Estimating Optimal TSP Routes Using A Genetic Algorithm

Benjamin Grandy
Student at Brock University
St. Catharines, Ontario, Canada
bg16pz@brocku.ca

I. Introduction

This document is an introduction to solving an NP-hard problem using a genetic algorithm. A genetic algorithm attempts to do this by replicating nature's environmental evolution. Genetic algorithms are often used when trying to solve a problem with a clear way to evaluate the fitness of a solution. Often when using genetic algorithms the goal is to find a very good solution not one that is optimal. Along with a clear way to evaluate a potential solutions fitness, genetic algorithms are often implemented when the amount of possible permutations is asymptotically large.

Some concepts that genetic algorithms try to replicate are genes, chromosomes, selection, crossover, and mutation. Genes in this example are represented by individual cities. Each gene also has an associated vector of where this city is located. Chromosomes are represented by a collection of cities. In this example, chromosomes have multiple restrictions. The chromosome must include every city exactly once, it must not have any duplicate cities or any empty positions. Selection is the process of selecting parents to breed. In this example a parent is represented by a chromosome. A percentage of these children go through the process of crossover. Crossover tries to replicate sexual reproduction between two parents. After the two parents are chosen the crossover method takes a portion of the first parent and second parent to create children chromosomes. This is crucial in attempting to produce a desired child permutation. This report will discuss two different implementations of crossover. One crossover method that is implemented is Uniform order crossover with bitmask. This crossover method creates two children from two parents by compare the parents permutation to a binary bitmask. Another crossover method that is implemented is 1-point crossover which selects a random integer between 0 and the amount of cities to create two child permutations. For this problem the mutation function that was implemented was the scramble mutation. This method takes a random subset of the cities in a child chromosome and scrambles the order of them. For more information and pseudo code on how crossover and mutation methods work please refer to sections II-B–II-D below.

The Travelling Salesman Problem is where given a set of potential cities, travel to each city exactly once and return back to the starting city. Each city is given a vector of it's location and the goal of this problem is to find the euclidean distance and optimize it. The euclidean distance between two cities is calculated by:

$$d = \sqrt{(city_2x - city_1x)^2 + (city_2y - city_1y)^2} \quad (1)$$

Where d is the euclidean distance between the two cities, $city_1$x and $city_1$y is the first city's x and y value respectfully, and $city_2$x and $city_2$y is the second city's x and y value respectfully. By taking a summation of the euclidean distance one can attain a chromosome fitness. By continuously optimizing this fitness it is possible to find a good solution to this problem.

II. Background

This section will be discuss the different types of algorithms that was implemented to help find a good solution for the TSP. This section aims to show how each algorithm works and why they're necessary in order to find a good solution.

A. Genetic Algorithm

A general genetic algorithm works by attempting to replicate Darwin's theory of evolution. To start a genetic algorithm, it needs to first initialize the generation number. This will set a max amount of times the code will run. Then it needs to initialize the starting population. the starting population will be a collection of random permutations that are feasible solutions. The genetic algorithm will then calculate the fitness of each permutation and give the collection of fitness's to the elitism function. The elitism function's goal is the select the best chromosome's fitness per generation and add it to the next generations population. This ensures at each generation, the best chromosome in the next generation won't be worse than the last generation. The algorithm then enters a loop that starts at the generation number that was previous initialized and ends at the max generation limit. Once it enters the loop the first thing that's done is to increment the generation number by 1 indicating the generation that is being worked on. The algorithm then enters the tournament selection function. This function receives a k value between 2 and 5. Tournament selection will select k parents from the previous generation. It will then take the best 2 of the k parents and pass the two parent chromosomes to a crossover function. The algorithm will allow a certain percentage of parent pairs of chromosomes to enter the crossover method. If the algorithm determines this pair of parents will continue into the crossover methods, the algorithm then has two paths it can take. The path the algorithm takes

is based on run-time parameters. After the parents have gone through the crossover method and 2 child chromosomes have been created, the algorithm then allows a certain percentage of those child chromosomes to be passed into the mutation function. The mutation function will then select a random subset within the child chromosome and randomizes the order. If the crossover and mutation doesn't occur the parents are passed onto the next generations untouched.

