;          // y-coordinate of graph's position
  private table t;
  private boolean pset;

  // ** Constructor ** //

  public group_GUI(int d1, int d2, int x_coord, int y_coord)
  {
    size = new Dimension(d1,d2);
    pset = false;
    x = x_coord;
    y = y_coord;
  }

  // ** set_graph ** //
  //
  // Set graph of permutation to be displayed
  //

  public synchronized void set_graph(permutation permu, table T)
  {
    p = permu;
    t = T;
    pset = true;
    update(this.getGraphics());
  }

  // ** set_position ** //
  //
  // Set position to display graph of permutation
  //

  public synchronized void set_position(int x_coord, int y_coord)
  { x = x_coord; y = y_coord; }

  // ** paint ** //
  //
  // Display graph of permutation
  //

  public void paint(Graphics g)
  {
    if (pset)
      util.graph.draw_graph(g,p,t,x,y);
  }

  public Dimension minimumSize() { return size; }

  public Dimension preferredSize() { return minimumSize(); }

} // End of class group_GUI
```

class group_stats

```
import java.awt.*;
import util.*;

// ** group_stats ** //
//
// Stores and graphically displays statistical information for
// each population.
//

class group_stats extends Panel
{
   private permutation best;               // Permutation with highest
                                              fitness value
   private int num_generations;            // Number of generations
   private double best_fitness;            // Fitness value of best
                                              permutation
   private double average_fitness;         // Average population
                                              fitness value
   private double av_fitness_change;       // Change on Average
  population fitness from previous generation
   private double best_fitness_change;     // Change on the value of
  best_fitness from previous generation
   private int best_generation;            // Generation with highest
  average fitness value
   private double best_generation_fitness;  // Average fitness value
  of best generation
   private double prev_best_fitness;          // Best fitness of
  previous generation

   private Label Name;
   private Label label1;
   private Label label2;
   private Label label3;
   private Label label4;
   private Label label5;

   private TextField numgenerations;
   private TextField bestfitness;
   private TextField avfitness;
   private TextField avchange;

   // ** Constructor ** //

   public group_stats(String pop_name)
   {
     num_generations = 0;
     best_fitness = 0;
     average_fitness = 0;
     av_fitness_change = 0;
     best_fitness_change = 0;
     best_generation = 0;
     best_generation_fitness = 0;

     Name = new Label(pop_name);
     label1 = new Label("gen # ");
     label2 = new Label("best f");
     label3 = new Label("av.  f");
     label4 = new Label("change");
```

```
   label5 = new Label("");

   numgenerations = new TextField(3);
   numgenerations.setEditable(false);
   bestfitness = new TextField(3);
   bestfitness.setEditable(false);
   avfitness = new TextField(3);
   avfitness.setEditable(false);
   avchange = new TextField(3);
   avchange.setEditable(false);

   setLayout(new GridLayout(5,2));
   setBackground(Color.yellow);

   add(Name);
   add(label5);
   add(label1);
   add(numgenerations);
   add(label2);
   add(bestfitness);
   add(label3);
   add(avfitness);
   add(label4);
   add(avchange);

}

// ** set_best ** //
//
// Records the permutation with the highest fitness value
//

public void set_best(permutation p)
{
  best = p;
  bestfitness.setText(Double.toString(best_fitness));
}

// ** get_best ** //
//
// Returns the permutation with the highest fitness value
//

public permutation get_best() { return best; }

// ** get_best_fitness ** //
//
// Returns the fitness value of the permutation with highest
// fitness value
//

public double get_best_fitness() { return best_fitness; }


// ** set_best_fitness ** //
//
// Records fitness value of permutation with highest fitness value
//

public void set_best_fitness(double fitness) { best_fitness =
fitness; }


// ** set_average_fitness ** //
//
// Sets the value of the average fitness
```

```
//

public void set_average_fitness(double average)
{
  average_fitness = average;
  avfitness.setText(Double.toString(average_fitness));
}

// ** set_best_generation ** //
//
// Records the generation with the highest average fitness value
//

public void set_best_generation(int generation) { best_generation
= generation; }

// ** set_num_generations ** //
//
// Records the number of generations in this population
//

public void set_num_generations(int number)
{
  num_generations = number;
  numgenerations.setText(Integer.toString(num_generations));
}

// ** get_num_generations ** //
//
// Returns the number of generations in this population
//

public int get_num_generations() { return num_generations; }

// ** increment_generation ** //
//
// Increments the number of generations in this population
//

public void increment_generation() { num_generations++; }

// ** do_stats ** //
//
// Performs local statistics
//

public void do_stats(permutation[] pop, int num_permutations)
{
  double total_fitness = 0;                // Sum of all fitness
values
  double prev_av_fitness = average_fitness; // Average fitness of
previous generation
  prev_best_fitness = best_fitness;

  // Find a permutation with the highest fitness value and
  // calculate the sum of all fitness values

  best_fitness = 0;

  for(int i=0; i < num_permutations; i++)
    {
      total_fitness += pop[i].getfitval();

      if (pop[i].getfitval() > best_fitness)
       {
         best_fitness = pop[i].getfitval();
```

```
            best = pop[i];
          }
        }

      // Find the new average fitness and the change in
   // average_fitness and best_fitness with respect to the        //
   previous generation

      average_fitness = total_fitness / num_permutations;

      best_fitness_change = best_fitness - prev_best_fitness;
      av_fitness_change = average_fitness - prev_av_fitness;

      // Change the best generation, if necessary

      if(average_fitness > best_generation_fitness)
       {
         best_generation = num_generations;
         best_generation_fitness = average_fitness;
       }

      // Update the graphical interface

      numgenerations.setText(Integer.toString(num_generations));
      bestfitness.setText(Double.toString(best_fitness));
      avfitness.setText(Double.toString(average_fitness));
      avchange.setText(Double.toString(av_fitness_change));

    } // End of do_stats

    // ** get_prev_best_fitness ** //
    //
    // Returns the fitness value of the previously best permutation
    //

    public double get_prev_best_fitness() { return prev_best_fitness;
    }
} // End of group_stats
```

class Permutation

```
import java.awt.TextArea;


public class permutation
{
   private int num_vertices;  // Number of vertices in graph
  corresponding to given permutation
   private int num_edges;     // Number of edges in the permutation
   private int[] permu;       // Contains the permutation of
  num_edges
   private int[] color;       // Contains the color of each edge in
  the permutation
   private double fitness;    // Fitness value for the permutation
   private boolean decoded;   // Indicates that this permutation has
  already been assigned a fitness value

