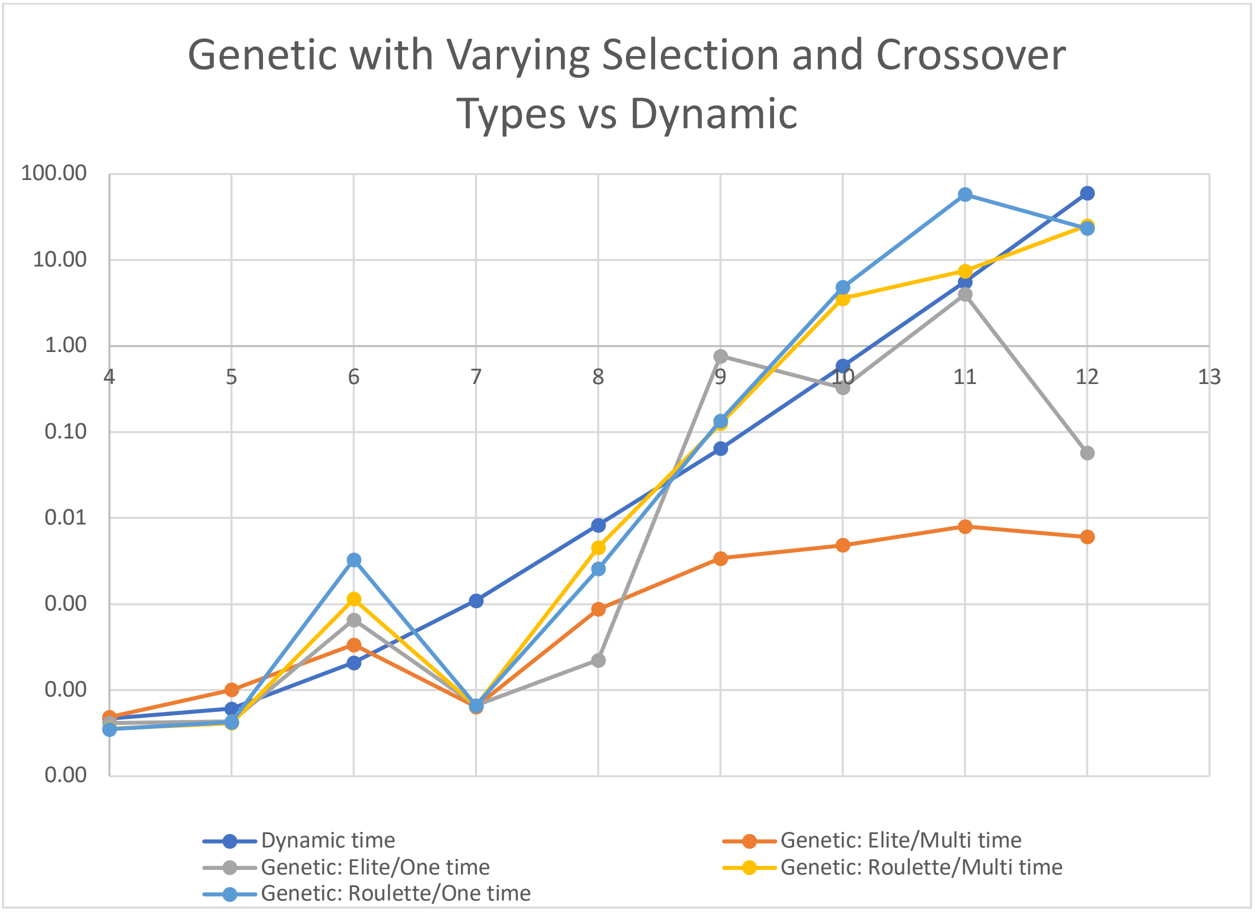