Algorithm 1: Genetic Algorithm

```
Read run-time GA Parameters Generate random
  permutations of cities to form chromosomes
for Generation = 1 to Max Generations do
    Evaluate Fitness of each chromosome
    Call elitism function to add best chromosomes to
      next Population
    Select next two parents using
      TournamentSelection
    Apply crossover and mutation functions
```

B. Tournament Selection

Tournament Selection works by receiving an array of chromosomes and choosing the best percentage of chromosomes to return. This function is essential in finding best parents to breed in order to hopefully create better children chromosomes. This function works by receiving a k value that states how many chromosomes should be selected each time two parents need to be selected. It then computes the fitness value of each of the chromosome and selects, and returns, the best two.

Algorithm 2: Tournament Selection

```
input: integer k, array population Require:1 < k < 6
for k parents do
    if k_i fitness < BestFitness then
        Parent_1[i] = k_i[i]
    else if k_i fitness < 2^nd BestFitness then
        Parent_2[i] = k_i[i]
return Parents
```

C. Crossover Implementations

The goal of the crossover function is to receive two parent chromosomes from $tournamentselection$ II-B and combine them in order to create two different child chromosomes. The algorithm tries to make chromosomes better at each generation and does so by filtering the best parents from the population. By starting with two of the best k parents from the population it hopes to start with a good foundation of chromosome and modifying it to represent a mix of $parent_1$ and $parent_2$. There are lots of different crossover functions that can be implemented but this report will be focusing on two that was implemented in this genetic algorithm.

1) Uniform Order Crossover with bitmask: Uniform Order Crossover with Bitmask is a type of crossover method that creates a binary array the same size as there are cities and compares it to the two parent chromosomes. The binary array is created by taking a random number between 0 and 1 and repeating that for the amount of cities in the TSP. Once the bitmask is created the parent chromosomes are compared to it. If the bitmask at a certain index is a 1, $child_1$ will take the city at that index of $parent_2$ and $child_2$ will take the city at that index of $parent_1$. If the bitmask is a 0, the algorithm will first attempt for the child to carry over the same city as it's respective parent. This will fail if the city at the parent has already been added to child. If this fails, the algorithm will check each city starting at the first city in the parent and continuing onto the next city until it finds one that hasn't been added yet.

Algorithm 3: Uniform Order Crossover with bitmask

```
input: Parent_1 array,Parent_2 array /*from Tournament
  Selection*/
for each element in bitmask denoted by i do
    if bitmask[i] == 1 then
        child_1[i] = parent_2[i]
        child_2[i] = parent_1[i]
    else
        child_1[i] = parent_1[i]
        child_2[i] = parent_2[i]
return Children
```

2) 1-Point Crossover: 1-point crossover is a lot more simple compared to $UniformOrderCrossoverwithBitmask$ II-C1. I have chosen to implement this method in order to compare a complex crossover method to a simple one. 1-point crossover works by first copying $parent_1$'s chromosome to $child_1$ and $parent_2$'s chromosome to $child_2$. Once both children are identical, the algorithm generates a random number between 0 and the number of cities per chromosome. The algorithm then takes a subset of cities starting from the city at the random value to the last city. This subset is then swapped between children. For example, $child_1$ will be identical to $parent_1$ up to the city before the random number. The order of the cities at and after the random number will be identical to $parent_2$'s cities at and after the same value.

Algorithm 4: 1-Point Crossover

```
input: Parent_1 array,Parent_2 array /*from Tournament
  Selection*/ i is a random integer between 0 and
  amount of cities for 0 to i do
    child_1[i] = parent_1[i]
    child_2[i] = parent_2[i]
for i to amount of cities do
    child_1[i] = parent_2[i]
    child_2[i] = parent_1[i]
return Children
```

D. Scramble Mutation Implementation

Scramble Mutation works by receiving in a child chromosome and then generating two random numbers between 0 and the number of cities in a chromosome. The two random number indicate indexes within that chromosome. The algorithm then takes a subset of the child chromosome starting at the lowest random number and ending at the other random number. The algorithm takes this subset of the child chromosome and scrambles the order of cities in that subset. After the order has been scrambled the subset is then placed back into the child chromosome in the same position it was taken from. Because each child chromosomes come from parents that were already filtered through a $CrossoverImplementation$ II-C, the algorithm rarely mutates a child chromosome. Even so, the mutation function can provide crucial support when the genetic algorithm hits a local minimum. This is because at each crossover function the children produced are heavily influenced by the parent. By changing this child's chromosome randomly, The algorithm will be able to find chromosomes that the crossover function might not find. Even though this potential to find new chromosomes is enticing, this now random chromosome has now undone what tournament selection and crossover method have done. The reliability that the crossover method provides is far more valuable than the random selection of cities mutation has given us. With a low mutation rate the algorithm allows a small amount of chromosome to search a potentially undiscovered path and hopefully a good solution. For more information on mutation rate and their affect on the outcome please refer to sections IV – V

