Project 3 Report

Brian Satzinger

December 4, 2009

Abstract

An extension to existing parallel genetic algorithms[1] is explored, in which candidate solutions migrate between distinct populations. The mutation rate varies among the populations, and individuals are selected for migration such that individuals with poor fitness migrate towards areas with higher mutation, and individuals with high fitness migrate toward areas with lower mutation. This method allows a greater exploration of the search space, while still preserving good solutions in the areas with low mutation.

The problem used to test this genetic algorithm is the graph coloring problem. This requires the genetic algorithm to develop a vertex coloring scheme for a graph so that no vertices which are connected by an edge share the same color. This problem is NP complete, and therefore brute force searches become intractable for larger graphs. The graph coloring problem is equivalent to certain scheduling problems, in which different processes require exclusive access to the same resources. Each process is represented as a vertex in the graph, and each resource conflict is represented as an edge between the two conflicting processes. When the graph is properly colored, the vertices with the same colors represent the processes which can be run in parallel with no conflicts.

The results suggest some benefit to the proposed techniques, although further exploration is necessary to draw any firm conclusions.

Contents

1 Introduction **2**
- 1.1 Island GA . . . 2
- 1.2 Random Migration . . . 3
- 1.3 Mutation Gradient . . . 3
- 1.4 Directed Migration . . . 4

2 Graph Coloring Problem **5**
- 2.1 Description . . . 5
- 2.2 Applications . . . 6
- 2.3 Brute Force Approach . . . 6
- 2.4 Chromosome Representation . . . 7
- 2.5 Fitness Function . . . 7

3 Implementation **7**
3.1 MPI . . . 7
3.2 Parallel Algorithm . . . 8
3.2.1 Main . . . 8
3.2.2 Root . . . 8
3.2.3 Child . . . 9

4 Results & Discussion **10**
4.1 Time to find first optimal solution . . . 10
4.2 Average Fitness Over Time . . . 12
4.3 Spatial Effects on Average Fitness . . . 14

5 Conclusion **17**

References **18**

1 Introduction

The genetic algorithm discussed in my previous genetic algorithm project consisted of a single population. However, genetic algorithms lend themselves well to parallelization. A group of populations can be developed in parallel, to allow a greater exploration of the search space.

1.1 Island GA

Island genetic algorithms are such parallel genetic algorithms. In an island GA, separate populations are maintained. However, individuals are allowed to migrate between these populations according to certain rules. For example, the populations can be arranged spatially, and individuals can only migrate to nearby populations. This allows genetic information to transfer between populations [1].

For my project, the populations are arranged in a grid. Although the dimensions can be set to any value, all experiments utilize a 4x4 grid, as shown in Figure 1. Each population is assigned a numeric identifier (also known as a rank, in MPI terminology), which determines the population's x and y coordinate. The origin for this coordinate system is in the upper left corner, and all coordinates are positive. Equations 1 and 2 are used to determine the x and y coordinates from the rank.

$$x = (rank - 1) \% WIDTH \tag{1}$$

$$y = \lfloor (rank - 1) / WIDTH \rfloor \tag{2}$$

-

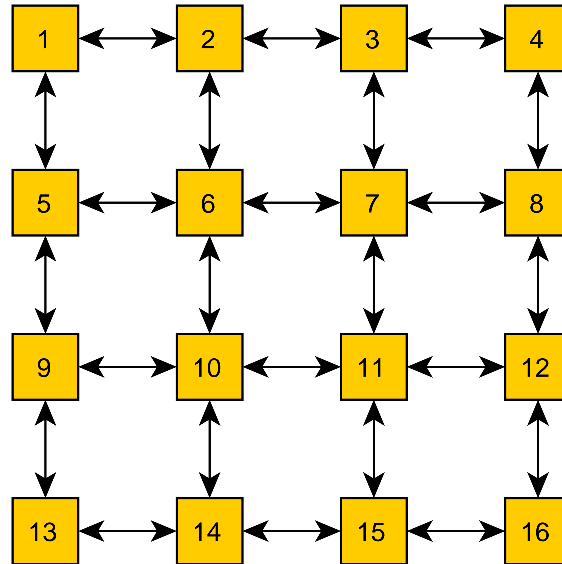

Figure 1: Spatial arrangement of populations, showing migration paths and ranks.

1.2 Random Migration

In random migration, each population exchanges one individual with each of its neighbors during each generation. The individuals selected for migration are selected randomly, without consideration for fitness. This provides an exchange of genetic information between neighbors.

1.3 Mutation Gradient

I was curious to see the effect of various mutation rates on the population. To this effect, I allowed the mutation rate to be a function of the y coordinate of the population. Those populations with a larger y coordinate would have a higher probability of mutation, while those populations with y=0 would have no mutation. Between these two extremes, the mutation rate was scaled linearly, as shown in Figure 2. For the experiments discussed in this report, when a mutation gradient is used, it is the one shown in Figure 2. When a mutation gradient was not used, the mutation rate was set globally to 0.025, which is the average value of the mutation gradient.