   // ** Constructor ** //
   //
   // Builds a random permutation according to the number of vertices
   // in the graph
```

```
 //

 public permutation(int num_vert)
 {
      int index1, index2, temp;

      // Compute the number of edges in the complete graph with
// num_vertices

      num_vertices = num_vert;
      num_edges = (num_vertices * (num_vertices - 1)) / 2;

      // Create an array of size num_edges to hold the permutation
// of edges and an array of size num_edges to hold the      //
coloring of the edges in the permutation

      permu = new int[num_edges];
      color = new int[num_edges];

      // Initialize the edges of the permutation and the colors

      for(int i=0; i < num_edges; i++)
      {
         permu[i] = i;
         color[i] = -1;
      }

      // Initially the fitness value equals the number of edges in
// the permutation

      fitness = num_edges;

      // Randomly swap the edge numbers in the permutation to
// create a random permutation of edges

      for(int i=0; i < (num_edges/2); i++)
      {
         index1 = (int) (Math.random() * num_edges);
         index2 = (int) (Math.random() * num_edges);

         temp = permu[index2];
         permu[index2] = permu[index1];
         permu[index1] = temp;
      }

      // Indicate that this permutation has not been decoded yet

      decoded = false;

  } // End of constructor

// ** getnumvert ** //
//
// Returns value of parameter Number Of Vertices
//

public int getnumvert() { return num_vertices; }

// ** getnumedge ** //
//
// Returns value of parameter Number Of Edges
//

public int getnumedge() { return num_edges; }

// ** getfitval ** //
```

```
//
// Returns fitness value of a permutation
//

public double getfitval() { return fitness; }

// ** setfitval ** //
//
// Sets the fitness value of a permutation
//

public void setfitval(double fit_val) { fitness = fit_val; }

// ** getedge ** //
//
// Returns an edge in a permutation
//

public int getedge(int index) { return permu[index]; }

// ** setedge ** //
//
// Sets an edge of a permutation
//

public void setedge(int index, int value) {permu[index] = value; }


// ** getcolor ** //
//
// Returns color of an edge in a permutation
//

public int getcolor(int index) { return color[index]; }

// ** setcolor ** //
//
// Sets color of an edge in a permutation
//

public void setcolor(int edge, int col) { color[edge] = col; }

// ** isdecoded ** //
//
// Returns true if a permutation has already been decoded
//

public boolean isdecoded() { return decoded; }

// ** set_decoded ** //
//
// Sets "decoded" when a permutation is decoded
//

public void set_decoded(boolean value) { decoded = value; }

// ** print_edges ** //
//
// Displays the edges of a permutation on a TextArea object
//

public void print_edges(TextArea paper)
{
  for(int i=0; i < num_edges; i++)
    paper.append(permu[i] + ",");
  paper.append("\n");
```

```
    }


    // ** print_colors ** //
    //
    // Displays the color of all the edges of a permutation on a
    // TextArea object
    //

    public void print_colors(TextArea paper)
    {
      for(int i=0; i < num_edges; i++)
        paper.append(color[i] + ",");
      paper.append("\n");
    }

} // End of class permutatio
```

class Ramsey

```
import java.applet.Applet;
import java.awt.*;

//      //
//** Main driver **//
//      //

public class Ramsey extends Applet
{

  private int num_populations;     // Number of populations
  private int pop_size;            // Number of permutation in a
   population
  private int num_colors;          // Number of colors used to draw
   graph edges
  private int num_vertices;        // Number of vertices in complete
   graph
  private int num_sub_vertices;    // Number of vertices in complete
   subgraph
  private int select_strategy;     // Represents selection strategy
  private int mig_selection;       // Represents migrant selection
   strategy
  private int mig_topology;        // Represents migration topology
  private int cross_strategy;      // Represents crossover strategy
  private double crossover_rate;   // Crossover rate (between 0 and 1)
  private double mutation_rate;    // Mutation rate (between 0 and 1)
  private int mig_frequency;       // Number of generations between
   migrations
  private int mig_size;            // Number of permutations in a
   migration
  private boolean elitism;         // Option to enable/disable elitism

  private decoder D;               // Decoder
  private table T;                 // Table

  private boolean numpop_set;      // True when num_populations is set
  private boolean popsize_set;     // True when pop_size is set
  private boolean numcolors_set;   // True when num_colors is set
  private boolean num_vert_set;    // True when num_vertices is set
```

```
private boolean select_set;     // True when select_strategy is set
private boolean mig_select_set; // True when mig_selection is set
private boolean mig_top_set;    // True when mig_topology is set
private boolean cross_set;      // True when cross_strategy is set
private boolean cross_rate_set; // True when crossover_rate is set
private boolean mut_rate_set;   // True when mutation_rate is set
private boolean mig_freq_set;   // True when mig_frequency is set
private boolean mig_size_set;   // True when mig_size is set
private boolean elitism_set;    // True when elitism option is
checked

private group island1;          // Thread to run on first
population
private group island2;          // Thread to run on second
population
private group island3;          // Thread to run on third
population
private group island4;          // Thread to run on fourth
population
private group island5;          // Thread to run on fifth
population
private group island6;          // Thread to run on sixth
population

private group_GUI island1_gui;  // GUI for first population
private group_GUI island2_gui;  // GUI for secod population
private group_GUI island3_gui;  // GUI for third population
private group_GUI island4_gui;  // GUI for fourth population
private group_GUI island5_gui;  // GUI for fifth population
private group_GUI island6_gui;  // GUI for sixth population

private group_stats island1_stats;  // Stats for first population
private group_stats island2_stats;  // Stats for second population
private group_stats island3_stats;  // Stats for third population
private group_stats island4_stats;  // Stats for fourth population
private group_stats island5_stats;  // Stats for fifth population
private group_stats island6_stats;  // Stats for sixth population


// Declaration of constants

final public static int NUMPOP = 1;
final public static int POPSIZE = 2;
final public static int NUMCOL = 3;
final public static int NUMVERT = 4;
final public static int START = 5;
final public static int SELECT = 6;
final public static int CROSSRATE = 7;
final public static int MUTRATE = 8;
final public static int ELITISM = 9;
final public static int CROSSOVER = 10;
final public static int PAUSE = 11;
final public static int RESUME = 12;
final public static int STOP = 13;
final public static int RESET = 14;
final public static int MIGFREQUENCY = 15;
final public static int MIGSIZE = 16;
final public static int MIGSELECTION = 17;
final public static int MIGTOPOLOGY = 18;
final public static int LAUNCH = 19;