Algorithm 5: Scramble Mutation

```
input: Child /*from a Crossover Method*/
i is a random integer between 0 and amount of cities
 for i to amount of cities do
 └ shuffle order of cities in subset
set the child array to include the shuffled subset
 return Child
```

E. Elitism

Elitism is the algorithms fail-safe. If the crossover method and mutation fail to achieve a better chromosome than the last generation, the best chromosome(s) will always be passed on to the next generation. The elitism function works by receiving the population array of chromosomes, calculating the fitness for each and returning the best chromosome(s). Based on run-time parameters the elitism function could return 1 chromosome or multiple chromosomes. If the multiple chromosomes return is selected the algorithm is given a percentage of chromosomes to return. This mean that the next generation will always have the same amount of chromosomes existing from the last generation. The algorithms goal is to ensure it never backtracks and has the highest chance of creating the best children chromosomes. By focusing on paths it already knows are the best in the previous population.

Algorithm 6: Elitism

```
input: Population array Population is an array full of
  permutations of chromosomes
for Chromosomes in Population array denoted by i do
    if Population[i] Fitness < Best Fitness then
     └ Best Fitness = Population[i] Fitness
return Child
```

F. Repair Chromosomes

The repair chromosomes function works by receiving a child chromosome that has gone through the crossover and mutation function and a parent chromosome. The function starts by checking for any empty slots in the chromosome. The algorithm can identify an empty slot by searching the chromosome for a -1 value instead of a city. If an empty index is found the algorithm will search each city in the parent chromosome for one that doesn't exist in the chromosome yet and add it. The function then checks for any duplicate cities. Since duplicate cities are a result of single point crossover the algorithm will start at the last city and work it's way to the beginning. Once a integer count has detected a city has been included more than once, it will replace it with a city from the parent chromosome that doesn't exist in the child. This function is essential in holding the integrity of the chromosomes and ensuring the algorithm does not pass on a broken chromosome. If a broken chromosome was passed on it would create a ripple effect. Since each child is dependant on it's parent chromosome, if one of the parents chromosomes isn't a feasible solution every child created from that parent would also not be a feasible solution.

III. Experimental Setup

The data collected for this reported came from a general algorithm programmed in java. In order to understand the result the reader must understand how the program was ran. Each run of the genetic algorithm produced one file into the folder 'AssignmentTwoOutputs'. If the folder does not exist, the folder will be created in order to save the outputs. If the folder does exist and there are files inside from previous runs, the files will be overridden each run. Once the genetic algorithm has been run there will be two options for the user to choose. The first option is to run the report version of the software. The second option allows for a single run of the algorithm on a separate file chosen by the user. The user will be able to explore their hard drive and select a .txt file to run on. If the second option is chosen, the file will have to follow a certain format to allow the genetic algorithm to run correctly. In the first line of the text file will need the following integers in this order: POPSize, generations, k, CR, TCR, MR, elitism, Size. POPSize refers to the size of the population. generations refers to the max amount of generations that should be tested. k refers to the amount of parent that should be selected from the population for tournament selection. CR and TCR refer to the crossover rate and the type of crossover. CR will be an