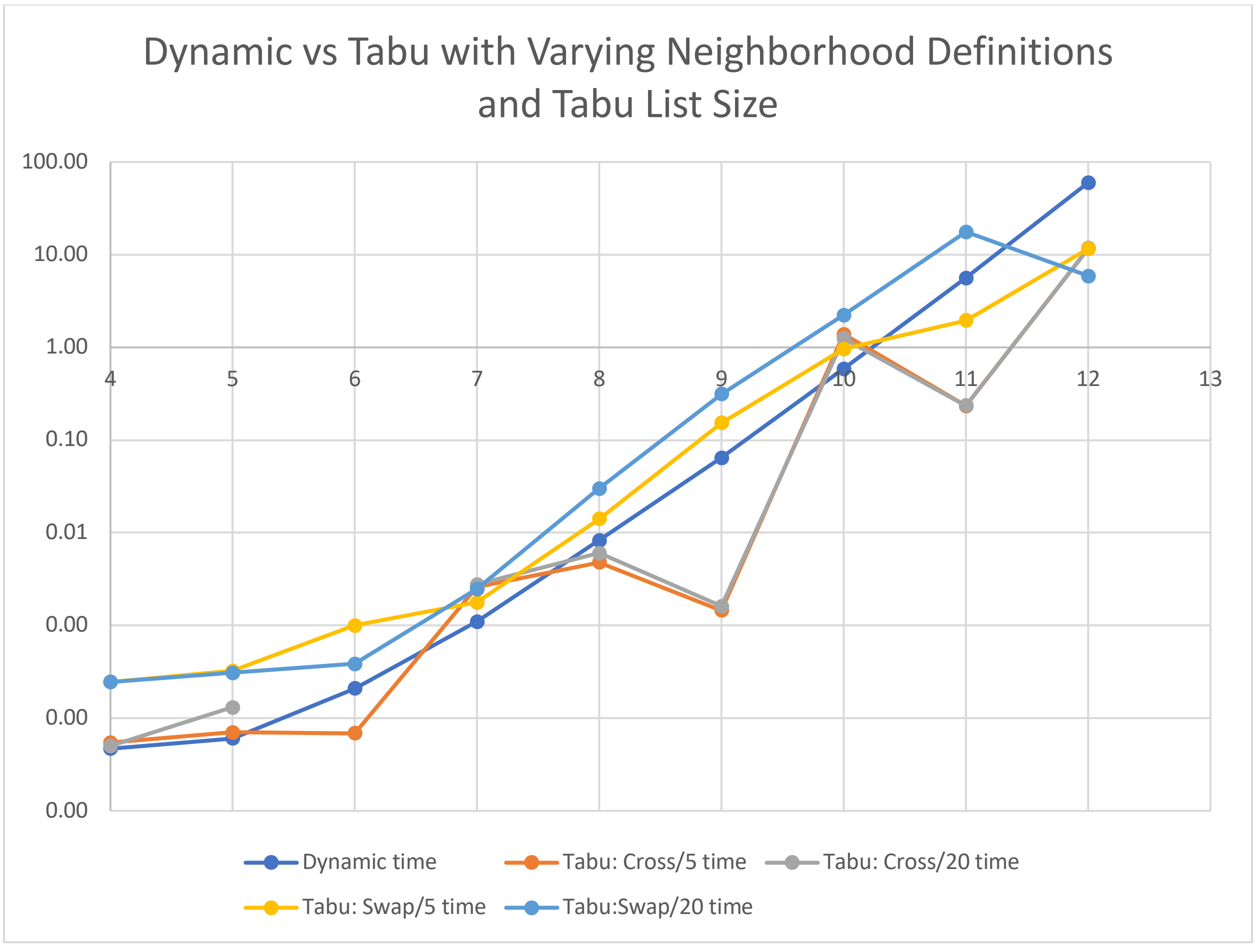
Emily Fashenpour
CSE 3353