// Create a GAException object

private GAException e = new GAException();

// Create a Gamigration object
```

```
private GAmigration migration;

// Create a global_stats object

private global_stats global;
// Create a group_GUI object

private group_GUI global_graph = new group_GUI(100,100,100,100);

// Create a graphical user interface

Ramsey_GUI gui = new Ramsey_GUI(this);


// ** init ** //
//
// Sets all parameters to their default values
//

public void init()
{
  num_populations = 0;
  pop_size = 0;
  num_vertices = 0;
  num_sub_vertices = 3;
  num_colors = 0;
  crossover_rate = 0;
  mutation_rate = 0;
  popsize_set = false;
  numpop_set = false;
  numcolors_set = false;
  num_vert_set = false;
  select_set = false;
  mig_select_set = false;
  mig_top_set = false;
  cross_rate_set = false;
  cross_set = false;
  mut_rate_set = false;
  select_strategy = 0;
  cross_strategy = 0;
  elitism_set = false;
  elitism = true;
  gui.init();
}

// ** set_popsize ** //
//
// Sets value for the Population Size parameter
//

public void set_popsize(int popsize)
{
  try{
    pop_size = popsize;
    popsize_set = true;

    if (mig_size_set && (pop_size < mig_size)) throw e;
     }
  catch(GAException e){
    e.report(gui.log, "Population size >= Migration Size.\n
Migration Size has been reset");
    mig_size_set = false; }
}

// ** set_numpopulations ** //
```

```
//
// Sets value for the Number Of Populations parameter
// Creates a GUI for each subpopulation
//

public void set_numpopulations(int numpop)
{
  gui.clear();

  try{
    num_populations = numpop;

    island1_gui = new group_GUI(50,50,40,40);
    island1_stats = new group_stats("POP1");
    gui.add_population(island1_gui);
    gui.add_stats(island1_stats);

    if (num_populations >= 2) {
     island2_gui = new group_GUI(50,50,40,40);
     island2_stats = new group_stats("POP2");
     gui.add_population(island2_gui);
     gui.add_stats(island2_stats);}

    if (num_populations >= 3) {
     island3_gui = new group_GUI(50,50,40,40);
     island3_stats = new group_stats("POP3");
     gui.add_population(island3_gui);
     gui.add_stats(island3_stats);}

    if (num_populations >= 4) {
     island4_gui = new group_GUI(50,50,40,40);
     island4_stats = new group_stats("POP4");
     gui.add_population(island4_gui);
     gui.add_stats(island4_stats);}

    if(num_populations >= 5) {
     island5_gui = new group_GUI(50,50,40,40);
     island5_stats = new group_stats("POP5");
     gui.add_population(island5_gui);
     gui.add_stats(island5_stats);}

    if (num_populations == 6) {
     island6_gui = new group_GUI(50,50,40,40);
     island6_stats = new group_stats("POP6");
     gui.add_population(island6_gui);
     gui.add_stats(island6_stats);}

     gui.window.paintComponents(gui.window.getGraphics());
     this.paintComponents(this.getGraphics());

    numpop_set = true;

    if((num_populations == 1) && (mig_freq_set || mig_size_set ||
mig_select_set || mig_top_set)) throw e;
    }
  catch(GAException e){
    e.report(gui.log, "No migration with #populations = 1.\n
Migration parameters have been reset");
    mig_freq_set = false; mig_size_set = false; mig_select_set =
false; mig_top_set = false;}
}

// ** set_numcolors ** //
//
// Sets the Number Of Colors parameter
//
```

```
public void set_numcolors(int numcolors)
{
  num_colors = numcolors;
  numcolors_set = true;
}

// ** set_num_vertices ** //
//
// Sets the Number Of Vertices parameter
//

public void set_num_vertices()
{
  try{
    num_vertices = Integer.parseInt(gui.num_vertices.getText());
    num_vert_set = true;
    if (num_vertices < 3) throw e;
     }
  catch(NumberFormatException a) {
    e.report(gui.log, "NumberFormatException");
    num_vert_set = false; }
  catch(GAException e) {
    e.report(gui.log, "Number of vertices must be >= 3");
    num_vert_set = false; }
}

// ** set_selection ** //
//
// Sets the Selection Strategy
//

public void set_selection(String strategy)
{
 if (strategy.equals("Roulette Wheel"))
   select_strategy = 1;
 select_set = true;
}

// ** set_mig_selection ** //
//
// Sets the Migrant Selection Strategy
//

public void set_mig_selection(String strategy)
{
 try{
   if (strategy.equals("Random"))
     mig_selection = 1;
   if (strategy.equals("Roulette Wheel"))
     mig_selection = 2;
   mig_select_set = true;

   if ((!numpop_set) || (num_populations == 1)) throw e;
    }
 catch(GAException e){
   e.report(gui.log, "Number of populations must be > 1 for
migration");
   mig_select_set = false; }
}

// ** set_mig_topology ** //
//
// Sets the Migration Topology
//
```

```
public void set_mig_topology(String topology)
{
 try{
   if (topology.equals("Ring"))
     mig_topology = 1;

   mig_top_set = true;

   if ((!numpop_set) || (num_populations == 1)) throw e;
    }
 catch(GAException e){
   e.report(gui.log, "Number of populations must be > 1 for
migration");
   mig_top_set = false; }
}

// ** set_crossover ** //
//
// Sets the Crossover Strategy
//

public void set_crossover(String strategy)
{
 if (strategy.equals("PMX"))
   cross_strategy = 1;
 cross_set = true;
}

// ** set_crossover_rate ** //
//
// Sets value for the Crossover Rate parameter
//

public void set_crossover_rate()
{
  try{
    Double temp = Double.valueOf(gui.crossover_rate.getText());
    crossover_rate = temp.doubleValue();
    cross_rate_set = true;
    if ((crossover_rate < 0) || (crossover_rate > 1)) throw e;
     }
  catch(NumberFormatException a) {
    e.report(gui.log, "NumberFormatException");
    cross_rate_set = false; }
  catch(GAException e) {
    e.report(gui.log, "Crossover rate must be between 0 and 1");
    cross_rate_set = false; }
}