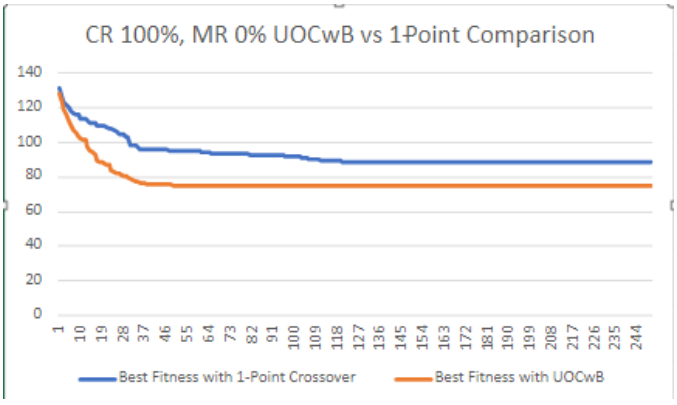
integer between 0-100 representing what percentage of parent chromosomes should undergo crossover implementation. The type of crossover will indicate if Uniform Order Crossover with Bitmask should be used or 1-point crossover. TCR can receive a value of 1 or 2, 1 being UOCwB and 2 being 1-point crossover. MR refers to what percentage of children should be mutated and will take a percentage from 0 to 100. Elitism has two different functionalities. If a 1 is inputted then the best chromosome will be carried over between each generation. Otherwise, a value from 01 to 100 can be received which indicates what percentage of the population will be carried over in between generations. Size refers to the amount of cities in each chromosome. Have parameters set inside the text file's data set is very handy for running multiple types of tests like in this report. This allows the user to run multiple files and not have to compile the code before each run. For this report a total of 300 files were outputted total time taking 1.315 minutes or 78.9 seconds. A total of 60 different parameters were used and each parameter was ran 5 times each run using a different seed. There were constant parameters that never changed each run. POPSize was always 250, generations was always 250, and the k value for tournament selection was always 4. The main sets of variable parameters were as followed: Crossover Rate was 100 percent and mutation rate was 0 percent, Crossover Rate was 100 percent and mutation rate was 10 percent, Crossover Rate was 90 percent and mutation rate was 0 percent, Crossover Rate was 90 percent and mutation rate was 10 percent. the last parameter set was determined empirically. The goal was to find the most optimal crossover rate and mutation rate. The program was ran with multiple different crossover rate and mutation rate and was shown that a low percentage of mutation rate and a high crossover rate achieved the best results. After many runs the best results were shown from a 85 percent crossover rate and a 02 percent mutation rate. More on the effects of changing parameters in sections IV – V.

IV. Results

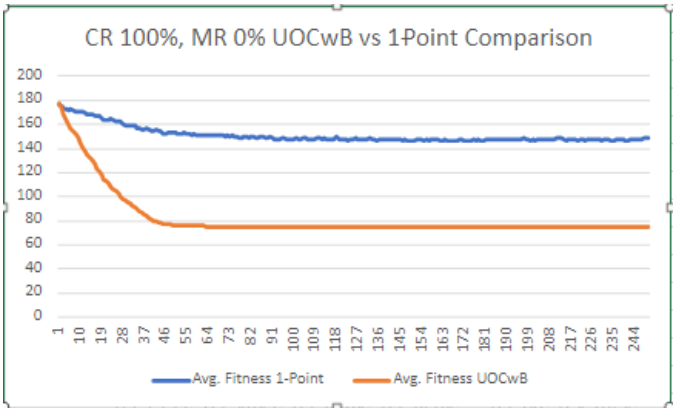
Changing run-time parameters have a large effect on the outcome of the results. In this section different parameters will be compared based on their best fitness and average fitness. Since each parameter was ran 5 times, the comparison will be an average of best fitness and an average of the average fitness. This helps reduce the effect of an anomaly and will hopefully provide more a more accurate picture for how the algorithm truly runs. Another comparison this report will test is the effect that elitism has on the best fitness and average fitness. The parameter that will be tested is the amount of chromosomes passed onto the next function. The comparison will be 1 chromosome, 5 percent of the best chromosomes, and 10 percent of the best chromosomes all with 5 runs each. Like the uniform order crossover with bitmask, the comparison will be made on the average of the 5 best and average fitness.

A. Crossover Rate of 100 Percent and Mutation Rate of 0 Percent

With a 100 percent crossover rate and a 0 percent mutation rate the graph shows that once it has found a best fitness that has been around for a couple generations it has a difficult time finding a chromosome with better fitness. This is because of the zero percent mutation rate. With zero percent mutation rate the algorithm doesn't get diversity with only the crossover method. The graph also shows that the uniform order crossover implementation with bitmask does better at finding a chromosome. This is likely because of the better diversity that UOCwB offers. The graph also shows that when it does find a better chromosome, the chromosome is similar to the previous best chromosome in terms of fitness. This is also due to a lack of diversity from no mutation function. Another thing the graph shows is that when the generations is low, Uniform order crossover with bitmask does a better job at finding significantly better chromosomes than 1-point crossover. Notice the 1-point crossover slowly gets better without making major jumps.