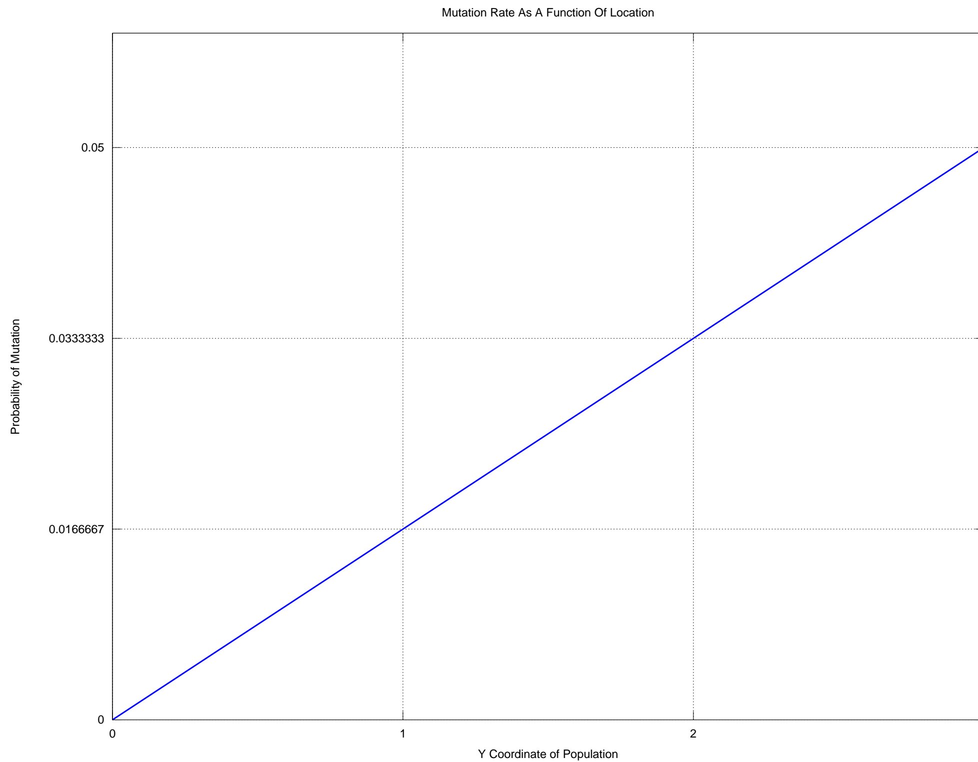

Figure 2: Mutation rate varies according to Y coordinate

1.4 Directed Migration

In directed migration, the mutation gradient plays a role in selecting individuals for migration. When selecting individuals to exchange with a horizontal neighbor (with the same mutation rate), selection is random. When selecting individuals to exchange with a population with a higher mutation rate, proportional selection is used to select for low fitness. When selecting individuals to exchange with a population with a lower mutation rate, individuals are selected for high fitness.

The purpose of directing migration in this way is to set up “convection currents” between the populations, in which high quality genetic material rises to the top where it is safe from mutation, while low quality genetic material sinks to the bottom, where it is subject to greater mutation (and therefore greater exploration of the search space). The horizontal exchange between neighbors promotes greater exploration, and provides more paths for migration.

In the case that directed migration is used without a mutation gradient, individuals are still sent preferentially in different directions based on their fitness, with better individuals being sent upward. However, in this case the mutation rate does not actually vary between populations.

2 Graph Coloring Problem

To evaluate these extensions of Island GAs, I chose the graph coloring problem as the problem to solve. It was chosen because it is relatively simple to devise a chromosome representation for the problem, it has useful applications in scheduling, and it is NP complete, so it is difficult to find a solution by ordinary search algorithms.

2.1 Description

The graph coloring problem is to assign each vertex in a graph a color such that no neighboring vertices have the same color. The graph shown in Figure 3 can be colored using only three distinct colors, as shown in Figure 4. Many other such colorings are possible which satisfy the problem.