// ** set_mutation_rate ** //
//
// Sets value for the Mutation Rate parameter
//

public void set_mutation_rate()
{
  try{
    Double temp = Double.valueOf(gui.mutation_rate.getText());
    mutation_rate = temp.doubleValue();
    mut_rate_set = true;
    if ((mutation_rate < 0) || (mutation_rate > 1)) throw e;
     }
  catch(NumberFormatException a) {
    e.report(gui.log, "NumberFormatException");
    mut_rate_set = false; }
  catch(GAException e) {
```

```
    e.report(gui.log, "Mutation rate must be between 0 and 1");
    mut_rate_set = false; }
}

// ** set_mig_frequency ** //
//
// Sets value for the Migration Frequence parameter
//

public void set_mig_frequency()
{
  try{
    mig_frequency =
Integer.parseInt(gui.migration_frequency.getText());
    mig_freq_set = true;
    if (mig_frequency < 1) throw e;
    if ((!numpop_set) || (num_populations == 1)) throw e;
     }
  catch(NumberFormatException a) {
    e.report(gui.log, "NumberFormatException");
    mig_freq_set = false; }
  catch(GAException e) {
    e.report(gui.log, "Migration Frequency must be >= 1\n or number
of populations is not > 1");
    mig_freq_set = false; }
}

// ** set_mig_size ** //
//
// Sets value for the Migration Size parameter
//

public void set_mig_size()
{
  try{
    mig_size = Integer.parseInt(gui.migration_size.getText());
    mig_size_set = true;
    if ((mig_size < 1) || (!popsize_set) || (mig_size > pop_size))
throw e;
    if ((!numpop_set) || (num_populations == 1)) throw e;
     }
  catch(NumberFormatException a) {
    e.report(gui.log, "NumberFormatException");
    mig_size_set = false; }
  catch(GAException e) {
    e.report(gui.log, "Migration Size must be between 1 and
population size\n or population size has not been set or number of
populations is not > 1");
    mig_size_set = false; }
}

// ** set_elitism ** //
//
// Sets the Elitism parameter
//

public void set_elitism(boolean on_off)
{
  elitism = on_off;
  elitism_set = true;
}

// ** initialize ** //
//
// Initializes all parameters and creates subpopulations
//
```

```
public boolean initialize()
{
    // Create the Decoder and the Table

    D = new decoder(num_sub_vertices,num_colors);
    T = new table(num_vertices);

    // Create the global statistics handler

    global = new global_stats(num_populations, num_vertices,
global_graph);
    gui.add_global_gui(global_graph);
    gui.add_global_stats(global);
    gui.window.paintComponents(gui.window.getGraphics());

    // Create and Initialize the populations and their graphical
interfaces

    island1 = new group(pop_size, num_vertices, D, T, island1_gui,
                        island1_stats, gui.log, select_strategy,
                        crossover_rate, mutation_rate, elitism,
                        cross_strategy, mig_frequency,
num_populations, global);

    if (num_populations >= 2)
    island2 = new group(pop_size, num_vertices, D, T, island2_gui,
                        island2_stats, gui.log, select_strategy,
                        crossover_rate, mutation_rate, elitism,
cross_strategy, mig_frequency, num_populations, global);

    if (num_populations >= 3)
    island3 = new group(pop_size, num_vertices, D, T, island3_gui,
                        island3_stats, gui.log, select_strategy,
                        crossover_rate, mutation_rate, elitism,
                        cross_strategy, mig_frequency,
num_populations, global);

    if (num_populations >= 4)
    island4 = new group(pop_size, num_vertices, D, T, island4_gui,
                        island4_stats, gui.log, select_strategy,
                        crossover_rate, mutation_rate, elitism,
                        cross_strategy, mig_frequency,
num_populations, global);

    if(num_populations >= 5)
    island5 = new group(pop_size, num_vertices, D, T, island5_gui,
                        island5_stats, gui.log, select_strategy,
                        crossover_rate, mutation_rate, elitism,
                        cross_strategy, mig_frequency,
num_populations, global);

    if (num_populations == 6)
    island6 = new group(pop_size, num_vertices, D, T, island6_gui,
                        island6_stats, gui.log, select_strategy,
                        crossover_rate, mutation_rate, elitism,
                        cross_strategy, mig_frequency,
num_populations, global);

    if (num_populations > 1)
    migration = new GAmigration(num_populations, mig_topology,
                                mig_size, mig_selection, gui.log,
island1, island2, island3, island4, island5, island6,
island1_stats, island2_stats, island3_stats, island4_stats,
island5_stats, island6_stats);
```

```
  return true;

} // End of initialize


// ** evolve ** //
//
// Starts the threas of all subpopulations and the thread of
// Gamigration
//

public void evolve()
{
  // Check that all necessary parameters have been set

  if(num_populations == 1)
  {
    mig_freq_set = true;
    mig_size_set = true;
    mig_select_set = true;
    mig_top_set = true;
  }

  if(popsize_set && numpop_set && numcolors_set && num_vert_set
     && select_set && cross_rate_set && mut_rate_set && elitism_set
     && cross_set && mig_freq_set && mig_size_set && mig_select_set
     && mig_top_set)
  {
    initialize();

    island1.start();

    if (num_populations >= 2)
      island2.start();
    if (num_populations >= 3)
      island3.start();
    if (num_populations >= 4)
      island4.start();
    if (num_populations >= 5)
      island5.start();
    if (num_populations == 6)
      island6.start();

    if (num_populations > 1)
      migration.start();
  }
  else
   {
     if(!popsize_set) e.report(gui.log, "Population size has not
been set");
     if(!numpop_set) e.report(gui.log, "Number of populations has
not been set");
     if(!numcolors_set) e.report(gui.log, "Number of colors has not
been set");
     if(!num_vert_set) e.report(gui.log, "Number of vertices has
not been set");
     if(!select_set) e.report(gui.log, "Selection strategy has not
been set");
     if(!cross_rate_set) e.report(gui.log, "Crossover rate has not
been set");
     if(!mut_rate_set) e.report(gui.log, "Mutation rate has not
been set");
     if(!elitism_set) e.report(gui.log, "Elitism option has not
been set");
     if(!cross_set) e.report(gui.log, "Crossover strategy has not
been set");
```