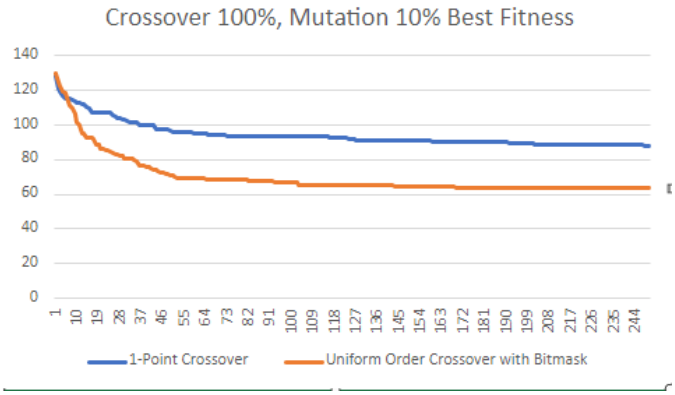


For average fitness, the graph show a very big discrepancy. Uniform order crossover with bitmask does better in every single way. The graph shows it is faster at finding a good solution as well as better at finding consistently better solutions at each generation during early generations. The shape of the average fitness for 1-point crossover shows what was stated in the analysis of best fitness for the same parameters. 1-point crossover has a difficult time finding better solutions. This is accentuated by the lack of mutation function.

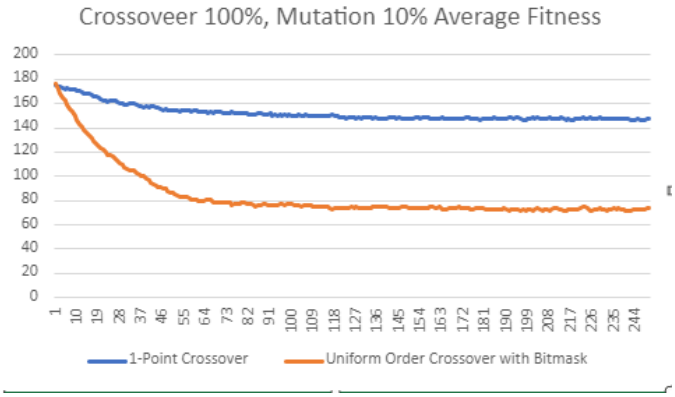


B. Crossover Rate of 100 Percent and Mutation Rate of 10 Percent

With a higher percentage of mutation the graph shows many differences. It shows a more jagged downward trend for both uniform order crossover and for 1-point crossover. It also shows that when the algorithm has a good solution the mutation function does what it's suppose to do and provides diversity. This difference shows when comparing the graphs with 10 percent mutation to the graphs with zero percent mutations. In the graphs with no mutations it shows that the functions hit a rock bottom and doesn't change for hundreds of generations. With mutation the graph shows that even though it gets stuck for 50 to 100 generations it find a new solution that is better. In this graph notice both crossover implementations came to a relatively good solution.

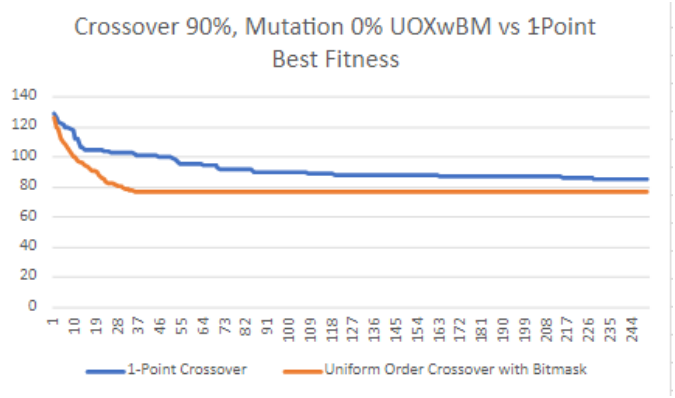


The average fitness graph looks very similar to the graph from the previous subsection. It shows that the average fitness of the population in each generation is much better in uniform order crossover with bitmask compared to 1-point crossover. Continues to prove the theory that the uniform order crossover is better at providing optimal children than 1-point crossover

