Andrew Breslauer

In this lab, we implemented Tabu and Genetic search algorithms, which use metaheuristic search techniques to try to solve the traveling salesman problem in fewer steps than through use of brute force or dynamic programming. For both algorithms, I have a fixed number of iterations. Despite the fact that the initial group of solutions is randomly initialized, they do always find the optimized solution for that algorithm’s randomly generated data much faster than either brute force or dynamic programming.

Both Tabu and genetic had a fixed number of iterations, meaning that their timing varied depending on the size of the graph as well as the number of nodes in the graph. As seen by the chart above, the metaheuristic searches did not perform as well at the start but performed much better towards higher graphs. This graph is also somewhat misleading because Tabu and Genetic search did not limit themselves based on the results of the Dynamic Programming and Brute Force searches (mostly because I didn’t realize that was a thing that had to be done until hours before and I didn’t want to risk breaking it). If that had been done, then the two newer algorithms likely would’ve performed much better for the entire graph range, rather than just past the 8/9 node mark. Brute Force and Dynamic were also not run past 10 nodes because of timing constraints imposed by my PC.

**Tabu Algorithm**

For the Tabu list size, setting a fixed value would’ve led to wasted time for smaller graph searches, as well as imprecise searches on larger graphs then the fixed size is meant for. Instead, I scaled the list size with the graph size, so that the range of list values scales approximately evenly with the size of the graph.

As far as neighborhood identification, by far the most successful method was simply performing a single swap on an already established solution. To avoid falling into local mins, each min got a slightly worse fitness value, so that the algorithm can climb out of that min without a problem. The far worse method of finding a neighborhood was to perform mathematical operations on the solution, because this led to many (many, many, MANY) solutions that were invalid, which is just wasting valuable processing time.

**Genetic Algorithm**

**Selection**

I tested three different methods for selection: randomly choosing parents with true random odds, taking the two chromosomes with the highest fitness (elitism), and doing a selection where parents are chosen randomly, but higher fitness chromosomes have a higher chance of being selected (biased random). By testing these options, I discovered that the biased random method reliably had the best results. Elitism was easily capable of getting caught within local minimums, while true randomness meant the genetic algorithm didn’t really learn from previous iterations and thus wouldn’t move closer to an expected minimum. The biased random method had the random element to introduce genetic diversity, but it also preferred higher fitness solutions, which caused the algorithm to converge on the absolute minimum.

**Crossover**

I tried several different methods of crossover, such as applying a mathematical operation based on the two parents to generate a child, as well as randomly taking from one parent or the other for each element of the child. The method that I found to be most effective was to choose a random point in the solution and copy from parent 1->child 1 and parent 2->child 2 until it reached that point, then swap to parent 1->child 2 and parent 2->child 1. The reason that this solution was effective is that it conserved parts of the path which might be optimal, unlike the other solutions. It also had a fairly high rate of success, where a successful crossover generated a valid solution.