```
     if(!mig_freq_set) e.report(gui.log, "Migration Frequency has
not been set");
     if(!mig_size_set) e.report(gui.log, "Migration Size has not
been set");
     if(!mig_select_set) e.report(gui.log, "Migrant Selection
Strategy has not been set");
     if(!mig_top_set) e.report(gui.log, "Migration Topology has not
been set");
   }

} // End of evolve

// ** GApause ** //
//
// Temporarily pause the execution of the genetic algorithm
//

public void GApause()
{
  island1.suspend();

  if (num_populations >= 2)
    island2.suspend();
  if (num_populations >= 3)
    island3.suspend();
  if (num_populations >= 4)
    island4.suspend();
  if (num_populations >= 5)
    island5.suspend();
  if (num_populations == 6)
    island6.suspend();

} // End of pause


// ** GAresume ** //
//
// Restart the execution of the genetic algorithm
//

public void GAresume()
{
  island1.resume();

  if (num_populations >= 2)
    island2.resume();
  if (num_populations >= 3)
    island3.resume();
  if (num_populations >= 4)
    island4.resume();
  if (num_populations >= 5)
    island5.resume();
  if (num_populations == 6)
    island6.resume();

} // End of GAresume


// ** GAstop ** //
//
// Stop the execution of the genetic algorithm
//

public void GAstop()
{
  GAresume();
```

```
    island1.stop();

    if (num_populations >= 2)
      island2.stop();
    if (num_populations >= 3)
      island3.stop();
    if (num_populations >= 4)
      island4.stop();
    if (num_populations >= 5)
      island5.stop();
    if (num_populations == 6)
      island6.stop();

    gui.window.dispose();

  } // End of stop


  //** GAreset **//
  //
  // Reset the genetic algorithm.
  // Clear the graphical user interface and reset all parameters.
  //

  public void GAreset()
  {
    stop();
    gui.clear_all();
    init();
  } // End of reset


  // ** display_window ** //

  public void display_window() { gui.window.show(); }

} // class Ramsey
```

class Ramsey Action Listener

```
import java.awt.event.*;

//
// ** Ramsey_Action_Listener **
//
// Handles the Action events for num_vertices, crossover_rate,
// mutation_rate, start, pause, resume, stop, reset, mig_frequency,
// mig_size, and launch.
//

public class Ramsey_Action_Listener implements ActionListener
{
  private Ramsey applet;
  private int command;


  // ** Constructor ** //
```

```
  public Ramsey_Action_Listener(Ramsey ramsey_applet, int
   listening_command)
  {
    applet = ramsey_applet;
    command = listening_command;
  }

  // ** actionPerformed ** //
  //
  // Invokes a procedure in Ramsey_GUI depending on action performed
  //

  public void actionPerformed(ActionEvent action)
  {
    switch(command)
    {
      case Ramsey.NUMVERT:
        applet.set_num_vertices();
        break;

      case Ramsey.START:
        applet.evolve();
        break;

      case Ramsey.CROSSRATE:
        applet.set_crossover_rate();
        break;

      case Ramsey.MIGFREQUENCY:
        applet.set_mig_frequency();
        break;

      case Ramsey.MIGSIZE:
        applet.set_mig_size();
        break;

      case Ramsey.MUTRATE:
        applet.set_mutation_rate();
        break;

      case Ramsey.PAUSE:
        applet.GApause();
        break;

      case Ramsey.RESUME:
        applet.GAresume();
        break;

      case Ramsey.STOP:
        applet.GAstop();
        break;

      case Ramsey.RESET:
        applet.GAreset();
        break;

      case Ramsey.LAUNCH:
        applet.display_window();
        break;
    }

  } // End of actionPerformed

} // End of Ramsey_Action_Listener
```

class Ramsey_GUI

```
import java.awt.*;
import java.awt.LayoutManager;

//
// ** Ramsey_GUI ** //
//
// Implements the applet's graphical user interface and
// connects events to listeners in class Ramsey_Listener
//

class Ramsey_GUI
{
   private Ramsey applet;
   private Ramsey_Item_Listener Item_Listener1;
   private Ramsey_Item_Listener Item_Listener2;
   private Ramsey_Item_Listener Item_Listener3;
   private Ramsey_Item_Listener Item_Listener4;
   private Ramsey_Item_Listener Item_Listener5;
   private Ramsey_Item_Listener Item_Listener6;
   private Ramsey_Item_Listener Item_Listener7;
   private Ramsey_Item_Listener Item_Listener8;
   private Ramsey_Action_Listener Action_Listener1;
   private Ramsey_Action_Listener Action_Listener2;
   private Ramsey_Action_Listener Action_Listener3;
   private Ramsey_Action_Listener Action_Listener4;
   private Ramsey_Action_Listener Action_Listener5;
   private Ramsey_Action_Listener Action_Listener6;
   private Ramsey_Action_Listener Action_Listener7;
   private Ramsey_Action_Listener Action_Listener8;
   private Ramsey_Action_Listener Action_Listener9;
   private Ramsey_Action_Listener Action_Listener10;
   private Ramsey_Action_Listener Action_Listener11;
   private GridLayout grid;
   private Panel panel1;
   private Panel panel2;
   public Panel panel11;
   private Panel panel12;
   private Panel panel21;
   private Panel panel22;
   private Panel panel23;
   private Choice num_populations;
   private Choice pop_size;
   private Choice num_colors;
   private Choice selection;
   private Choice crossover;
   private Choice migration_topology;
   private Choice migrant_selection;
   private Checkbox elitism;
   public TextField num_vertices;
   public TextField crossover_rate;
   public TextField mutation_rate;
   public TextField migration_frequency;
   public TextField migration_size;
   public TextArea log;
   private Button start;
   private Button pause;
   private Button resume;
   private Button stop;
```

```
private Button reset;
private Button launch;
private Label label1;
private Label label2;
private Label label3;
private Label label4;
private Label label5;
private Label label6;
private Label label7;
private Label label8;
private Label label9;
private Label label10;
private Label label11;
private Label label12;

public Frame window;