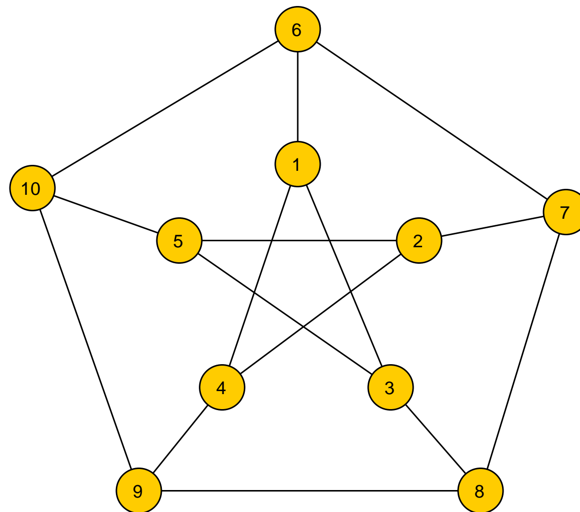

Figure 3: An uncolored graph

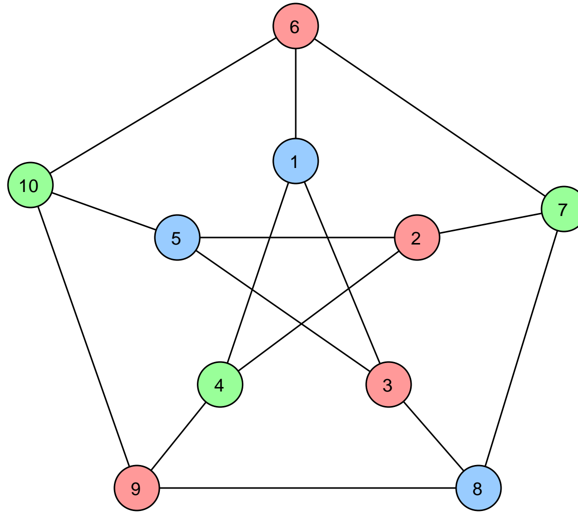

Figure 4: One optimal graph coloring produced by the genetic algorithm

2.2 Applications

The graph coloring problem can be considered an expression of a scheduling problem. Given a set of processes which each require exclusive access to some resources, how can the processes be scheduled in an optimal way? If each process is represented as a vertex in a graph, and each resource conflict is an edge between the two conflicting processes, then the optimal coloring of the graphs gives an optimal schedule. All processes with the same color can be run simultaneously without conflict. The number of distinct colors used is equivalent to how serialized the schedule is.

2.3 Brute Force Approach

To provide a comparison to the performance of the genetic algorithm, I wrote a program that performs a brute force search for graph coloring schemes for the graph in Figure 3. This program performs a naive search, in that it tries all possible combinations of 10 colors on all 10 vertices. This gives rise to 10^{10} possible solutions, which is a very large (but not infeasible) number of combinations to try. Obviously many of the optimal solutions found in this search will be equivalent to other solutions through rotation, reflection, and color renaming transformations.

Of the 10,000,000,000 candidate solutions, only 14,400 were found to be optimal. This search of the entire solution space was very computationally intensive, requiring 130 minutes to complete on my 2.6 GHz single core computer. It required 76 seconds of searching to find the first optimal solution. For larger graphs, the search space grows exponentially, making a brute force approach impractical for many real problems. It is hoped that the genetic algorithm will be good at providing approximate solutions in much less time.

2.4 Chromosome Representation

A candidate solution is represented as an array of integers, with one element of the array for each vertex in the graph. The integers are restricted in value to between 0 and (N-1) where N is the number of vertices. This allows for the trivial solution of giving each vertex a unique color (which is the optimal solution in the case of a fully connected graph). Mutation is performed by randomly adding or subtracting 1 from the chosen gene. In the event of an overflow, the integer value wraps around so that it remains in the valid range.

2.5 Fitness Function

The fitness function depends on the layout of the graph. This information is given to the program in an adjacency table, which identifies the connections between all vertices in the graph. The fitness function uses the chromatic information in the chromosome in combination with the adjacency table to count the number of connected vertices which have the same color (n_{same}) and the number with a different color ($n_{different}$). It also counts the number of unique colors used in the chromosome (n_{colors}). The fitness is given by equation 3. This rewards configurations that give different colors to adjacent vertices, while punishing configurations that give the same color to adjacent vertices, or that use an excessive number of distinct colors.

$$fitness = \frac{n_{different} - 2 \cdot n_{same}}{n_{colors}} \tag{3}$$

3 Implementation

The island genetic algorithm with directed migration and a mutation gradient is implemented in the C programming language, and is designed to be run on a Linux cluster using the message passing interface (MPI). This takes advantage of the parallelism inherent to the algorithm, and allows it to be run on a high performance supercomputing cluster, as well as on a single computer with MPI installed (effectively a cluster of one machine).

The program can be recompiled with options for directed migration, random migration, and a mutation gradient. These versions of the program are used to compare the effectiveness of these techniques.

3.1 MPI

MPI, or message passing interface, is a technology designed for exchanging variables between processes running on different computers. It is commonly used to communicate between different instances of the same program. MPI provides each process with a unique identifier, the rank, which allows the program to take on different roles, as well as identify its "neighbors". In my program, the process with rank 0 (known as the root) takes on a special role as a coordinator, while the remaining processes (known as the children[1]) run the parallel genetic algorithm.