Lab 4 Report: Traveling Salesman Problem

Genetic Algorithm & Tabu Search

Timing





Dynamic		Genetic: Elite/Multi		Genetic: Elite/One	
node	time	node	time	node	time
4	4.69E-05	4	4.88E-05	4	4.15E-05
5	6.07E-05	5	0.000101	5	4.34E-05
6	0.000208	6	0.000339	6	0.00065742
7	0.00109973	7	6.41E-05	7	6.61E-05
8	0.00828905	8	0.000874	8	0.000225
9	0.064221	9	0.003398	9	0.766755
10	0.585676	10	0.004828	10	0.328374
11	5.59331	11	0.008032	11	3.99102
12	60.0054	12	0.006077	12	0.057081

Genetic: Roulette/Multi

node	time
4	3.58E-05
5	4.14E-05
6	0.00115
7	6.60E-05
8	4.57E-03
9	0.125902
10	3.58186
11	7.43282
12	25.2583

Genetic: Roulette/One

node	time
4	3.53E-05
5	4.26E-05
6	0.003284
7	6.67E-05
8	0.002577
9	0.136148
10	4.80356
11	57.6953
12	23.3634

Tabu: Cross/5

node	time
4	5.46E-05
5	7.03E-05
6	6.88E-05
7	2.60E-03
8	4.78E-03
9	0.00145
10	1.38885
11	0.234488
12	11.6539

Tabu: Cross/20

node	time
4	5.02E-05
5	0.000131
6	6.4249e-05
7	0.00276244
8	0.00601921
9	0.001609
10	1.25066
11	0.235687
12	11.7105

Tabu: Swap/5

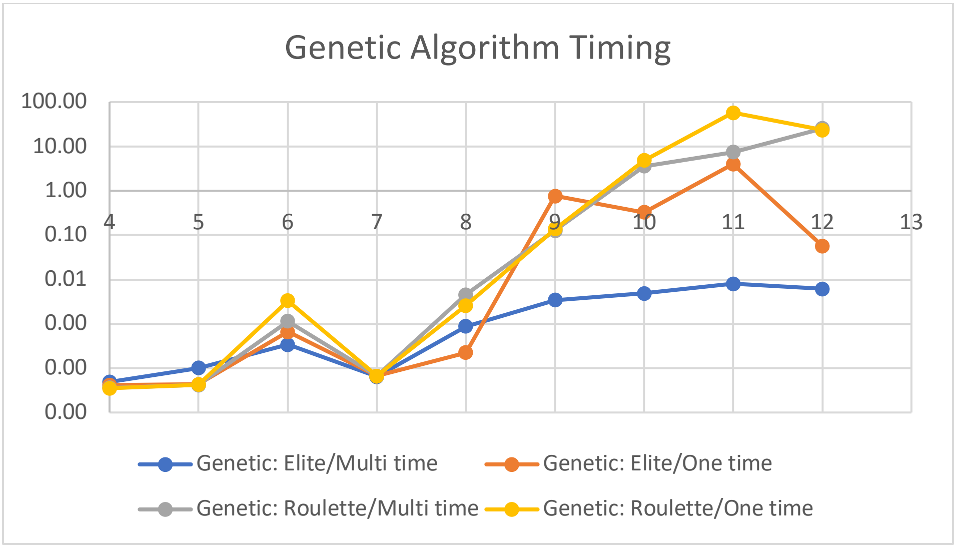
node	time
4	2.45E-04
5	0.00032289
6	0.00100433
7	0.0017739
8	0.014154
9	0.153247
10	0.959346
11	1.94767
12	11.6053

Tabu: Swap/20

node	time
4	0.000245
5	0.00031
6	0.00038499
7	0.002483
8	0.030121
9	0.313972
10	2.23182
11	17.6399
12	5.89754

