Lizzy Hanna
CS 3353 – Algorithms
12/5/19

Lab 4

This lab explored heuristic and metaheuristic algorithms for the NP-Hard Traveling Salesman Problem, specifically a Tabu Search Algorithm (Glover 1986) and a Genetic Algorithm (Holland 1975). I found that my Genetic Algorithm outperformed my Tabu search up until around 15 nodes, when Tabu became the more efficient choice. Both Tabu Search and my Genetic Algorithm were more efficient than Brute Force at greater than 7 nodes. However, my Dynamic Programming algorithm from Lab 3 outperformed both consistently, as shown below in Figures 1 and 2.

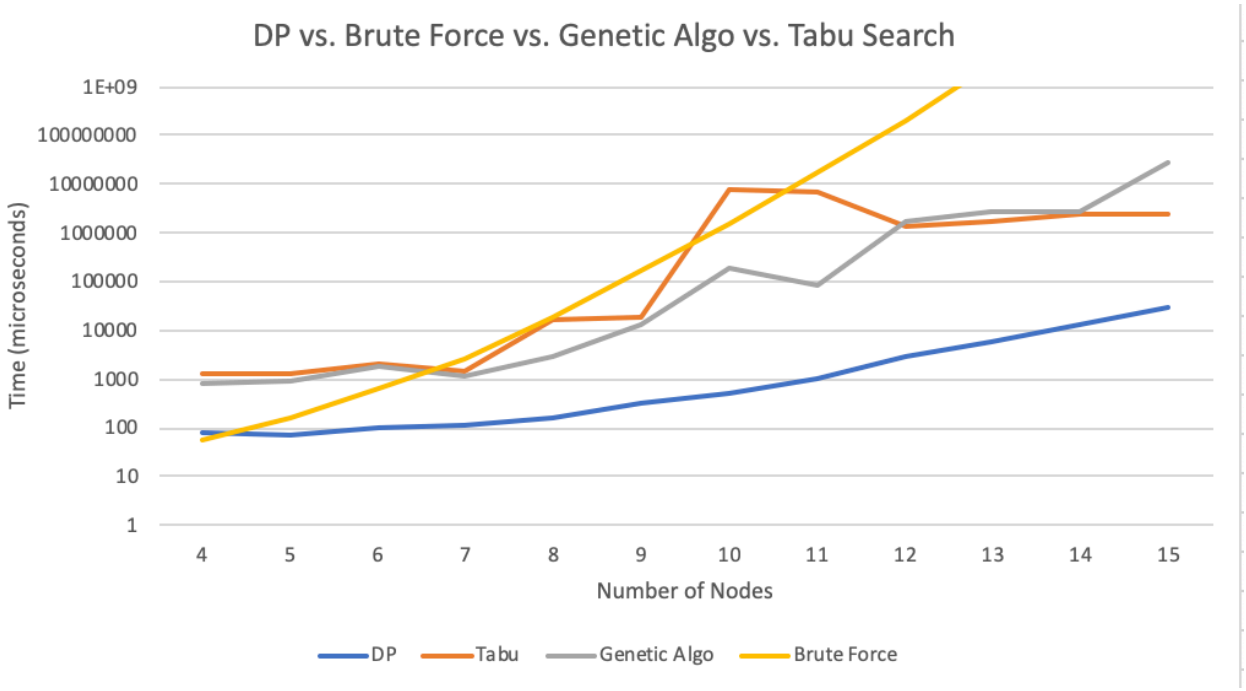


Figure 1. All methods (Dynamic Programming, Brute Force, Genetic Algorithm and Tabu Search) graphed logarithmically by number of nodes versus time taken to find solution.

