**Introduction**

Lab 4 was an extension of the previous lab. For this lab, I implemented two additional algorithms for the travelling salesman problem. The Tabu Search and Genetic Algorithm techniques. Tabu Search is a modification of the basic local search which attempts to get over local search problems by not getting stuck in a local minimum. This algorithm is achieved by allowing the acceptance of non-improving moves in order to avoid being stuck in a locally optimum solution and in the hope of finding the global best. This algorithm also gives us the ability to escape from sub-optimal solutions by the use of the tabu list. A tabu list is a list of possible moves that could be performed on a solution. These moves performed could be swap operations as used in this project or other forms of moves include, subtractions, additions in the case of dealing with numeric optimization problems. For the tabu search technique, if a move is accepted, that is the new best solution is found, it’s move is made a *“tabu”* for a certain number of iteration, which for this project, is a fixed number (numOfPerm defined in the header file of my Tabu class) when a move is made tabu, it means we cannot perform the same move for that number of iterations. If a move is found to be tabu, it is added to the tabu list with a value called listSize in my project which basically signifies the Tabu Tenure or length. With each iteration, only when they tabu tenure of a certain move is zero (0) can the move be performed and accepted. For the project, I implemented two different techniques to analyze the neighborhood and tabu list. For neighborhood identification, I have a function that searches and stores the best solution for the shortest path as well as a function that goes thorough the neighbors of the nodes to find the best solution. For the tabu list, I made a simple implementation, where I can easily change the size for the tabu list and test. This change is done in the header file and is independent of the other functions.   
 We also had to use the genetic algorithm technique to solve the travelling salesman problem. Genetic algorithm is a search technique used in computing to find true or approximate solutions to optimization and search problems. Genetic algorithms are categorized as global search heuristics. They are a particular class of evolutionary algorithms that use techniques inspired by evolutionary biology such as inheritance, mutation, selection, and crossover (also called recombination in some articles). This algorithm was more challenging than I anticipated. Although there were a number of available online resources, I had a hard time fully understanding the logic of the algorithm and how some of its methods worked. I was able to implement the algorithm, after multiple attempts and long hours, I was able to come up with an implementation for the algorithm.

**Algorithm Techniques**

My algorithm starts from a population of randomly generated population of individuals or nodes and occurs in generations. In each generation, the fitness of every individual in the population is evaluated, multiple individuals are selected from the current population (based on their fitness) and modified to form a new population. The new population is used in the next iteration of the algorithm. The algorithm terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population. I implemented several techniques for selection, mutation and crossover. For selection, I created an enumeration where I have a selectGA variable that has 3 types of selections (Scaling, Elitist and Roulette wheel) the scaling method uses a random generator for selection while the elitist attempts to find the best solution based on the fitness of the individuals analyzed from the population. The roulette wheel is somewhat of a biased method where each individual gets a piece of the wheel, however, parents with more fitness get a large piece compared to the less fit ones. I use a random generator for randomness of the fitness probability. For the mutation, I implemented a random unique distribution and generator which basically achieves mutation through random swapping of nodes. For crossover technique I made use of a random generator and used a cutoff method for deciding what individuals or nodes are able to cross. For this technique, two random cross points are selected, Genotype from parent1 that fall between the two cross points are copied into the same positions of the offspring. The remaining genotype order is determined by parent2. nonduplicative genotype is copied from parent2 to the offspring, beginning at the position following the second cross point. Another technique I used was mutating nodes before crossing, making use of a mutation rate for comparison with the random number, which is basically cyclic crossover technique. The random crossover

A screenshot of a cell phone

Description automatically generated

**Architecture Design/Modifications**

I made changes to my Lab 3, I realized my design architecture was not accurate and so I modified that. My design architecture makes use of the components for the factory and strategy design pattern with a common algorithm interface that handles the execute function which is overridden in the specific class it is implemented, as shown in the UML diagram below.