// ** Constructor ** //

public Ramsey_GUI(Ramsey ramsey_applet)
{
  applet = ramsey_applet;
  Item_Listener1 = new Ramsey_Item_Listener(applet,Ramsey.NUMPOP);
  Item_Listener2 = new
Ramsey_Item_Listener(applet,Ramsey.POPSIZE);
  Item_Listener3 = new Ramsey_Item_Listener(applet,Ramsey.NUMCOL);
  Item_Listener4 = new Ramsey_Item_Listener(applet,Ramsey.SELECT);
  Item_Listener5 = new
Ramsey_Item_Listener(applet,Ramsey.ELITISM);
  Item_Listener6 = new
Ramsey_Item_Listener(applet,Ramsey.CROSSOVER);
  Item_Listener7 = new
Ramsey_Item_Listener(applet,Ramsey.MIGSELECTION);
  Item_Listener8 = new
Ramsey_Item_Listener(applet,Ramsey.MIGTOPOLOGY);
  Action_Listener1 = new
Ramsey_Action_Listener(applet,Ramsey.NUMVERT);
  Action_Listener2 = new
Ramsey_Action_Listener(applet,Ramsey.START);
  Action_Listener3 = new
Ramsey_Action_Listener(applet,Ramsey.CROSSRATE);
  Action_Listener4 = new
Ramsey_Action_Listener(applet,Ramsey.MUTRATE);
  Action_Listener5 = new
Ramsey_Action_Listener(applet,Ramsey.PAUSE);
  Action_Listener6 = new
Ramsey_Action_Listener(applet,Ramsey.RESUME);
  Action_Listener7 = new
Ramsey_Action_Listener(applet,Ramsey.STOP);
  Action_Listener8 = new
Ramsey_Action_Listener(applet,Ramsey.RESET);
  Action_Listener9 = new
Ramsey_Action_Listener(applet,Ramsey.MIGFREQUENCY);
  Action_Listener10 = new
Ramsey_Action_Listener(applet,Ramsey.MIGSIZE);
  Action_Listener11 = new
Ramsey_Action_Listener(applet,Ramsey.LAUNCH);

  grid = new GridLayout(1,2);
  panel1 = new Panel();
  panel2 = new Panel();
  panel11 = new Panel();
  panel12 = new Panel();
  panel21 = new Panel();
  panel22 = new Panel();
  panel23 = new Panel();
```

```
    num_populations = new Choice();
    num_populations.addItemListener(Item_Listener1);
    pop_size = new Choice();
    pop_size.addItemListener(Item_Listener2);
    num_colors = new Choice();
    num_colors.addItemListener(Item_Listener3);
    selection = new Choice();
    selection.addItemListener(Item_Listener4);
    migration_topology = new Choice();
    migration_topology.addItemListener(Item_Listener8);
    migrant_selection = new Choice();
    migrant_selection.addItemListener(Item_Listener7);
    num_vertices = new TextField(3);
    num_vertices.addActionListener(Action_Listener1);
    log = new TextArea(41,32);
    log.setEditable(false);
    start = new Button("START");
    start.addActionListener(Action_Listener2);
    pause = new Button("PAUSE");
    pause.addActionListener(Action_Listener5);
    resume = new Button("RESUME");
    resume.addActionListener(Action_Listener6);
    stop = new Button("STOP");
    stop.addActionListener(Action_Listener7);
    reset = new Button("RESET");
    reset.addActionListener(Action_Listener8);
    crossover_rate = new TextField(3);
    crossover_rate.addActionListener(Action_Listener3);
    mutation_rate = new TextField(3);
    mutation_rate.addActionListener(Action_Listener4);
    elitism = new Checkbox("Elitism");
    elitism.addItemListener(Item_Listener5);
    crossover = new Choice();
    crossover.addItemListener(Item_Listener6);
    migration_frequency = new TextField(3);
    migration_frequency.addActionListener(Action_Listener9);
    migration_size = new TextField(3);
    migration_size.addActionListener(Action_Listener10);
    launch = new Button("LAUNCH SIPAGAR");
    launch.addActionListener(Action_Listener11);

    label1 = new Label("Number of Populations ");
    label2 = new Label("Population Size       ");
    label3 = new Label("Number Of Colors      ");
    label4 = new Label("Number of Vertices    ");
    label5 = new Label("Selection Strategy    ");
    label6 = new Label("Crossover Rate        ");
    label7 = new Label("Mutation Rate         ");
    label8 = new Label("Crossover Strategy    ");
    label9 = new Label("Migration Frequency   ");
    label10 = new Label("Migration Size       ");
    label11 = new Label("Migrant Selection    ");
    label12 = new Label("Migration Topology   ");

    window = new Frame("SIPAGAR (Simulated Parallel Genetic
Algorithm For Finding Ramsey Numbers)");

 } // End of Constructor

 // ** init ** //
 //
 // Initializes the GUI
 //

 public void init()
 {
```

```
  migration_topology.addItem("Ring");

  panel21.add(log);
  panel22.add(label1);
  panel23.add(num_populations);
  panel22.add(label2);
  panel23.add(pop_size);
  panel22.add(label3);
  panel23.add(num_colors);
  panel22.add(label4);
  panel23.add(num_vertices);
  panel22.add(label5);
  panel23.add(selection);
  panel22.add(label6);
  panel23.add(crossover_rate);
  panel22.add(label8);
  panel23.add(crossover);
  panel22.add(label7);
  panel23.add(mutation_rate);
  panel22.add(label9);
  panel23.add(migration_frequency);
  panel22.add(label10);
  panel23.add(migration_size);
  panel22.add(label11);
  panel23.add(migrant_selection);
  panel22.add(label12);
  panel23.add(migration_topology);
  panel23.add(elitism);
  panel23.add(start);
  panel23.add(pause);
  panel23.add(resume);
  panel23.add(stop);
  panel23.add(reset);

  window.setLayout(grid);
  window.add(panel1);
  window.add(panel2);
  window.resize(850,600);

  applet.add(launch);

} // End of init

// ** add_population ** //
//
// Adds a subpopulation to the GUI
//

public void add_population(group_GUI island)
{
  panel11.add(island);
} // End of add_population

// ** add_stats ** //
//
// Adds a subpopulation's local statistics to the GUI
//

public void add_stats(group_stats stats)
{
  panel11.add(stats);
} // End of add_stats

// ** add_global_stats ** //
//
// Adds global statistics to the GUI
```

```
    //

    public void add_global_stats(global_stats gstats)
    {
      panel12.add(gstats);
    } // End of add_global_stats

    // ** add_global_gui ** //
    //
    // Adds area to display globally best permutation to the GUI
    //

    public void add_global_gui(group_GUI g)
    {
      panel12.add(g);
    } // End of add_global_gui