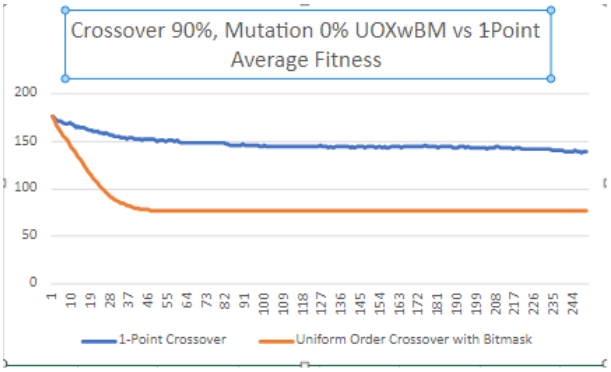


C. Crossover Rate of 90 Percent and Mutation Rate of 0 Percent

In this graph it shows a lot of similarities to the other graph with no mutation rate. The graph show a lot of plateaus especially in the 1-point crossover graph. In this example it shows both graphs come close to finding a solution that is similar in fitness. This is unlike any of the graphs that have seen thus far but shows that 1-point crossover can perform as well as or potentially better than uniform cross over in certain scenarios. Even though the algorithm can not use the mutation function, as the generations goes on the 1-point crossover graph still exits it's local minimum and was able to find a better solution. This could be attributed to luck but also proves a high mutation rate is not required.

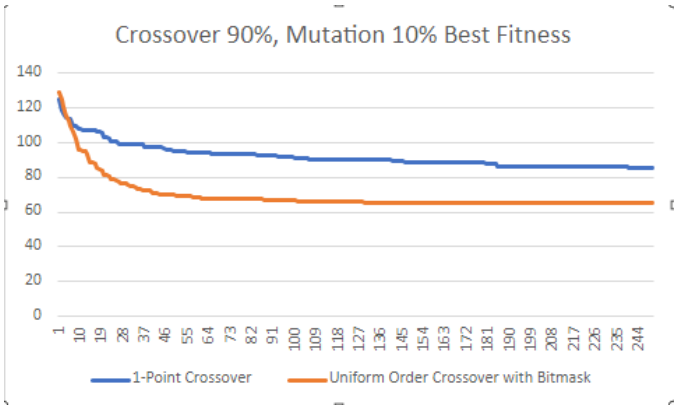


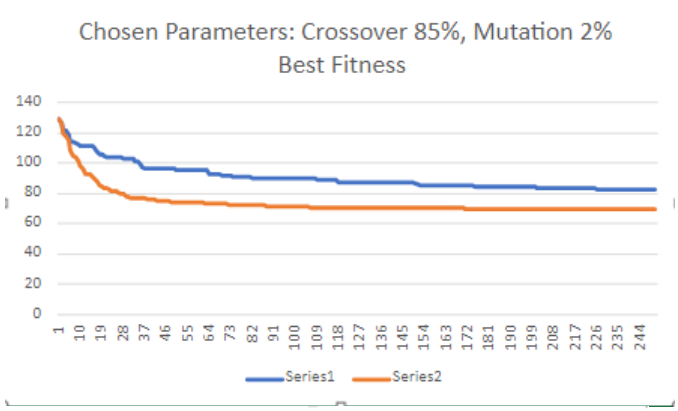
The average fitness graph looks very similar in shape as previous average fitness graphs. With 1-point crossover slowly getting better as generations increases, and uniform order crossover getting sharp increases at earlier generations and then plateauing.



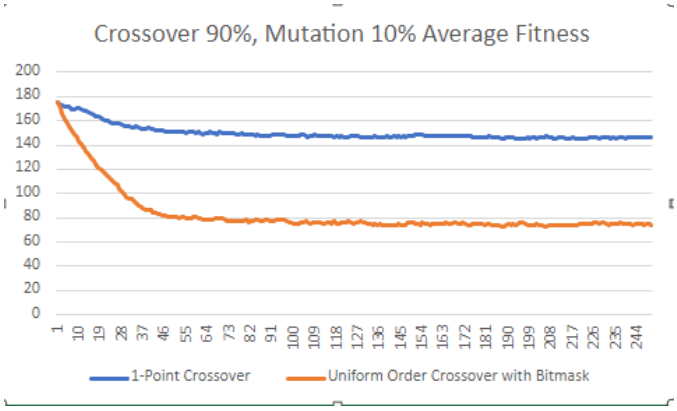
D. Crossover Rate of 90 Percent and Mutation Rate of 10 Percent