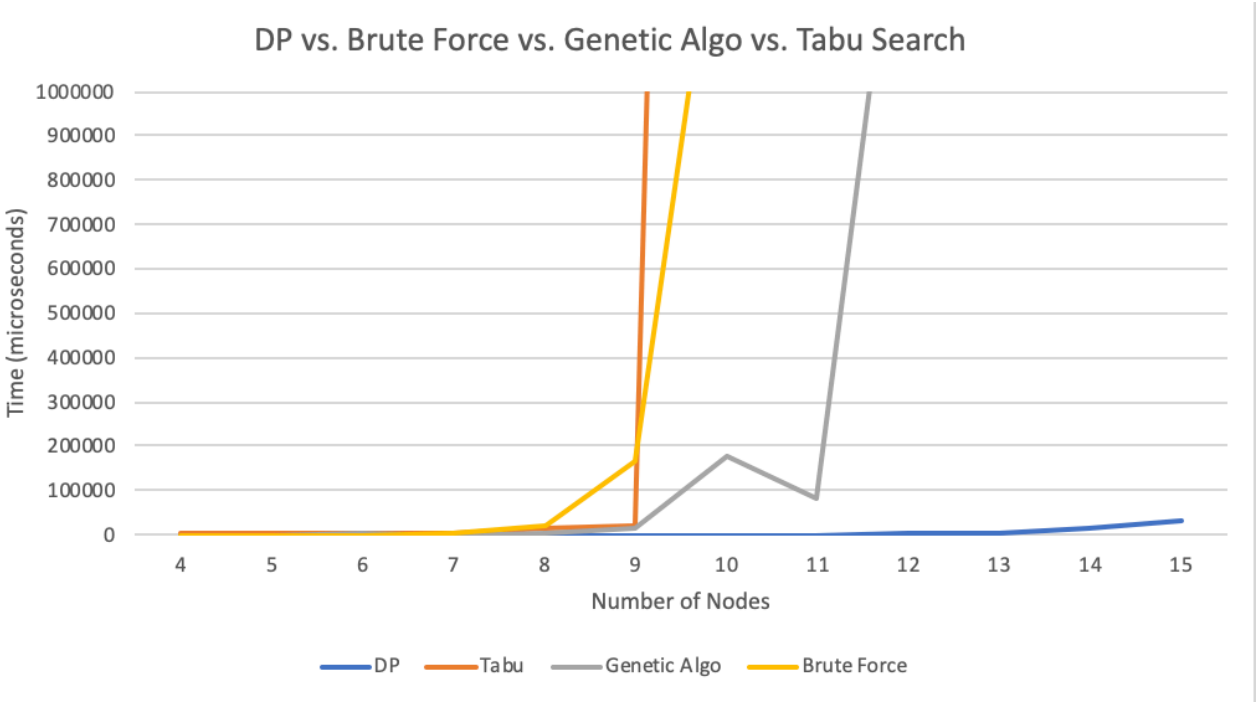


Figure 1.2. All methods (Dynamic Programming, Brute Force, Genetic Algorithm and Tabu Search) graphed by number of nodes vs. time taken to find solution (y axis is not logarithmic.)

Node Size	DP timing	Tabu Time	GA time	Brute Force time
4	77	1268	824	59
5	75	1336	860	166
6	101	2060	1785	617
7	112	1452	1211	2599
8	158	16486	2805	19364
9	304	18232	12805	167884
10	530	7502668	180141	1590186
11	1064	6989450	80554	17363993
12	2939	1288140	1674688	199244000
13	6153	1736970	2725815	2.90E+09
14	12856	2.40E+06	2694742	
15	30713	2.33E+06	26426031	
16	57604	1.42E+06		
17	142887	4.80E+06		
18	352892	1.27E+07		
19	1059585	7.01E+06		
20	2427056	1.08E+07		
21	5134791	2.43E+07		
22	13728760	1.23E+07		
23	29805781	2.82E+07		

Figure 2. Table of results from Lab 3 and Lab 4.

The maximum number of nodes Genetic Algorithm could feasibly calculate in a reasonable time frame was around 15. Tabu was much faster and could go up to 23 and even

beyond with the right seed. As for timing complexity, both heuristic algorithms are non-deterministic because they rely on random processes so it is difficult to assign a Big O timing complexity to the algorithms. They lie somewhere roughly between the range of O(n!) of brute force and O(n^2logn) of dynamic programming as seen on the graphs. However, the random element of the algorithms can make different trials finish at completely different times. If the shortest distance is not already known, these algorithms will run forever, so I believe we are unable to assign a time complexity to the algorithms.

For the Genetic Algorithm, I tested two different techniques each for mutation, selection, and crossover then tested all the combinations of the techniques to find the most efficient version of my algorithm. I graphed each learning curve and then compared them to see if any were actually learning. I found that only one graph (technique 5) appeared to have any intelligence.

Mutation Techniques

- **Mutation Technique 1** consists of taking a path and swapping 2 nodes.
- **Mutation Technique 2** takes a path and performs what I refer to as an "intense" shuffle on it, which is where I swap 2 random nodes 100 times and reverse the path after each swap to try and get as far from the original path as possible.