[1] Not to be confused with children in the sense of offspring

Multiple processes can share the same physical computer, in which case MPI uses shared memory to provide inter-process communication. When processes on different computers communicate, the communication is over the network. MPI abstracts these details, and provides functions for sending and receiving data to other processes identified only by their rank. This allows the same program to run on a single computer (using shared memory), or on a cluster of computers (using network communication).

3.2 Parallel Algorithm

A single executable can take on the role of either root or child, depending on the rank it is assigned. The root collects aggregate data from the children, and is responsible for telling the children to terminate. The children send data to the root, receive commands from the root, and exchange individuals between them.

3.2.1 Main

The main program starts MPI, and determines the processes own rank and the number of other processes running. It them starts the root subroutine or the child subroutine, depending on its rank.

Algorithm 1 The main program

- Initialize MPI
- Determine my own rank
- Determine how many other processes are running
- If my rank is 0
  - Start the root subroutine (passing the number of processes running)
- Else
  - Start the child subroutine (passing my rank and the number of processes running)
- End MPI
- Exit

3.2.2 Root

The root subroutine synchronizes the child subroutines, and collects data about their populations. It is responsible for defining the convergence criteria (since it can see data about the entire distributed algorithm), and instructs the children to continue or to stop after each generation.

Algorithm 2 The root subroutine

- Do Until Convergence
  - Receive status updates from all children
  - Print the status updates
  - Send all children a command to continue
- Send all children a command to stop running
- Return

3.2.3 Child

The child subroutine is the most complex, and implements the genetic algorithm.

The exchange of individuals with neighbors presented a few implementation problems. The roulette wheel used for proportional selection cannot be modified to include new individuals without completely recalculating it (since the new individuals might change the maximum fitness used to scale the fitness scores). However, the roulette wheel must be calculated before performing migration, since proportional selection is used to pick individuals. These individuals cannot subsequently be used for reproduction without recalculating the roulette wheel. A solution is to store the received individuals in a buffer until the start of the next generation, when they are added to the population at random locations.

Algorithm 3 The child subroutine

- Determine the population's X and Y location from the rank
- Calculate the population's mutation rate from its location
- Create an initial population
- Determine the ranks of the neighbors
- Do until the root sends a stop command
  - Add travelers received from other populations during the last generation
  - Fitness:
    - Calculate the fitness of each individual
    - Construct a roulette wheel for selecting for high fitness
    - Construct a roulette wheel for selecting for low fitness
  - Send a status message to the root
  - Migration:
    - Select individuals to send to neighbors (using random and proportional selection to pick random, high fitness, and low fitness individuals, depending on the direction of the neighbor)
    - Send those individuals to neighbors
    - Receive individuals from neighbors (these do not participate in selection or crossover until the next generation)
  - Reproduction:
    - Select the parents using proportional selection
    - Perform one point crossover with the parents to generate children
    - Combine the parents and children to form the new population
  - Mutation
  - Receive a stop/continue message from the root
- End

4 Results & Discussion

4.1 Time to find first optimal solution

These results are a comparison of the effect of migration (both random and directed) and mutation gradients on the effectiveness of the genetic algorithm. The statistic of most interest is the number of generations required to produce the first optimal offspring. Various combinations of random

migration, directed migration, and mutation gradients were evaluated for this criteria. For each configuration, four independent trials were conducted, and the results combined to give average (Figure 5) and minimum (Figure 6) times (in generations).

These graphs include the item “None,” which includes no mutation gradients, and no migration of any kind. This is equivalent to a genetic algorithm with multiple independent populations that do not interact in any way.

The Island GA described in [1] is closest to the configuration with only random migration (although the Engelbrecht text describes some algorithms which select travelers based on fitness, the direction of travel does not depend on fitness).

For these trials, the number of individuals in each of the 16 populations was 512. For those trials using a mutation gradient, the mutation rate is given by Figure 2. For those trials not using a mutation gradient, the mutation rate is the average rate given by Figure 2, which is 0.025.

These graphs suggest that directed migration provides better results than random migration, which is in turn better than no migration. However, it is not clear if the mutation gradient provides any benefit. Although the average times (Figure 5) do not show improvement from the mutation gradient, the minimum times (Figure 6) do show better improvement with a mutation gradient in all cases. The shortest time with a mutation gradient (2 generations) is half of the shortest time without a mutation gradient (4 generations). More study is required to draw a firm conclusion about the mutation gradient.