Summary of Results

Genetic Algorithm



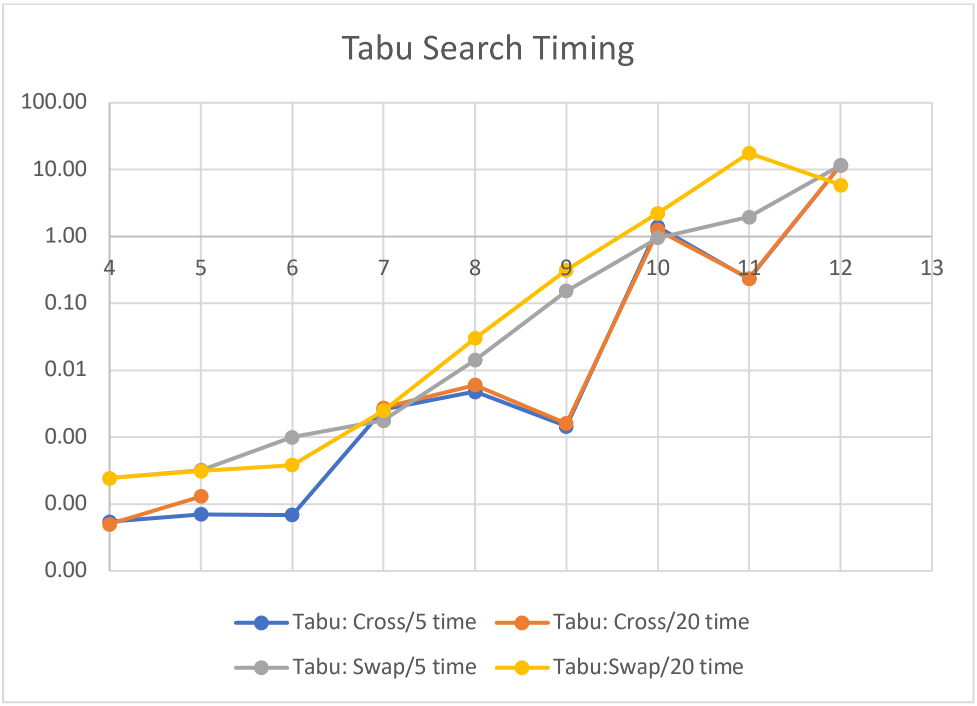
Overall, the best combination of selection and crossover type was Elite Selection and Multipoint Crossover. This makes sense because the best chromosomes move on to the next generation and only a small percentage of the best chromosomes found so far are crossed over with another elitist chromosome, producing some randomness but because the chromosomes that are being crossed are elitist, the probability that the children produced from the crossover are better chromosomes than the parents is higher compared to using the one point crossover or the roulette selection. However, just because this combination performed the best overall based on the data collected and displayed in the table, there is still a chance for another combination of selection, crossover, and mutation rate to outperform elite selection and multipoint crossover. If the initial population that is randomly generated and the randomly selected chromosomes are initially better than another combination type, then the chances of the algorithm finding the best solution faster than another combination is higher. For example, when searching 8 nodes for the optimal chromosome, elitist selection and one-point crossover outperformed elitist selection and multipoint crossover.

There were also times when the algorithm didn't find the best solution in the given generation size and once running the algorithm once or twice more, it did find the best solution. There were also times when the best chromosome wasn't found but by then increasing the generation size the best chromosome was found.

Compared to the dynamic approach to solve TSP in Lab 3, based on the data collected above, dynamic outperformed the GA when there were less than 6 nodes to search. Once there are seven nodes to be searched, GA greatly outperformed the dynamic approach.

Since Genetic Algorithm is a stochastic search algorithm, meaning that the different parameters (selection type, crossover type, and mutation rate) used to find an optimized chromosome use randomness to select which chromosomes to move to the next generation, which parts of a chromosome get moved to a new chromosome and in what order, and which chromosomes get randomly mutated. Because there is so much parameters that use randomness, GA does not have definitive asymptotic complexity. However, GA's complexity is relative to the number of nodes, the number of generations, and the complexity of each crossover type, mutation type, and selection type.