Selection Techniques

- **Selection Technique 1** uses a roulette style selection to generate the parent(s). The fitness is calculated by finding the additive inverse of each distance. To find the additive inverse, each distance is subtracted from the largest distance of the current population, so that distances that are further from the largest have a higher fitness score. The equation is **fitness = largestDistance – currentDistance + 1.** For example, if the highest distance is 10, then a distance of 10 would have a fitness score of 1 and a distance of 2 would have a fitness score of 9. Then, I normalize the fitnesses by distributing the fitnesses across an array. The roulette function then chooses a random number between 1 and the last value in the array (which is the sum of all fitnesses), then cycles through the array until the random number is less than the value at the current index. That index is then selected.
- **Selection Technique 2** finds the best solution(s) of the current generation to serve as the parent(s). For two parents, this finds the best and second-best paths to assure that the two parents are not identical. However, I think this would hurt the algorithm if there are two different paths with the same best distance in a population and only one is selected.

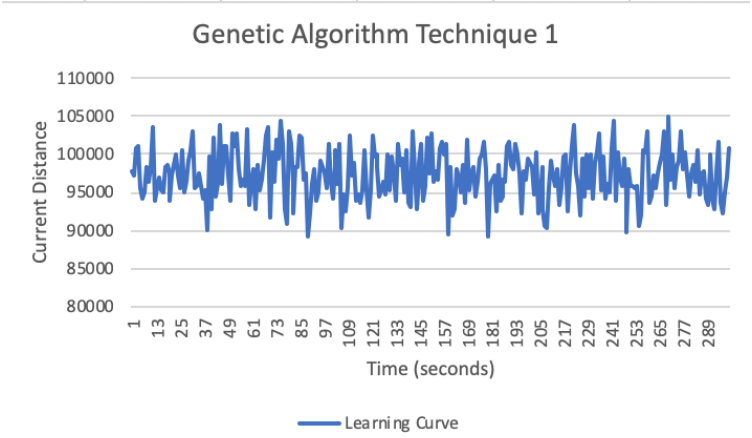
Crossover Techniques

- **Crossover Technique 1** finds the largest common substring between two parents and inserts the substring into the child nodes at random locations, then fills in the rest of the child node from elements randomly selected from one parent that are not included in the substring. For example,

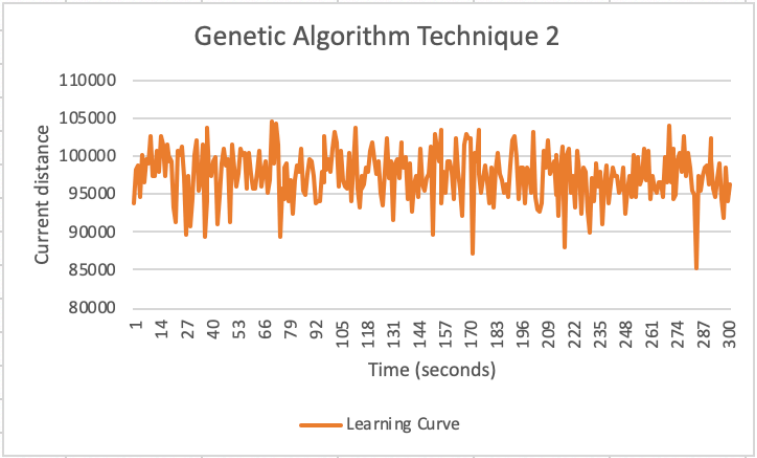
  Parent 1: [1,3,5,2,4]
  Parent 2: [1,5,3,2,4]
  LCS: [2,4]
  So the children could look like:
  Child: [1,2,4,3,5]
  Child: [1,3,2,4,5]
  And so on for the size of the population.
- **Crossover Technique 2** only uses one parent to generate children by swapping elements within the parent.