**Mutation**

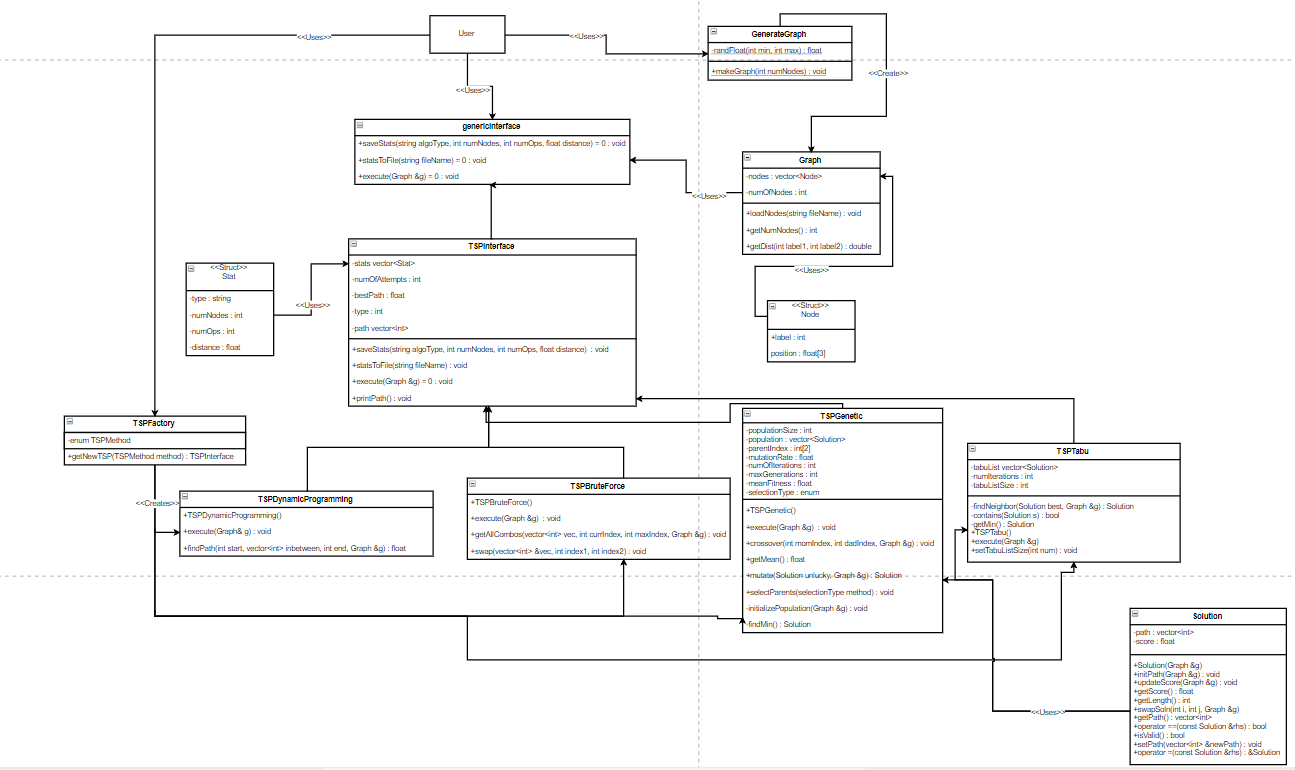
I tried changing one random element of one solution, as well as performing a mathematical operation on every element, but the low rate of mutation coupled with the very high failure rates of those methods led me to change to selecting two elements at random and swapping them. This mutation always generated a valid solution, so it was able to create a new solution every time. These new solutions introduced genetic diversity into the population, which is important to help it escape local minimums. My mutation rate was fairly low, since if it was very high, it annulled the benefit of having non-random crossover.

Other changes I made to genetic were the population size. Generally, a larger population size required fewer iterations to reach the optimal result.

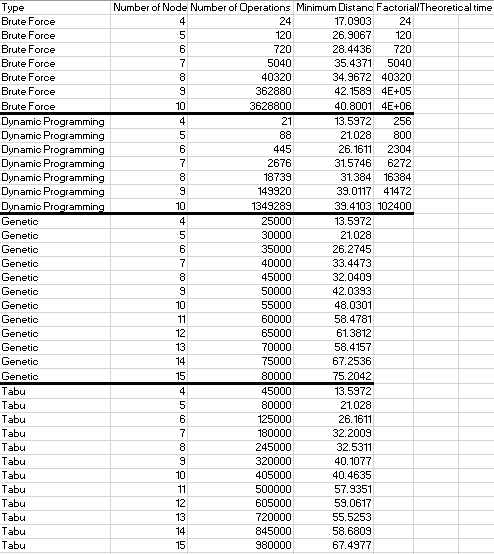
**Design**

Since my code was meant for reusability, I used the strategy pattern to implement my travelling salesman algorithm. I also used the factory pattern to create the instances of the algorithm objects. The idea behind the strategy pattern is that I would be able to use the same interface to implement other classes in the future. The combination of the factory pattern and strategy pattern abstracts a lot of the functionality away from the user. For example, all the user does to create an instance of a genetic algorithm traveling salesman problem is make a factory class and call the method to create a new class, passing in the correct enum as defined within the factory. An example of this expandability is the fact that this is the exact same file setup as Lab 3, just with two new sorting classes (as well as a helper “Solution” class).

**UML**



This UML diagram shows the classes that I used to make my program. The user only interacts with the factory class, the generateGraph class, and the interfaceInterface. This abstraction makes it easy for the user to dynamically change what type of algorithm that they want to use, as well as limits the number of objects that they are handling at once. Other algorithms that I implement in the future can use the same genericInterface class, meaning that the user will not need to use different syntax to use the new algorithms. A larger version of this image is included in the report folder.



This table shows some collected information for the search methods for graph sizes of 4 nodes to 10/15 nodes. Brute Force and Dynamic were also compared to their expected values, because those can be calculated mathematically due to a lack of randomness. One unexpected thing that leads me to doubt the accuracy of my algorithms is that the minimum distance for brute force is consistently higher than the other 3, which means brute force is calculated incorrectly or the other 3 algorithms don’t loop back to the beginning.