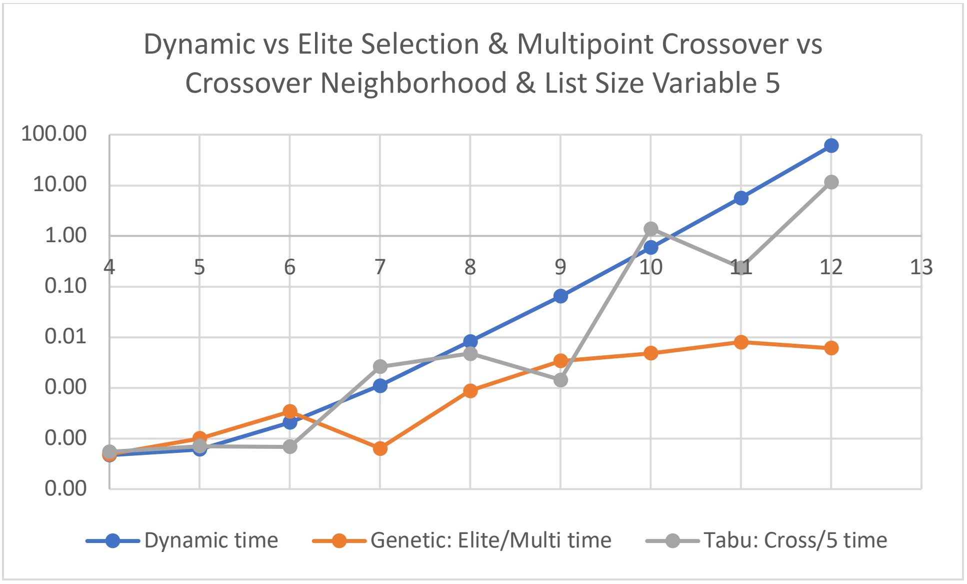
Tabu Search



The crossover neighborhood definition, based on the overall data, found the best solution in the shortest amount of time compared to the other swap neighborhood definition. Also, when looking at how the Tabu list size variable impacts timing, the smaller that Tabu list size variable is, then the quicker the best solution is found in both the swap and crossover neighborhood definition. This outcome makes sense because the best solution is being crossed with a random solution, so a subset of the best solution is being added to a child and then added to the neighborhood along with another random permutation solution, so the chances that an optimal solution is generated is higher because all the best solutions are being crossed.

Since Tabu Search is also a stochastic algorithm, meaning the input parameters (how a neighborhood is defined and the Tabu list size) used to find the most optimized path use randomness to find that optimized path. Because of this randomness, Tabu Search also does not have a definitive asymptotic complexity. However, the algorithm's complexity is relative to the number of nodes, the stop case to make the search stop, and the complexity of the neighborhood definitions.

Comparison



When comparing the best, on average, of the three different algorithms, Dynamic Programming, Genetic Algorithm, and Tabu Search, the genetic algorithm outperforms the other two once the number of nodes reaches 7 and Tabu search outperforms the dynamic programming approach.

<u>Design Documentation</u>

Design Patterns Utilized

- Strategy Pattern
    - This design pattern will easily allow for other algorithm types to be derived, allowing for future expandability. This also allows for code reusability. In my program, AbsAlgorithm is the parent class and TSP is the derived class. In the future, if another problem needs to be solved like TSP, it can become a derived class of AbsAlgorithm. TSP uses the two algorithm classes Genetic and Tabu as two different algorithms to solve the TSP problem.
- Factory Pattern
    - This pattern generates an AbsAlgorithm object pointer that creates the TSP object. In the future, if another problem were derived from AbsAlgorithm, there would be a way to create an object pointer of that type in the create function. This helps with future expandability.
- Chain of Responsibility
    - This pattern is utilized in the TSP, TSPGenetic, and TSPTabu classes. Chain of responsibility makes it easier to call multiple functions on an object in one line instead of having several lines with one function call. I used this in my program because I needed to call several functions to set variables and finally to run the

algorithm and wanted to reduce the number of lines with one object function call when it was not necessary.