Genetic Algorithm Learning Curves

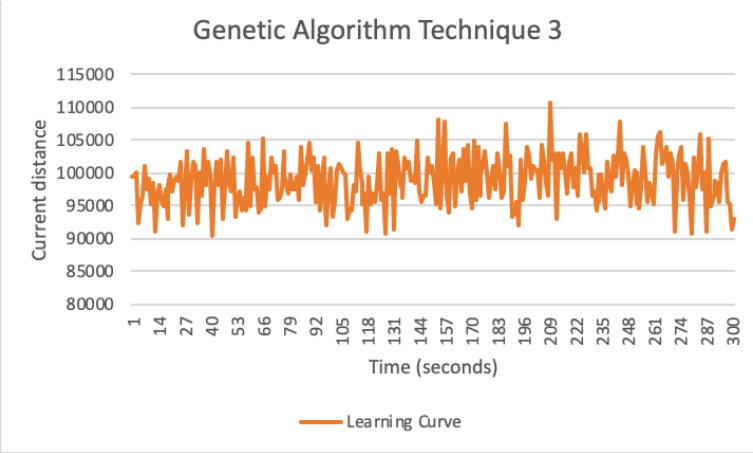
- Technique 1
  - Mutation Technique: 1
  - Selection Technique: 1
  - Crossover Technique: 1



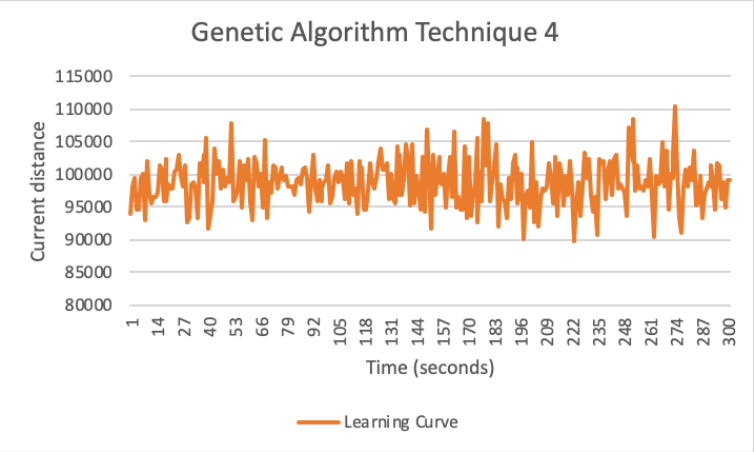
- Technique 2
  - Mutation Technique: 2
  - Selection Technique: 1
  - Crossover Technique: 1



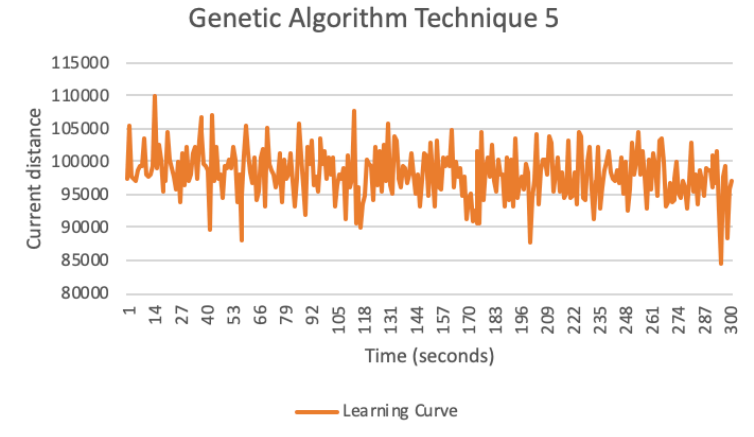
- Technique 3
  - Mutation Technique: 2
  - Selection Technique: 2
  - Crossover Technique: 1



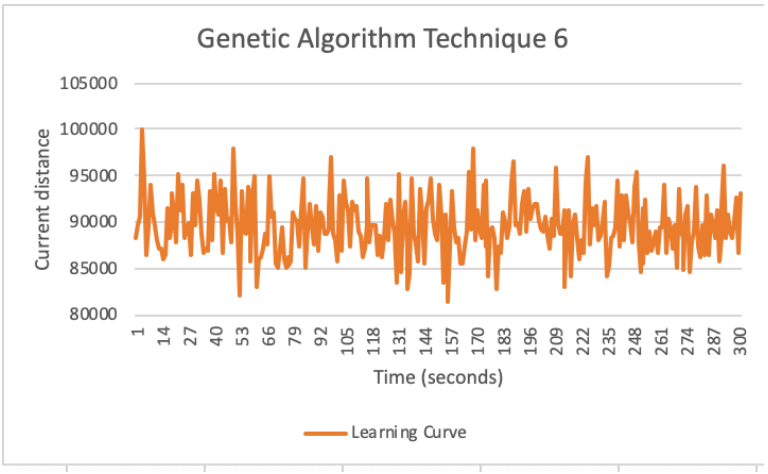
- Technique 4
  - Mutation Technique: 2
  - Selection Technique: 2
  - Crossover Technique: 2