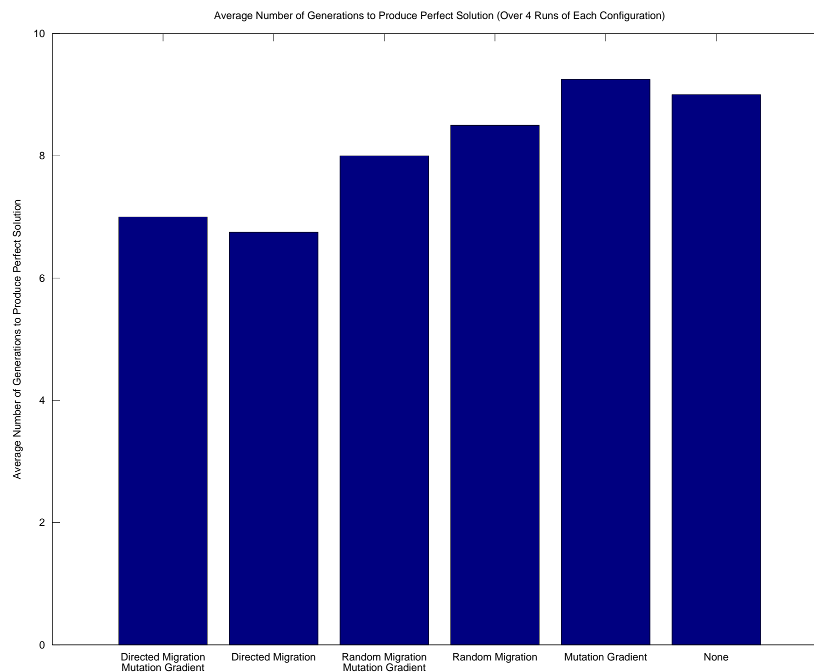

Figure 5: Average time to produce the first optimal offspring

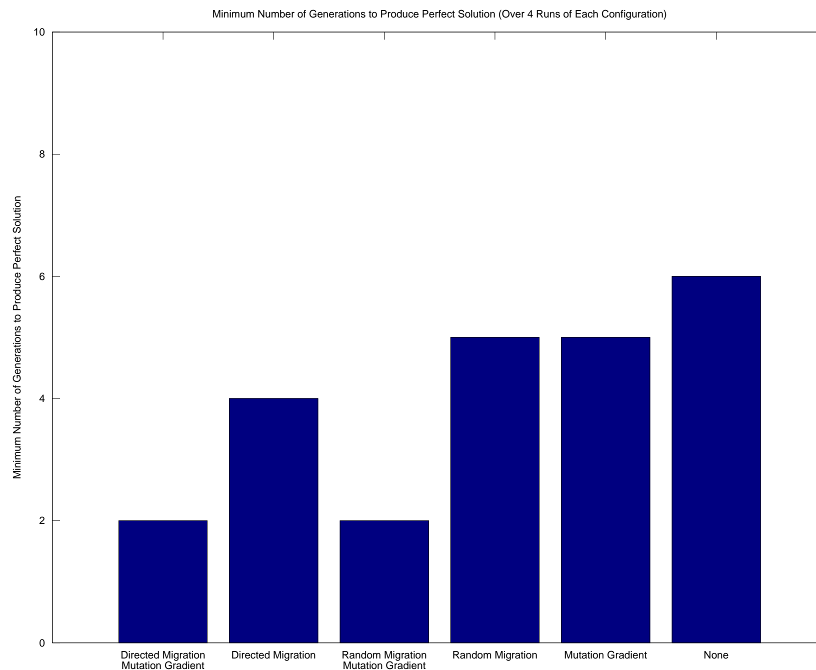

Figure 6: Minimum time to produce the first optimal offspring

4.2 Average Fitness Over Time

The average fitness of each population is an interesting statistic, because it can reveal subtler effects of the different migration and mutation strategies. My experiments collected a large amount[2] of data from different configurations, and it is not practical at this time to do a comprehensive analysis of all of it (The graphs in section 4.1 are aggregated from the entire data set).

However, I did notice an interesting point of comparison between two experimental runs. One of these runs took place with directed migration and a mutation gradient, and the other run took place with neither. Both runs happened to find the optimal solution after the 11th generation. Since they reached this milestone of maximum fitness simultaneously, it is interesting to compare the average fitness over time in both cases (Figure 7).