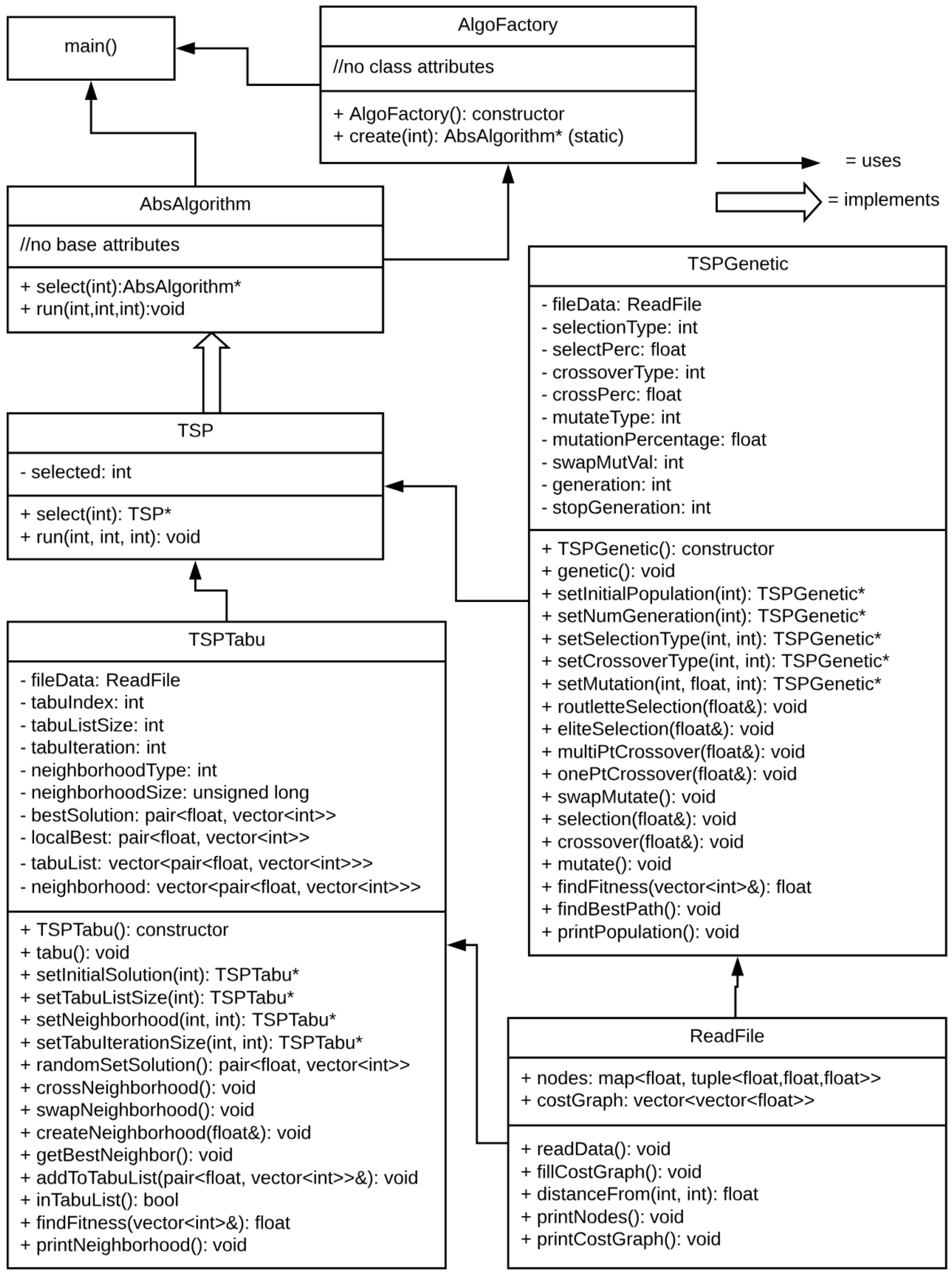
In main, I use the factory design pattern by calling the create function and setting the returning value to an AbsAlgorithm object pointer. The create function takes a problem option enum which determines what type of object is returned by the create function.

Once the AbsAlgorithm object pointer is initialized to a problem option type, the algorithm to be used on the problem is set in the select function and then the run function accepts any arguments that are needed for running the algorithm.