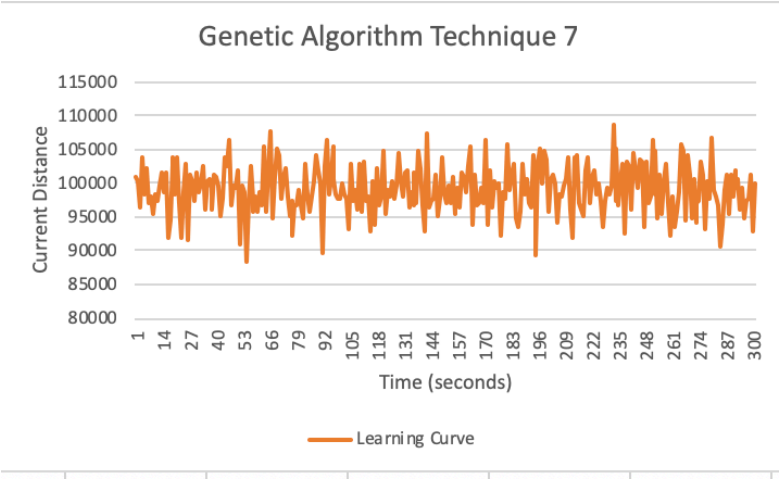
- Technique 5
  - Mutation Technique: 1
  - Selection Technique: 2
  - Crossover Technique: 1



- Technique 6
  - Mutation Technique: 1
  - Selection Technique: 1
  - Crossover Technique: 2



- Technique 7
  - Mutation Technique: 1
  - Selection Technique: 2
  - Crossover Technique: 2



Although the techniques all proved nearly totally unintelligent, Technique 5 usually managed to achieve a better distance in the same time span as the other algorithms throughout my trials, so I used Technique 5 when doing my Genetic Algorithm timing comparisons with the other algorithms (Brute Force, DP, etc.). In the graphs above, I ran the algorithm for 300 seconds on a graph size of 100 nodes. Below I have the learning curve for Technique 5 after running for 10 minutes on the same 100 node graph (Figure 3) with output every second, along with the graph of the best distance versus epoch number (Figure 4).

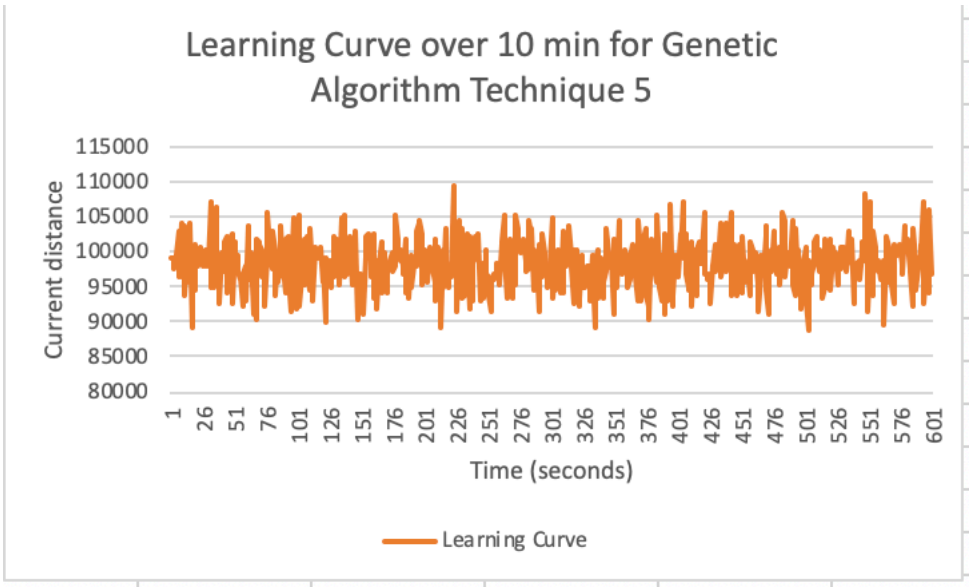


Figure 3. Learning curve of Genetic Algorithm Technique 5 time versus current distance.