[2] Approximately 14MB of text

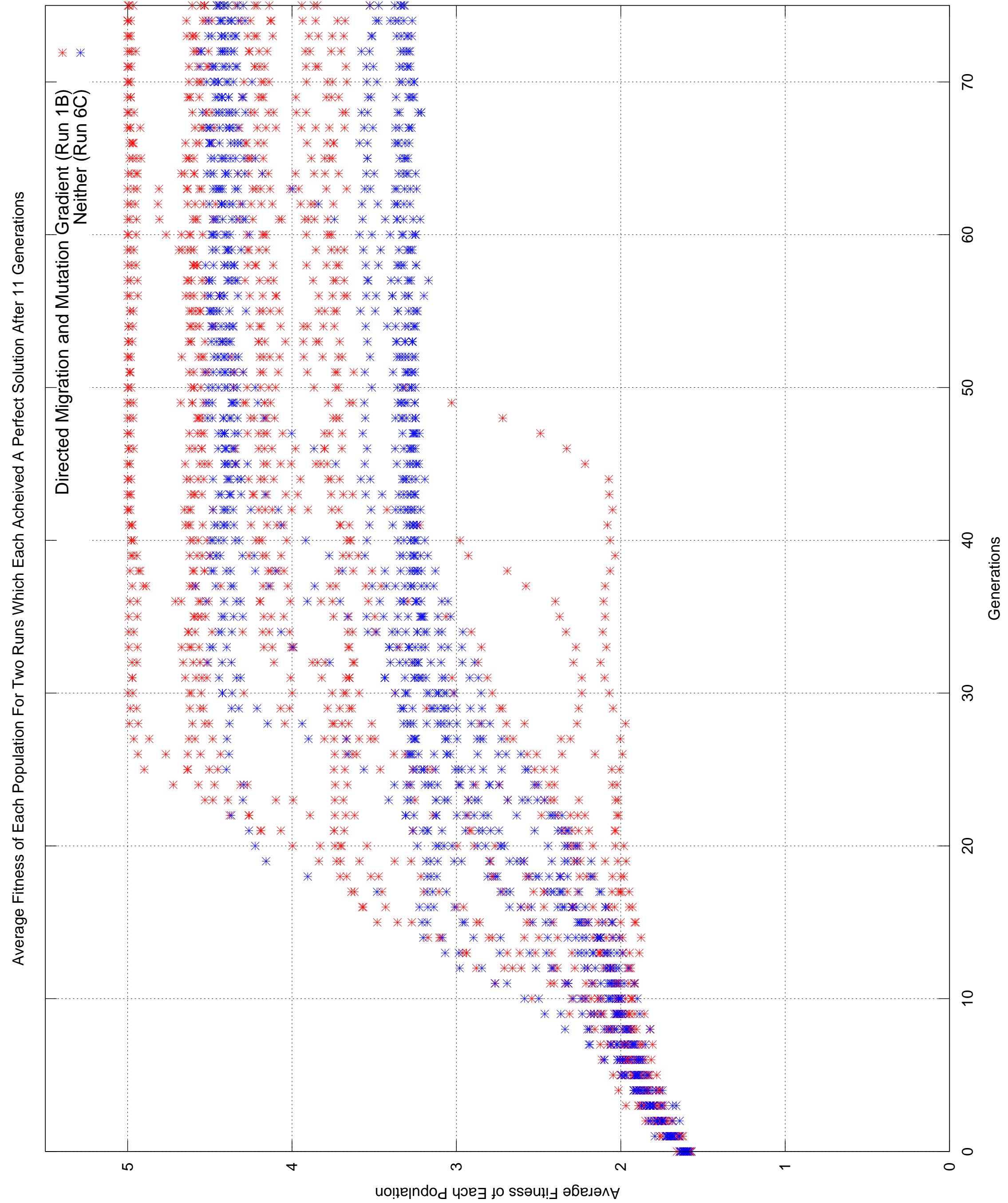

Figure 7: Average fitness over time for 16 populations in two different configurations

Figure 7 shows that the average fitness can remain relatively low at the time when the first optimal solution is found (generation 11). In this graph, the average fitness from each of the 16 populations are shown independently. The averages from the populations were not combined because different populations tended to follow different trajectories.

In this case, the highest possible fitness is 5. The first population to approach this on average took about 25 generations to do so, and was in the run with directed migration and a mutation gradient (the red points). The run with neither directed migration nor a mutation gradient (the blue points) did not have any populations reach an average of 5 ,although this run did first produce a single optimal solution at the same time as the other one. The average fitness over time was also somewhat lower for all of its populations as well.

The run with directed migration and a mutation gradient maintained a population with very low average fitness (around 2) for about 45 generations before the average fitness in this population suddenly rose, possibly due to a lucky mutation, or because a more fit individual transferred from another population and was heavily selected for. This population with low fitness and high mutation is potentially a source of high exploration, and may have contributed to the rapid rise of the other populations. This is the main advantage of this technique. It allows for prolonged exploration without risking damage to already strong individuals, and allows multiple strategies to develop in parallel.

4.3 Spatial Effects on Average Fitness

The populations maintained by this genetic algorithm are affected by their x and y coordinates. Those populations which are higher in the grid tend to have higher fitness, while those populations on the edge of the grid (which have fewer neighbors) take longer to achieve high fitness because they receive fewer individuals from other populations. Figures 8 and 9 explore these effects. They show data from a single run, with directed migration and a mutation gradient. The points are colored by y coordinate in figure 8, and by x coordinate in figure 9.

Figure 8 shows that there is a strong correlation between the y coordinate of a population and its average fitness, with populations lower in the grid having a lower average fitness. This is expected, based on the effect of directed migration (which moves good individuals upward), and the mutation gradient (which applies mutation more frequently to populations lower in the grid). The average fitness scores seem to separate into four distinct bands, corresponding to each of the four levels in the grid.