- For Genetic Algorithm, those accepted arguments include the selection type and the crossover type that needs to be used in the argument.
    - Selection options: elite(best) selection and roulette(probability) selection.
    - Crossover options: multi point crossover (a random index is found for the start index of that subset that is going to be passed from parent to child) and one-point crossover (a subset of starting index at the beginning is passed from parent to child).
- For Tabu Search, those accepted arguments include how a neighborhood is defined and the size of the Tabu list.
    - Tabu list size options: any variable integer.
    - Neighborhood definition options: Swap (where the initial solution has random indexes values swapped) and cross (where the initial solution is crossover with another randomly generated solution and its children are added to the neighborhood).

There were small changes made to the structure form Lab3. The design patterns where kept the same, just reimplemented for the new algorithms. However, I changed the use of the readFile class. Before, the readFile method was would return a value that would initialize the cost graph and map of nodes in the algorithm classes but in this lab, the object's map of nodes and cost graph adjacency matrix were made public. Once of readFile object is initialized in the algorithm class constructor, the public members are accessed through the object instead of being stored in the actual algorithm class.

UML Diagram


main()
AlgoFactory
//no class attributes
+ AlgoFactory(): constructor
+ create(int): AbsAlgorithm* (static)
= uses
= implements
AbsAlgorithm
//no base attributes
+ select(int):AbsAlgorithm*
+ run(int,int,int):void
TSP
- selected: int
+ select(int): TSP*
+ run(int, int, int): void
TSPGenetic
- fileData: ReadFile
- selectionType: int
- selectPerc: float
- crossoverType: int
- crossPerc: float
- mutateType: int
- mutationPercentage: float
- swapMutVal: int
- generation: int
- stopGeneration: int
+ TSPGenetic(): constructor
+ genetic(): void
+ setInitialPopulation(int): TSPGenetic*
+ setNumGeneration(int): TSPGenetic*
+ setSelectionType(int, int): TSPGenetic*
+ setCrossoverType(int, int): TSPGenetic*
+ setMutation(int, float, int): TSPGenetic*
+ routletteSelection(float&): void
+ eliteSelection(float&): void
+ multiPtCrossover(float&): void
+ onePtCrossover(float&): void
+ swapMutate(): void
+ selection(float&): void
+ crossover(float&): void
+ mutate(): void
+ findFitness(vector<int>&): float
+ findBestPath(): void
+ printPopulation(): void
TSPTabu
- fileData: ReadFile
- tabuIndex: int
- tabuListSize: int
- tabuIteration: int
- neighborhoodType: int
- neighborhoodSize: unsigned long
- bestSolution: pair<float, vector<int>>
- localBest: pair<float, vector<int>>
- tabuList: vector<pair<float, vector<int>>>
- neighborhood: vector<pair<float, vector<int>>>
+ TSPTabu(): constructor
+ tabu(): void
+ setInitialSolution(int): TSPTabu*
+ setTabuListSize(int): TSPTabu*
+ setNeighborhood(int, int): TSPTabu*
+ setTabuIterationSize(int, int): TSPTabu*
+ randomSetSolution(): pair<float, vector<int>>
+ crossNeighborhood(): void
+ swapNeighborhood(): void
+ createNeighborhood(float&): void
+ getBestNeighbor(): void
+ addToTabuList(pair<float, vector<int>>&): void
+ inTabuList(): bool
+ findFitness(vector<int>&): float
+ printNeighborhood(): void
ReadFile
+ nodes: map<float, tuple<float,float,float>>
+ costGraph: vector<vector<float>>
+ readData(): void
+ fillCostGraph(): void
+ distanceFrom(int, int): float
+ printNodes(): void
+ printCostGraph(): void

Outcomes

Genetic Algorithm:

- Elite Selection: takes the top best percent, a value determined by user, and moves them adds them into the next generation. In the graph shown above, I determined that 50% of the original population will move onto the next. Elitism allows for the best chromosomes to be passed along to the next generation and can lead to determining the best or optimized chromosome quicker because the best chromosomes found so far move along with each generation.
- Roulette Selection: each chromosome in the population has a P probability to move onto the next generation. In the results shown above, I determined 50% of the original population would move onto the next generation. The chromosomes that moved onto the next were randomly selected until the size of the generation was half the of the number of nodes times 10.
- Multipoint Crossover: a subset starting at a random index is moved over to the child and then the remaining nodes not included in the subset from the first parent are added to the child in the order the nodes are found in the second paren. The number of crossed over chromosomes was also half the size if the number of nodes times 10. Once the selection and the crossover methods are completed, the new population is the same size as the old.
    - Crossover Example using 8 nodes
    - Subset size = 2, Start index = 3, end index = 4