A screenshot of text

Description automatically generated

**Figure 1:** UML diagram for design architecture. (A raw expanded version can be found on the repository.)

For fixing the design pattern, I created an AlgoFact class which I hold my factory method and I modified my algorithm class to become an abstract class by adding virtual functions. This modification helped my clean up my code as my main function contains only declaration to the abstract and factory method. The interface handles the execution of the project and I just have a for loop for going over what algo\_type to run. This helps improve extensibility as I am able to add new algorithms and functionality without breaking the already existing class functionalities. I had problems moving the file input handler function to a separate class for the naïve and dynamic program algorithms from previous lab. However, I was able to implement a Map that utilizes the file input handling for the newly added algorithms, by using the struct method that contains the input file. The Solution class extends the Map class as it helps in computing paths and cost of the algorithms. In implementing the variations of the newly added algorithms, I decided to keep the number of permutations for the tabu search technique at a fixed number (this can be easily modified) also for the genetic algorithm, I make use of a population size of 1500 and a mutation rate of 0.6 with the number of permutations similar to that of the tabu search technique.

**Analysis and Results**

During analysis, I noticed that changing the selection method changed the rate at which the genetic algorithm was executed. The timing was different, from best to worst (Elitist, Scaling and Roulette wheel) I also noticed that changing the size of the tabu list affected the speed of execution for the tabu search, as increasing the size from 10 to 200 added significant amount of extra time to the execution time. And I believe these changes to the output were generated due to the increasing number of size and for the genetic algorithm, the various techniques make use of different ways om handling selection, crossover and mutation. Although, I noticed that with less than 5 nodes, my scaling technique for selection had the best timing based on performance, I assume there is only a small subset for the random numbers generated to analyze.

For my graph and tables, I used the Elitist technique for my selection, and the random technique for mutation and cross over technique is selected based on the value generated. The maximum nodes analyzed on my computer for all algorithms is 13. For uniformity, I obtained the graph using the same number of nodes across all algorithm, starting with 4 nodes per the lab instructions. Results and screenshots below:

A screenshot of text

Description automatically generated

**Figure 2:** Output of all algorithms with 4 nodes

A screenshot of a cell phone

Description automatically generated

Figure 3: Output of all algorithms with 8 nodes

With a quick look at the screenshot above, you notice the path of nodes are not printed in the same order, I had this same problem in the previous lab and I was unable to address it in this lab, but I believe the results are somewhat accurate as the shortest path obtained is for the given path of nodes displayed and so the paths are different for each algorithm.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **# Nodes** | **Tsp Naïve** | **Tsp DP** | **TspTabu** | **TspGa** |
| 4 | 84 | 52 | 5543 | 13624 |
| 5 | 91 | 54 | 10961 | 14810 |
| 6 | 107 | 58 | 19583 | 17332 |
| 8 | 780 | 63 | 50536 | 19666 |
| 10 | 49153 | 143 | 87001 | 18905 |
| 13 | 746048 | 220 | 199789 | 22386 |

**Table 1:** Nodes for Algorithms and time in Microseconds

**Time Complexity**

For time complexity, I generated a chart with the table above, and due to the fact that I had a limited number of nodes tested, I decided to plot a linear fit curve as well as a logarithmic curve for better viewing and analysis, charts are shown below;