Figure 9 presents the same data set, but with the points colored by x coordinate instead of y coordinate. This shows that there is not a strong relationship between average fitness and the horizontal position of the population. This is also expected, since populations which are horizontally adjacent experience equivalent pressure from directed migration and the mutation gradient.

However, there appears to be one effect which is related to horizontal position. The first four populations to reach an average fitness of 5 are all in the upper row of the grid. The first two populations to reach this average fitness (shown as green and blue in Figure 9) are the inner two populations in this row, and have 3 neighbors. The last two populations in this row to reach an average fitness of 5 (red and magenta) are the outer populations in the row, and only have two neighbors. This suggests that exchanging individuals with more neighbors enables the population to improve more quickly.

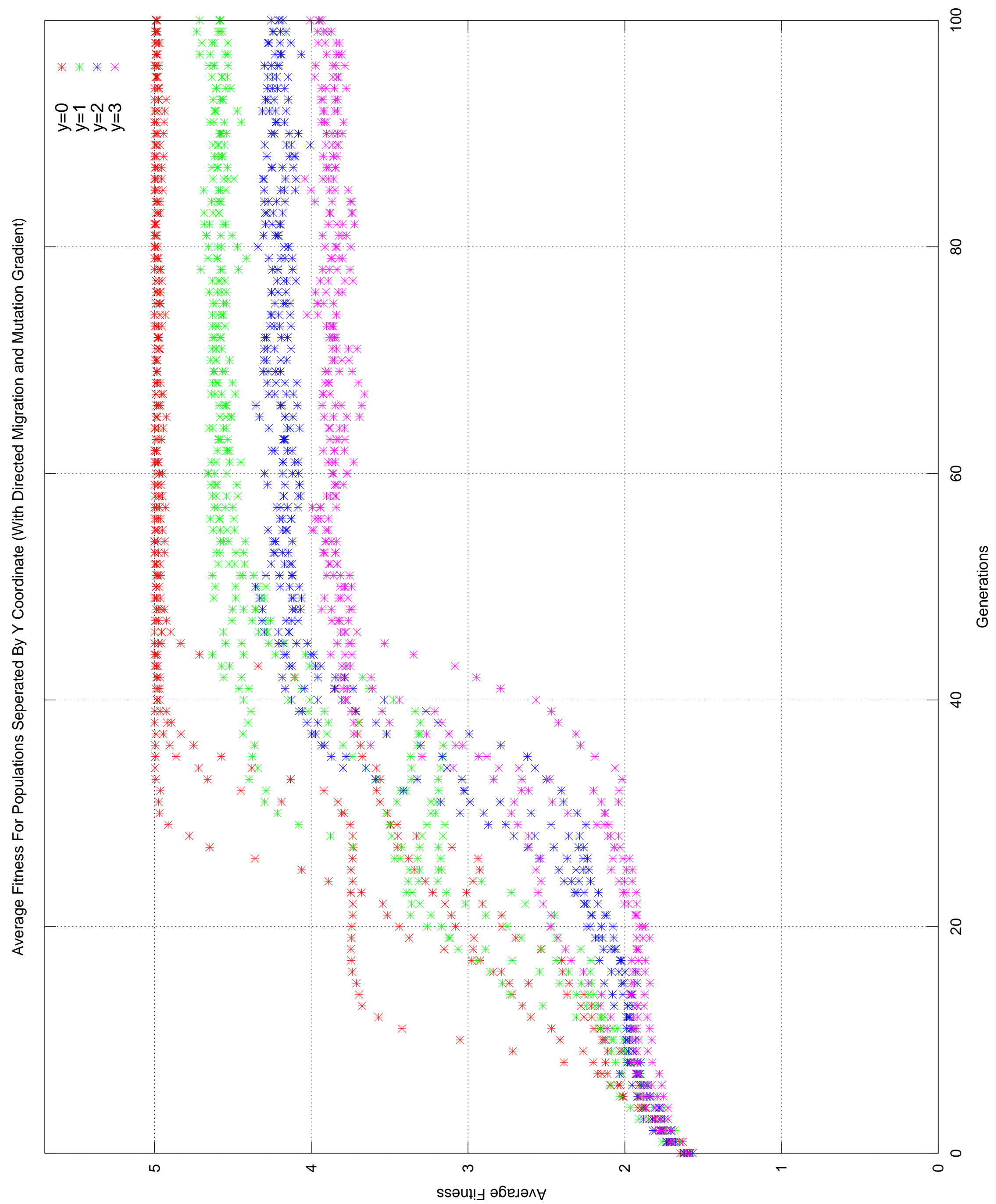

Figure 8: Average fitness colored by y coordinate

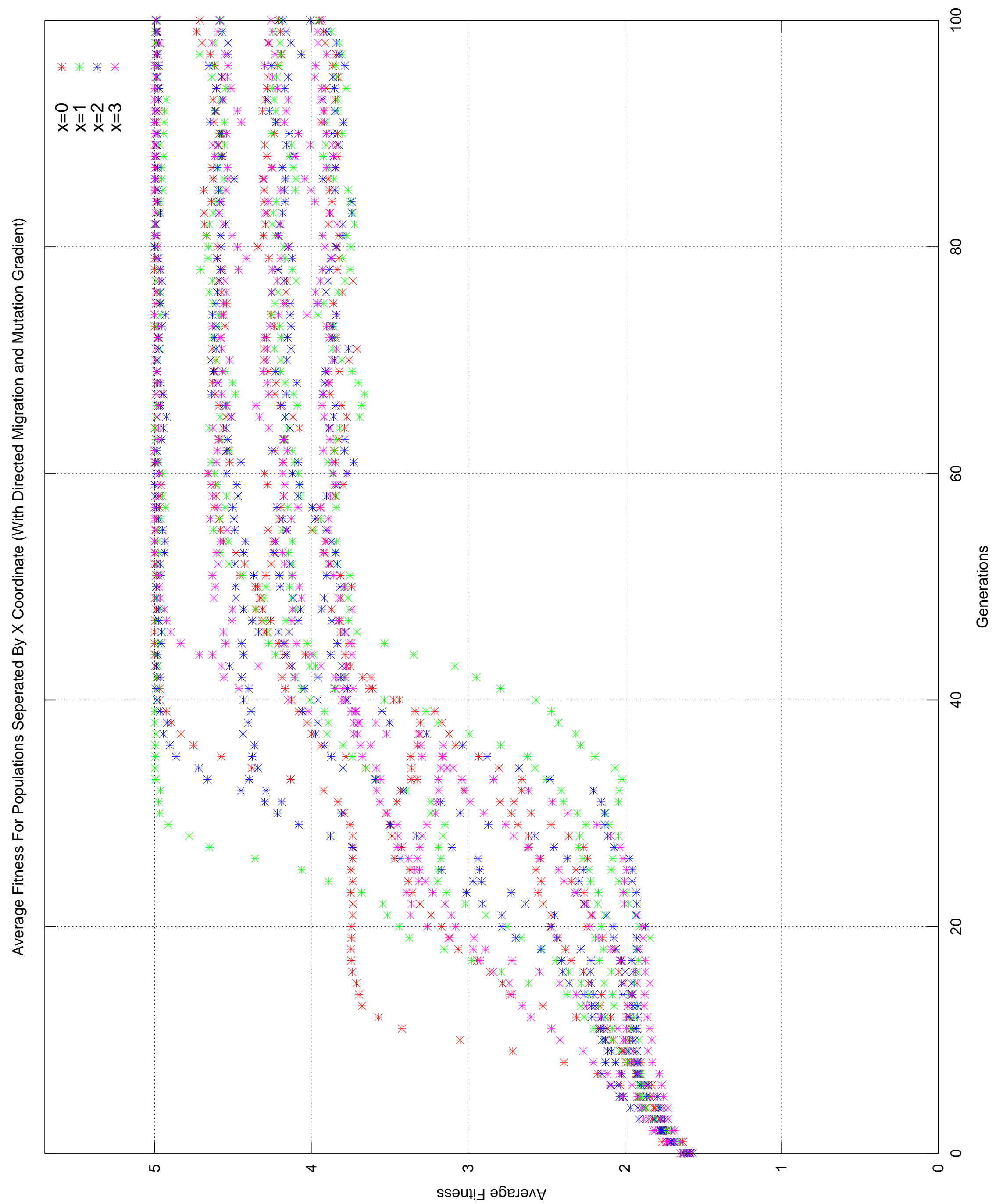

Figure 9: Average fitness colored by x coordinate

5 Conclusion

The results presented above show that directed migration and mutation gradients show promise for improving performance in island genetic algorithms. However, it is not clear at this time exactly how significant the advantages are, since the data set used to draw the conclusions is relatively small.

The data could be improved by performing more experimental runs to give better averages of the time to produce the first optimal offspring. It would also be interesting to use a graph with more than ten vertices, since this would likely require more generations to solve and might more clearly show any performance differences between the techniques.

Another avenue of comparison would be to compare the real execution time of the various techniques. Since migration involves inter-process communication, the additional overhead incurred might increase the execution time, even though the number of generations elapsed might be smaller. However, it would be difficult to compare these fairly, since execution time for MPI programs is heavily dependent on the computers and network used.

This project also differed from my previous genetic algorithm project in that it was implemented in a purely procedural language (C), while the previous project made heavy use of Java's object orientation. This gives a different flavor to the code, and made it more difficult to do some things which were very easy in Java. An improvement to the program would be to rewrite it using objects in C++, since this provides more modularity, and might make it easier to adapt the program to different problem domains.

References

[1] A. P. Engelbrecht, *Computational Intelligence*. Wiley, 2007.