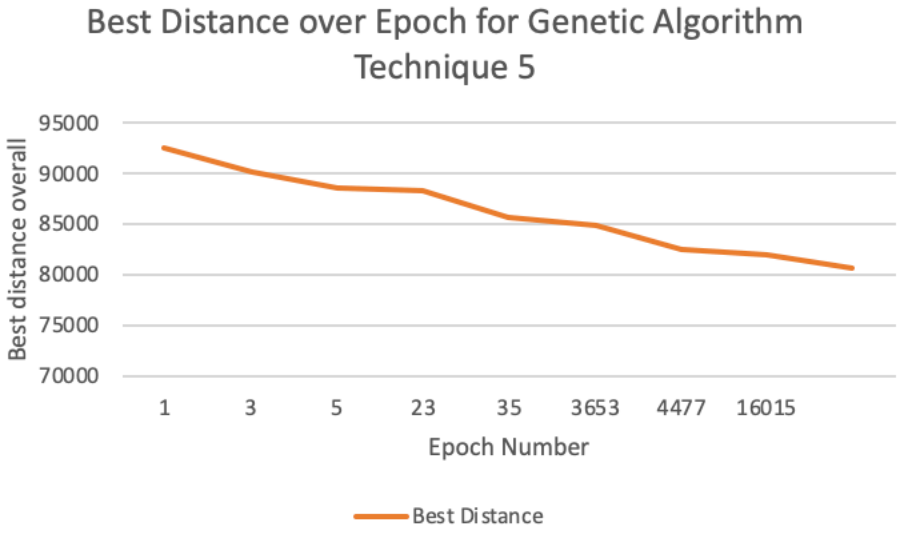


Figure 4. Graph of best distance versus epoch for Genetic Algorithm Technique 5.

I adapted my Genetic Algorithm code many times to try to improve its intelligence and yet it still does not appear to learn over time. Some solutions I tried were finding the substring between the current best path and the current parent, then using that to generate children; doing a mix of elitism and roulette where roulette is only used to select a parent a certain percentage of the time; and changing the mutation rate to try and improve the performance. I honestly still am unsure why my code behaves so erratically even after implementing the LCS algorithm and will continue to research different techniques to implement in the hopes of improving its intelligence. Since every combination of techniques ended up with similar and

poor results, I cannot verify what differences the different techniques make on its performance. In theory, I believed that an algorithm that implements a crossover using the LCS of two roulette-selected parents with a 10% chance of mutation via extreme shuffling would perform the best, but my results did not reflect that. Some possible theories for why my algorithm behaved unintelligently could be that I was using elitism wrong by selecting the best and second best parents, did not implement the LCS algorithm correctly, my mutation rate is too high, or my population size is too small. I may just need to research better crossover methods as well.

For Tabu List Search, I tested two different methods each for neighborhood identification and tabu list size. I also implemented a feature that would cause my Tabu List to randomly restart if the global best distance has not updated after 2000 attempts or if all the neighbors in the current neighborhood are on the Tabu List. The solution that the algorithm will move to next is selected via a roulette function using the normalized fitnesses (a function of distance) of the neighborhood while excluding neighbors on the Tabu list. The underlying structure of my Tabu list is a deque so that I can easily push back and pop old solutions from the front without having to write a resize function every time the deque reaches the maximum Tabu list size. For my timing demonstration in Figures 5 and 6 below I used a static tabu list size of 150 and neighborhood identification technique 1 for a graph size of 100 nodes.

Tabu List Size

- **Static list size:** I tried multiple different constants for list size, including 15 nodes, 50 nodes, and 100 nodes. I found a list size of 50 worked well for the upper end of graph sizes, i.e. 15 and higher.
- **List size dependent on graph size:** I tried making the Tabu list 3 times the input graph size. I preferred the static list size because for a graph of 4 nodes, a tabu list size of 12 seemed like overkill, whereas for a graph of 100 nodes, a tabu list size of 300 seemed taxing on the memory. This was like a Goldilocks situation in which it was just right for graph sizes in the middle of the range.

Neighborhood Identification

- **Technique 1** uses a function called spotExchange() which will start at the second node of a path and swap that node with every other place after it in the path, then move on to the third node and so on until the neighborhood is full. For example,

  Current Solution: [1,2,3,4,5]

  Neighborhood where neighborhood size is 6:

  [1,3,2,4,5], [1,3,4,2,5], [1,3,4,5,2], [1,2,4,3,5], [1,2,4,5,3], [1,2,3,5,4]
- **Technique 2** takes the current solution and swaps two random elements within it to create a neighbor. I prefer Technique 1 instead because it is more systematic and its

technique closer to more deterministic algorithms like the brute force and DP, whereas Technique 2 is much more reliant on randomness to achieve results.