    // ** clear ** //
    //
    // Deletes all elements from panel11 of the GUI
    //

    public void clear()
    {
      panel11.removeAll();
    } // End of clear

    // ** clear_all ** //
    //
    // Deletes all elements of the GUI
    //

    public void clear_all()
    {
      panel11.removeAll();
      panel12.removeAll();
      panel21.removeAll();
      panel22.removeAll();
      panel23.removeAll();
      panel1.removeAll();
      panel2.removeAll();
    } // End of clear_all

} // End of class Ramsey_GUI
```

class Ramsey_Item_Listener

```
import java.awt.event.*;

//
// ** Ramsey_Item_Listener ** //
//
// Handles the item events for the domain of num_populations,
// pop_size, num_colors, elitism, selection, crossover,
// migrant_selection, and migration_topology.
//

public class Ramsey_Item_Listener implements ItemListener
{
```

```
    private Ramsey applet;
    private int command;

    // ** Constructor ** //

    public Ramsey_Item_Listener(Ramsey ramsey_applet, int
   listening_command)
    {
      applet = ramsey_applet;
      command = listening_command;
    }



    // ** itemStateChanged ** //
    //
    // Invokes a procedure in Ramsey_GUI according to the selected
   // item.
    //

    public void itemStateChanged(ItemEvent event)
    {
      switch(command)
      {
        case Ramsey.NUMPOP:

   applet.set_numpopulations(Integer.parseInt((String)(event.getItem(
   ))));
          break;

        case Ramsey.POPSIZE:

   applet.set_popsize(Integer.parseInt((String)(event.getItem())));
          break;

        case Ramsey.NUMCOL:

   applet.set_numcolors(Integer.parseInt((String)(event.getItem())));
          break;

        case Ramsey.SELECT:
          applet.set_selection((String)event.getItem());
          break;

        case Ramsey.ELITISM:
          applet.set_elitism(event.getStateChange() ==
   ItemEvent.SELECTED);
          break;

        case Ramsey.CROSSOVER:
          applet.set_crossover((String)event.getItem());
          break;

        case Ramsey.MIGSELECTION:
          applet.set_mig_selection((String)event.getItem());
          break;

        case Ramsey.MIGTOPOLOGY:
          applet.set_mig_topology((String)event.getItem());
          break;
      }

    } // End of itemStateChanged

} // End of class Ramsey_Item_Listener
```

class Table

```
// provides supporting functions to obtain the (i,j) coordinates of a
// particular edge

public class table
{
    private int[] look_upi;  // Table to find the i-th component of a
   given edge
    private int[] look_upj;  // Table to find the j-th component of a
   given edge

    //** Constructor **//
    //
    // Build the look-up table according to the number of vertices in
   // the graph
    //

    public table(int num_vertices)
    {
         int k=0;
         int num_edges=0;

         // Compute the number of edges in the complete graph with
   // num_vertices

         num_edges = (num_vertices * (num_vertices - 1)) / 2;

         look_upi = new int[num_edges];
         look_upj = new int[num_edges];

         for(int i=0; i < num_vertices; i++)
            for(int j=0; j < i; j++)
               {
                  look_upi[k] = i;
                  look_upj[k] = j;
                  k = k+1;
               }
    } // End of Constructor

    //** i **//
    //
    // Takes as input an edge number and returns the corresponding i
   // coordinate
    //

   public int i(int edge) { return look_upi[edge]; }

    //** j **//
    //
    // Takes as input an edge number and returns the corresponding j
   // coordinate
    //

   public int j(int edge) { return look_upj[edge]; }


  //** edge **//
  //
  // Takes as input the (i,j) coordinates and returns the
  // corresponding edge number
  //
```

```
   public int edge(int i, int j)
   {
        if (i>j)
           return i * (i-1)/2 + j;
        else
           return j * (j-1)/2 + i;
   }

} // End of class table
```

class triangle

```
class triangle
{
   private int num_triangles;  // Number of monochromatic triangles
   private int current_color;  // Current color being used to color
  edges

   // ** Constructor ** //

   public triangle()
   {
      num_triangles = 0;
      current_color = 0;
   }

   // ** reset ** //
   //
   // Sets "num_triangles" and "current_color" to their default
  // values
   //

   public void reset()
   {
      num_triangles = 0;
      current_color = 0;
   }

   // ** find_triangle ** //
   //
   // Checks if a triangle is being formed for all possible colorings
  // of an edge and assigns to the edge the color that results in
  // the fewest number of monochromatic triangles being formed.

public void find_triangle(permutation p, table t, decoder d, int i,
 int j)
 {
    num_triangles = 0;
    current_color = 0;

  while(current_color != d.getnumcolors())
  {
       for(int k=0; k < p.getnumvert(); k++)
       {
          if((i != j) && (i != k) && (j != k))
          {
```

```
              p.setcolor(t.edge(i,j), current_color);

              if ((p.getcolor(t.edge(i,j)) ==
   p.getcolor(t.edge(i,k))) && (p.getcolor(t.edge(i,k)) ==
   p.getcolor(t.edge(j,k))))
                       num_triangles++;
            }
        }

        if (num_triangles < d.getfewestsub())
        {
           d.setfewestsub(num_triangles);
           d.setbestcolor(current_color);
        }

        if (num_triangles == 0)
           current_color = d.getnumcolors();
        else
           current_color++;

        num_triangles = 0;
     }
  } // End of find_triangle

} // End of triangle
```

package crossover

```
package crossover;
import permutation;

 // ** PMX (Partially Matched Crossover) ** //
 //
 // A matching section consisting of two crossover points is
 // randomly chosen. Elements of each parent that occur in the
 // matching section of the other parent are replaced. The
 // matching section maintains its original position in the
 // new chromosome(s)
 //
public class pmx
{
   public static void mate(permutation father, permutation mother,
                           permutation child1, permutation child2)
   {
     // Copy *father* into *child1* and *mother* into *child2*

     for(int i=0; i < father.getnumedge(); i++)
      {
        child1.setedge(i,father.getedge(i));
        child2.setedge(i,mother.getedge(i));
      }

     // Select two random crossover points to form the matching
  // section

     int start = (int) (Math.random() * (child1.getnumedge() - 1));
     int end = (int) (Math.random() * (child1.getnumedge() - 1));

     if(start > end)
```

```
        {
            int tmp = start;
            start = end;
            end = tmp;
        }