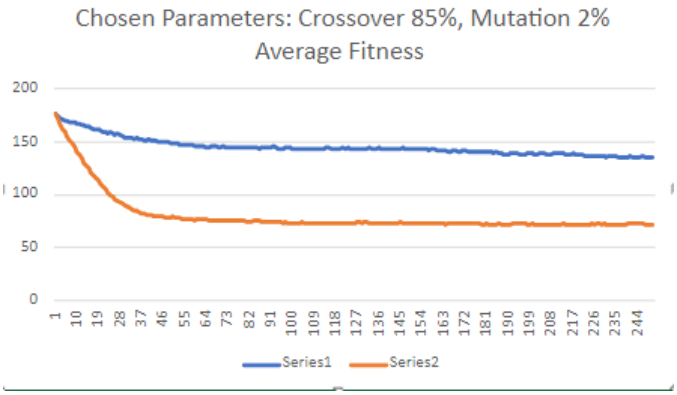
This graph with 90 percent crossover rate and 10 percent mutation rate has a lot of similarities and a lot of differences. The graph shows that the mutation function helped the 1-point crossover line get out of multiple local plateaus. It also shows that it didn't have much effect on the uniform order crossover with bitmask's line. The uniform order crossover with bitmask has very good early generations and slowly plateaus once it runs out of better chromosomes it can find. This line is the ideal genetic algorithm graph and it closely represents a negative log functions graph.





Like the majority of the other average fitness graphs this one looks very similar. The shape is almost identical in each of the average fitness graphs. This graph solidifies the theory that on average the uniform order crossover with bitmask provides more optimal children.





E. Empirically Decided values for Crossover and Mutation

The values for this best fitness graph were empirically decided. The empirical evidence showed that the best combination of crossover rate and mutation rate were a high percentage of children went through crossover and a very low percentage went through mutation. This makes sense as it aligns with Charles Darwin's theory of evolution. This evidence is also applied in nature. If humans have too similar of chromosomes and genes they are susceptible to diseases and illnesses. Humans also wouldn't want a high mutation rate as it could lead to abnormalities that interfere with living.

The empirically chosen parameters were 85 percent crossover rate and 2 percent mutation rate. The graph shows a good resolute for both uniform crossover and 1-point crossover. The algorithm is given a very small chance to mutate in order to break local minimums but treads a fine line with too high of a mutation rate that could negatively impact the results. In this graph both crossover methods show a quick decrease in fitness in early generations that eventually plateau.

V. Discussions and Conclusions

In order to get these result many experiments were made. The algorithm was run a lot of times with really extreme values in order to see how it would perform. At first, the algorithm was ran with 10000 POPSize and 10000 generations. This was in order to see if there was any errors that didn't show up within a small amount of time but would become more obvious as the program ran. This experiment also was to test how long it would take for the algorithm to compute a lot of calculations. The tournament value k was set to max of 5, and had a 100 percent crossover rate and mutation rate. This experiment took 22 minutes and got a best fitness value of 74.31760656368134. Another experiment that was done was to figure out what was the best percentages for crossover and mutation rates. The algorithm showed very quickly that a higher mutation rate was counter intuitive. When mutation rate went past 5 percent it had the opposite effect and instead of helping it hindered the algorithm. Crossover rated was the complete opposite where a high percentage was desired and increasing it helped the algorithm.

The results were very definitive from the runs of this algorithm. I often felt that the mutation rates given were too high to help the algorithm. and empirically tested showed that even 5 percent mutation rate was too much and made the algorithm productivity worse. Another thing that surprised me was that 10 percent of elitism was too high. Which made sense after I saw the results. If we carrying over too many chromosomes between generation it hurts the diversity of the population. The uniform crossover crossover with bitmask also surprised me. Even though it is more complex than the single

point crossover I did not expect it to outperform it by so much. The uniform crossover implementation beat the single point crossover in every way possible. After this implementation of a genetic algorithm I probably will not implement single point crossover again because of this comparison. The only functionality I could find for single point crossover is the simplicity of it's implementation.

REFERENCES

[1] Professor Beatrice Ombuki, "Introduction to Genetic Algorithms," In class lectures and online powerpoints, October 20Th, 2021.

[2] Overleaf, Learn latex algorithms, online at https://www.overleaf.com/learn/latex/Algorithms, October 8Th, 2021.