A screenshot of a cell phone

Description automatically generated

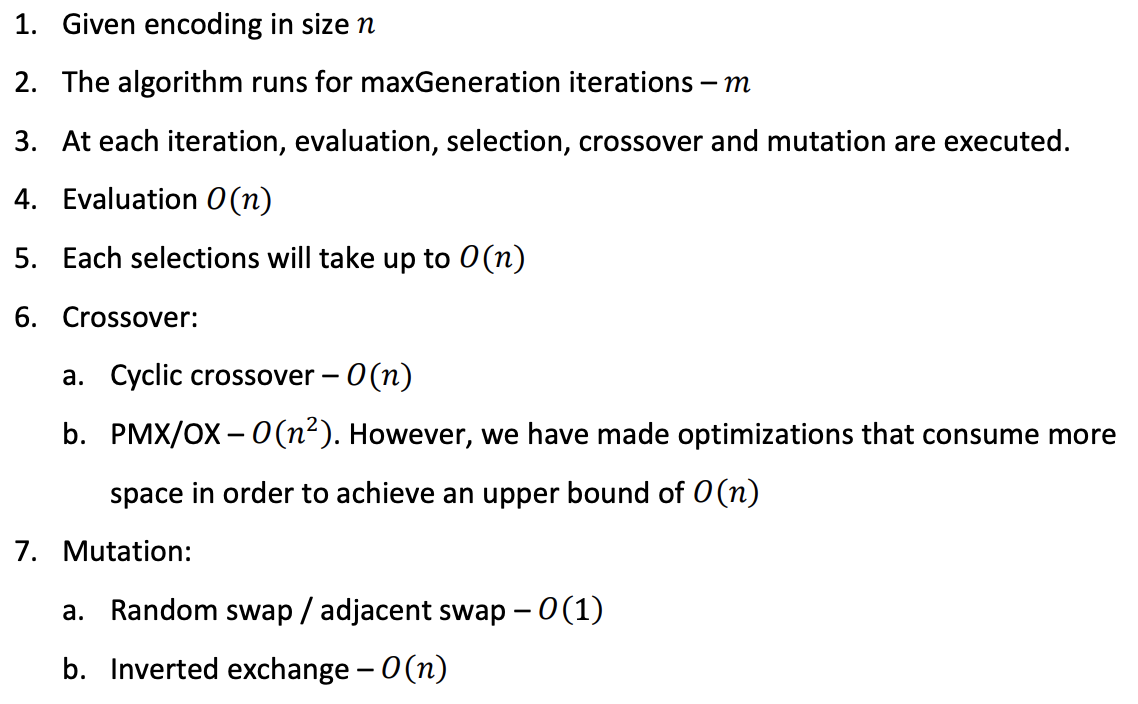
**Chart 1:** Linear Graph of Nodes vs Time

A screenshot of a cell phone

Description automatically generated

**Chart 2:** Logarithmic Scale Graph of Nodes vs Time

Analyzing the charts above, we are able to infer the time complexity of the algorithms in this project, in previous labs, we discussed the time complexity for the naïve and dynamic programming algorithm for the travelling salesman problem. Recall that, the travelling salesman problem (TSP) is an NP- hard problem. The brute force (naïve) algorithm for this problem takes **O (nn)** time. while using the dynamic programming algorithm, we see understand that there are *n* possible start vertices and 2n possible subgraphs, hence the function will be called on at most n ⋅2n distinct arguments. Each call performs at most O(n) work, therefore the total work done is **O (n2 2n).** for the tabu search, going through its steps with the algorithm graph obtained, we can deduce that it can theoretically search thorough the nodes in the graph and so I assume there is no time complexity for this algorithm in terms of big O notation.

Lastly for the genetic algorithm, I was having a hard time understanding its relationship to the time complexity, I found a useful article, shown below which breaks down the complexity:



Thus, following these steps and the graph obtained above, we can presume the total complexity for the genetic algorithm for the travelling salesman problem is O (mn).

**References**

<http://www.cs.huji.ac.il/~ai/projects/old/tsp2.pdf>

<http://www.cs.cmu.edu/~02317/slides/lec_8.pdf>

<http://www.umlet.com/umletino/umletino.html>

<https://github.com/CaoManhDat/TSP-TabuSearch>

<https://stackoverflow.com/questions/19346532/uml-class-diagram-c-struct>

<https://github.com/marcoscastro/tsp_genetic>

<https://github.com/GAStudyGroup/TSP>