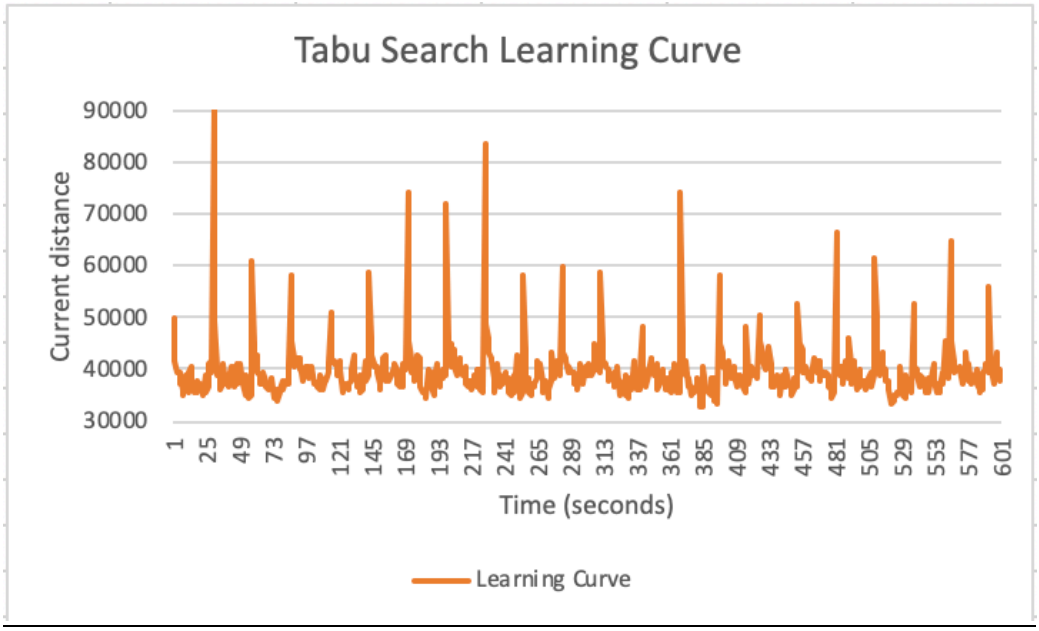


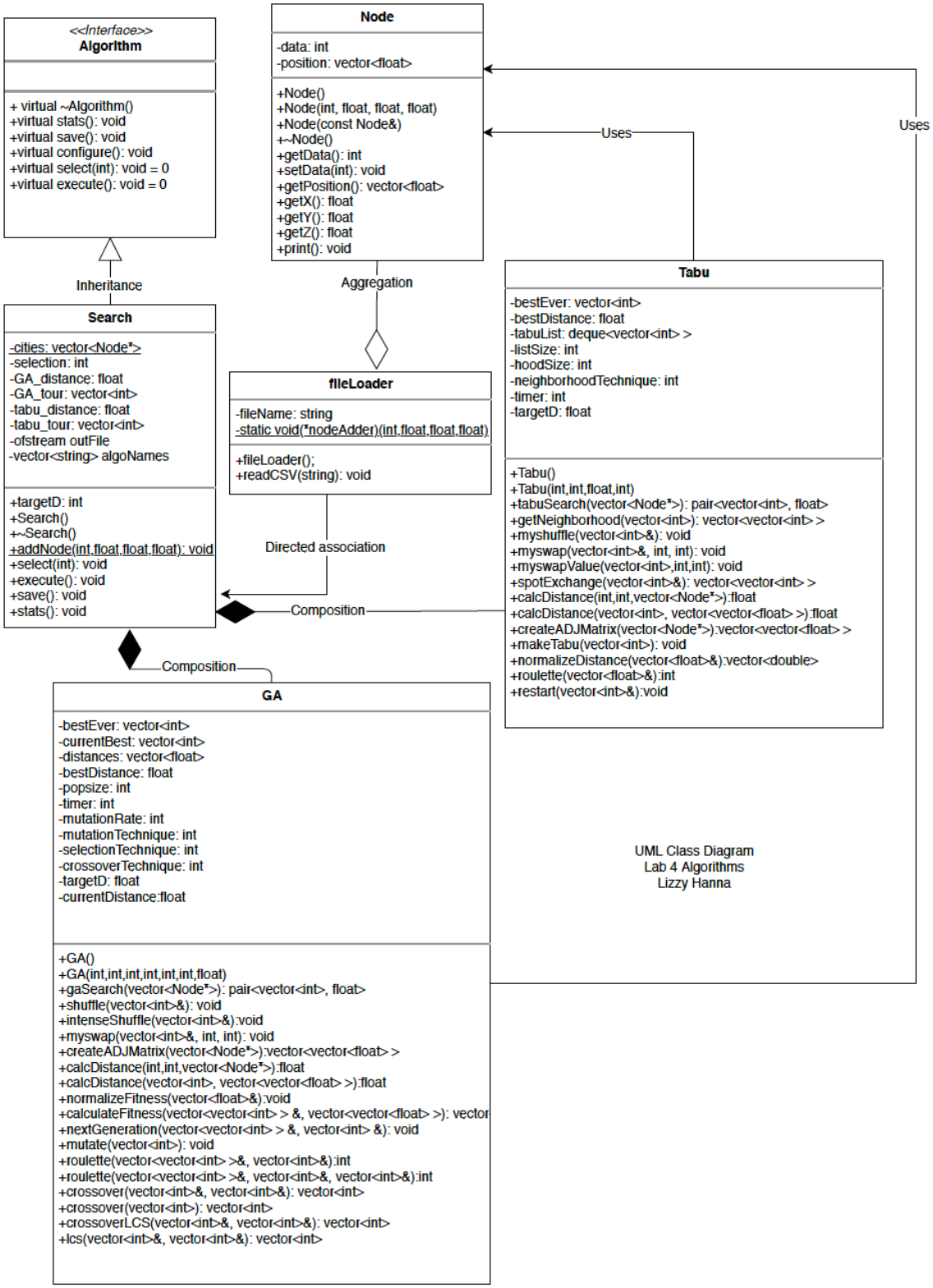
Figure 5. Current distance versus time for Tabu Search over 10 minutes, representing the learning curve of the algorithm.

The spikes in the graph are where the algorithm initiated a random restart, meaning that it may have gotten stuck in a local minima and needed to jump to a random part of the graph for the chance to improve. The steep slope after each spike is evidence that the algorithm is behaving intelligently and seeking better solutions over time. The best distance is plotted over the life of the algorithm below.

Figure 6. Best global distance tracked over the course of the Tabu Search algorithm (10 min)

Design Decisions

.



My UML class diagram gives a layout of the classes I have and how they interact. As in Lab 3, Algorithm is an abstract class, AKA an interface, so it is notated by the word *<<Interface>>* in italics. Search class inherits from Algorithm, denoted by an unfilled arrow pointing towards the parent class. The relationship between Search class and fileLoader class is a directed association (notated with an arrow pointing in the direction of the flow of information) because fileLoader supplies data to the Search class via a function pointer to a function in the Search class and the flow of information is in a single direction only, however fileLoader does not contain any objects of type Search or vice versa so this is not an aggregation nor composition relationship. The Search class contains instances of the Node class but the Node class can live independently of the Search class, so this is an aggregation relationship. Tabu and GA both reference Node in some of their functions so this is an using relationship.

As in Lab 3, the use of static functions and variables (underlined) is to facilitate the use of function pointers between classes to allow for classes to easily provide information to each other without needing to own instances of each other. For example, my fileLoader class has a function pointer to the addNode(int,float,float,float) function in Search so that it can provide data to the Search class before the Search class is even instantiated as an object. Through this, fileLoader never has to know what a "Node" is; it just passes the information to Search and Search does all the creation of Node objects in its static function.

However, I did not use function pointers within my Search class to point to the algorithms in this lab, which is a design departure from previous labs. This is because I found that the GA and Tabu algorithms were so complex with so many different unique functions that I wanted to separate them into their own classes with their own instance variables. The way I change between different configurations of each algorithm is by creating new objects in Seach initialized with the settings I want to use as instance variables, with the set-up happening in the constructor. I found this to be the easiest way to streamline creation of unique configurations of objects without having to make every function static to account for function pointers. Since GA and Tabu each have many functions, having that many static functions seemed like it would not be as memory-efficient as creating objects with those functions inside, so we can delete the objects when we are done. Since Search has the objects of GA and Tabu, this is a composition relationship, since these objects are destroyed when Search is destroyed.