      // Create an array for *child1* which records the position
      // of every edge of *child1*, similarly for *child2*

      int child1_position[] = new int[child1.getnumedge()];
      int child2_position[] = new int[child2.getnumedge()];

      for(int i=0; i < child1.getnumedge(); i++)
      {
        child1_position[child1.getedge(i)] = i;
        child2_position[child2.getedge(i)] = i;
      }

      int child1_tmp = 0;
      int child2_tmp = 0;

      for(int i=start; i <= end; i++)
      {
        child1_tmp = child1.getedge(i);
        child2_tmp = child2.getedge(child2_position[child1_tmp]);

        // Swap the contents at position i of *child1* with content
        // of *child1* at the position indicated by the content
        // of *child1_position* at position i of *child2*

        child1.setedge(i,
   child1.getedge(child1_position[child2.getedge(i)]));
        child1.setedge(child1_position[child2.getedge(i)],
   child1_tmp);

        // Swap the contents at position i of *child2* with the
   // content of *child2* at the position indicated by the      //
   content of *child2_position* at position i of *child1*

        child2.setedge(child2_position[child1_tmp],
   child2.getedge(i));
        child2.setedge(i, child2_tmp);
      }

    } // End of mate

} // End of pmx
```

package mutation

```
package mutation;

import permutation;

// ** Swap Mutation ** //
//
// Two randomly selected edges in a permutation are swapped
//
```

```
public class swap
{
  public static void mutate(permutation child)
  {

    // Generate two random numbers from 0 to (num_edges-1)

    int edge1 = (int) (Math.random() * (child.getnumedge() - 1));
    int edge2 = (int) (Math.random() * (child.getnumedge() - 1));

    // Swap the edges at positions edge1 and edge2 in the permutation

    int tmp = child.getedge(edge1);
    child.setedge(edge1, child.getedge(edge2));
    child.setedge(edge2, tmp);

  } // End of mutate

} // End of swap
```

package Selection

```
package selection;

import permutation;


 //** Roulette **//
 //
 // This function implements Roulette-Wheel based selection.
 // It takes a population of permutations as input, and
 // returns a single permutation which is selected with a
 // probability proportional to its fitness value relative
 // to the sum average population fitness value.
 //

public class Roulette
{

   public static int select(permutation[] population, int popsize)
   {
     double totalsum;        // Sum of all fitness values in the
  population
     double partialsum;      // Partial sum of fitness values in the
  population
     int stop;               // Random number between 0 and totalsum
     int index;              // Index of selected permutation

     totalsum = 0;
     index = 0;
     partialsum = 0;

     // Calculate totalsum

     for(int i=0; i < popsize; i++)
       totalsum += population[i].getfitval();

     // Generate a random number between 0 and totalsum
```

```
    stop = (int) (Math.random() * totalsum);

    // Select the first permutation whose fitness value makes
  // partialsum greater than or equal to stop

    while((index < popsize) && (partialsum < stop))
      {
        partialsum += population[index].getfitval();
        if (partialsum < stop) index++;
      }

    return index;

  } // End of select

} // End of Roulett
```

package util

```
package util;
import java.util.Vector;
import java.lang.Math;
import java.awt.Graphics;
import java.awt.Color;
import permutation;
import table;

public class graph
{

  // ** set_points ** //
  //
  // This function is used to plot complete graph on num_vertices.
  // This function returns num_vertices equally spaced points
  // around the circumference of a circle centered at center
  // and with radius radius.
  //

  public static int[] set_points(int num_vertices, int radius, int
  center)
   {
     int x,y;

     // Create an array to hold the (x,y) coordinates for all the
  // vertices

     int[] C = new int[2*num_vertices];

     // Calculate the angle between equally spaced points along the
  // circumference

     double angle = 2*Math.PI/num_vertices;

     int count = 0;

     for(int i=0; i < 2*num_vertices; i=i+2)
     {
       x = (int) (radius * Math.cos(count*angle)) + center;
       y = (int) (radius * Math.sin(count*angle)) + center;
       count++;

       C[i] = x;
```

```
        C[i+1] = y;
      }
      return C;

   } // End of set_points


   // ** edge_color ** //
   //
   // This function sets the color of a line to the corresponding
   // color of an edge.
   public static void edge_color(Graphics g, int color)
   {
     switch(color) {
        case 0:
             g.setColor(Color.black);
             break;
        case 1:
             g.setColor(Color.red);
             break;
        case 2:
             g.setColor(Color.blue);
             break;
        case 3:
             g.setColor(Color.yellow);
             break;
        case 4:
             g.setColor(Color.green);
             break;
        case 5:
             g.setColor(Color.pink);
             break;
          }

   } // End of edge_color


   // ** draw_graph ** //
   //
   // This function draws the complete graph corresponding to a
   // particular permutation given as input.
   //

   public static void draw_graph(Graphics g,permutation p,table t,
   int radius, int center)
   {
     int[] points = set_points(p.getnumvert(),radius,center);

     for(int i=0; i < (p.getnumvert() * 2); i=i+2)
       g.drawOval(points[i], points[i+1], 1,1);

     int x1,y1,x2,y2;

     for(int i=0; i < p.getnumedge(); i++)
      {
        x1 = 2 * t.i(p.getedge(i));
        y1 = x1 + 1;

        x2 = 2 * t.j(p.getedge(i));
        y2 = x2 + 1;

        edge_color(g,p.getcolor(p.getedge(i)));
        g.drawLine(points[x1], points[y1], points[x2], points[y2]);
      }
   } // End of draw_graph
} // End of graph
```

VITA

Iker Gondra

Candidate for the Degree of

Master of Science

Thesis: A COARSE-GRAIN PARALLEL GENETIC ALGORITHM TO IMPROVE THE BOUNDS OF SOME RAMSEY NUMBERS

Major Field: Computer Science

Biographical:

Personal Data: Born in Bilbao, Vizcaya, Spain, August 20, 1977, son of Maria Luisa Luja and Jose Enrique Gondra.

Education: Graduated from the Sagrado Corazon High School, Sucre, Bolivia, in December 1994; received Bachelor of Science in Computer Science from Oklahoma State University, Stillwater, Oklahoma, US, in December 1998; completed the requirements for the degree of Master of Science in Computer Science at the Computer Science Department of Oklahoma State University in May 2002.

Experience: Employed by Computing and Information Services, Oklahoma State University, as a Computer Lab Assistant from August 1997 to August 1999; employed by Computer Science Department, Oklahoma State University, as a Graduate Teaching Assistant since August 1999.