Parent 1:	1 3 4 7 8 6 5 2 1
Parent 2:	1 2 3 4 5 6 7 8 1
Child 1:	1 2 3 7 8 4 5 6 1
Child 2:	1 3 7 4 5 8 6 2 1

- One-Point Crossover: a subset starting at the beginning of the chromosome to the middle of the chromosome is moved over to the child and then the remaining nodes not included in the subset from the first parent are added to the child in the order found in the second parent. Once the selection and the crossover methods are completed, the new population is the same size as the old. Now the new population just needs to be mutated.
    - Crossover Example using 8 nodes
    - Subset size = 2, Start index = 1, middle index = 3

Parent 1:	1 3 4 7 8 6 5 2 1
Parent 2:	1 2 3 4 5 6 7 8 1
Child 1:	1 3 4 7 2 5 6 8 1
Child 2:	1 2 3 4 7 8 6 5 1

- Mutation: Using the value that's 5% of the population size, that number of chromosomes in the new population will be randomly selected and two indexes in the chromosome are randomly swapped.

Elitist selection allows for the best chromosomes to move on while roulette randomly selects chromosomes to move onto the next population. Elitist selection should almost always outperform roulette selection because the chances of roulette choosing the best or some of the best chromosome to move to the next generation are very small, however, if the initial population has better fitness scores than the initial population when using roulette, it is possible for roulette selection to outperform.

Comparing one-point crossover to multipoint crossover, depending on which selection type is used, multipoint crossover should outperform the other. This is because unless the first half of the chromosomes being crossed over using one point have the smallest cost, there is a large chance that the best chromosome will not be found, but this also depends on the randomly generated initial population and the type of selection that was used in previous generation. Elite would work best with this most times because the best chromosomes have been moved on from generation to generation and there is a large chance that the fitness of the first half of each chromosome is optimized.

Both multipoint crossover and one-point crossover are implemented using ordered crossover, meaning that once the subsets from the parents have been copied over to the children, the opposite parent from which the subset was not copied from is used to copy the nodes missing from the child in the order that they are found in the opposite parent. By implementing crossover in this way, it prevents the crossed over chromosomes from having duplicate nodes and making the newly generated children invalid and unable to be used and passed onto the next generation.

Tabu Search:

- Initial solution: a randomly generated permutation.
- Crossover Neighborhood Definition: the neighborhood is created starting with the randomly generated initial solution and performing a multipoint crossover (defined and given an example in the genetic algorithm selection type explanation) with a another randomly generated solution until the neighborhood is filled. Then finds the best solution in that neighborhood, compares to the global best solution, and adds to the Tabu list if the solution is not already in the list.
- Swap Neighborhood Definition: the neighborhood is generated starting by randomly swapping indexes in the initial solution. Starting from the beginning of the solution, randomly generate an index between the start index and the end of the solution, swap those values, increment the start index, and repeat until the neighborhood size is met. By generating neighborhoods using this implementation, duplicates are avoided and there is a good chance that the best solution is generated because the solution being swapped is the best solution found so far.
- Tabu List Size: Two different variables were used to generate different sizes of the Tabu list. The variable size multipliers used to collect the data above were 5 and 20, meaning that the Tabu list size changes with the number of nodes it needs to search. If the number of nodes that needs to be searched is 8, then the Tabu list size when the multiplier is 5 is 40 and when the multiplier is 20, the Tabu list size is 160.

Crossover neighborhood definition should outperform the swap neighborhood definition because there is so much randomness in the swap definition. The best solutions found will be crossed over will a random solution but there is a subset of the best solution that gets added to the neighborhood and if that subset’s fitness is optimized, then there is a higher probability that a better solution will be generated.