Charles Nguyen 101184017

COMP1406Z – Intro to Computer Science II
Project Report
Helping Santa Claus Delivering Presents in Time!

I. Project Problem

Christmas is coming! That means Santa Claus and his Elves must start the grind by planning and preparing presents for the kids around the world. You are one of his Elves that needs to bring him the optimal route map, i.e. a single closed route that goes through every cities and covers the least amount of distance, so Santa Claus can deliver the presents in time. Given a text file data containing the cities' coordinate gathered by other Elves already, you task is to compute this route and display it to Santa.

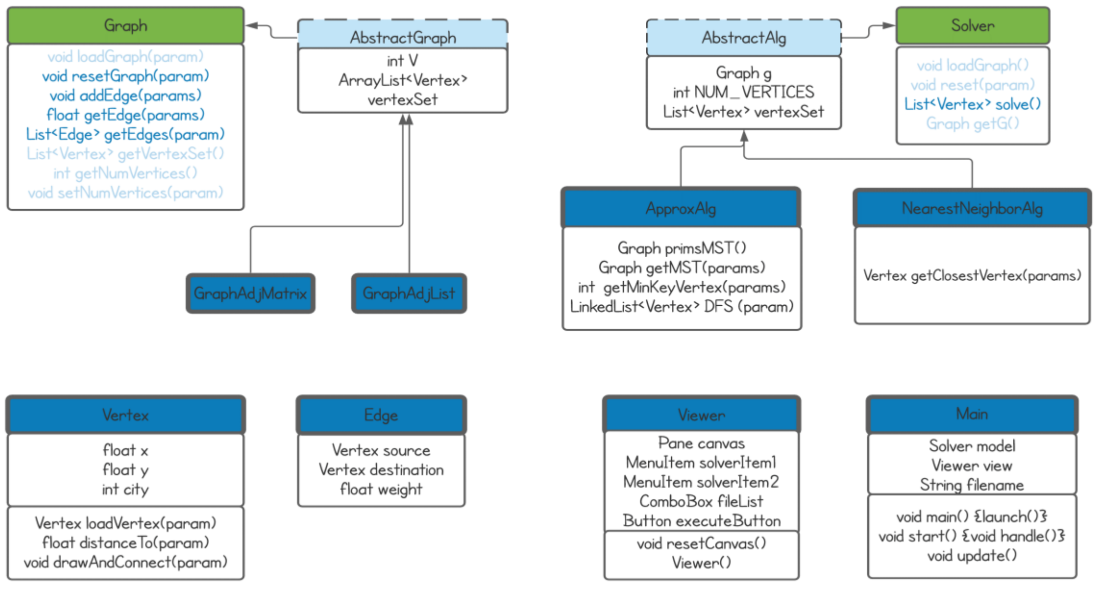
II. Project Analysis

1. Overview

The project incorporates the MVC paradigm where the model is the algorithm doing the heaviest work of loading, processing the coordinate data from text file, and solving the problem. The view is the front-end (JavaFX GUI) where the user interacts, i.e. chooses the algorithm as the model and the text file to use, and the view finally displays the route map. The controller is the main application that connects the view and model by instantiating view & model objects as well as handling the events and errors.

The given problem is an instance of the Traveling Salesman Problem and can be translated into a graph problem where given a complete weighted undirected graph G containing vertices (the city's coordinate from text file) and edges (Euclidean distance between city), what is the minimum Hamiltonian path, i.e. a path that covers every vertex with the minimum weight total and returns back to the origin point.

2. Project Design



i. OOP Design

- The above diagram provides a detailed overview of the entire project structure and is fairly intuitive to follow. Interface, abstract and concrete classes are all used, and the methods & attributes are colored to indicate which class will implement, override or inherit them. The diagram only shows the information I think are the most relevant to consider when understanding the project structure as well as the algorithms at a high level.
- Beside the project problem, an important implicit goal here is the project being modular by using MVC paradigm and all OOP principles well, and much emphasis is on the extensibility/scalability of the project where data, algorithms and new graph data structure can be easily added.

- Abstract and Encapsulation:

In the entire project, most class attributes are declared private along with get/set methods provided to ensure privacy of each class and thus the entire application together is safe and encapsulated when instances of complex different classes interact with each other. Additionally, some attributes are declared as protected but only where there's inheritance occurs, in this case when the abstract classes are subclassed and only the subclasses specifically chosen to extend from these abstract classes can inherit and access these information, which is still considered a safe programming practice and secured application.

Abstract classes are used, in terms of encapsulation, because they have the common attributes and behaviours of all of its concrete important subclasses that user will use and therefore it critical that these common parts are encapsulated (by not being able to instantiate the abstract class and work with it) to ensure users don't cause error by accessing it leading to the entire subclasses malfunction.

In terms of abstraction, these abstract classes process a lot of details that are better abstracted so the general users don't have to worry about. For example, indexing is a problem because the given text files has the city's name identified by number index but it start with index 1 whereas the programming requires working with 0 based indexing. We use the city name to index into our Graph representation and different arrays, and this is clearly a trouble when new added concrete classes will have to manage this indexing problem too. Luckily, this kind of low-level, "ugly" task have been handled by the abstract class.

Additionally, using abstract classes is just pure logical because, for example, Graph alone doesn't mean much to users because it's not specific enough to implement anything on its own. A more specified Graph is what Graph representation/data structure used will be much more helpful to user. Therefore we don't want anybody to instantiate this class and use it just like any other specific objects, and the reason we still have it is for inheritance purposes.

In our concrete algorithm classes, the algorithms can be very complex with many helper functions, but in the end we're only interested in the main solve() method when calling theses algorithm instances. All other helper methods are declared as private to ensure encapsulation as we're dealing with a specific algorithm and these helper methods only apply to this algorithm and should only be accessible within the scope of this class.

In general, the entire project is written in an OOP style therefore there's a lot of abstraction as the project develops and becoming more high-level. For example, the Vertex classes handle some details such as loading information from the text file, compute the edge weight by calculating Euclidean distance and even draw the vertex itself on JavaFX Pane. As the project scales up and we now have to deal with tasks from loading and building a complete Graph, adding an edge to a graph up to the front-end task of creating GUI, simply call the Vertex methods and the tasks are handled. Likewise, the algorithm (instance of Solver) is required to deal with Graph by loading it and resetting it, but logically the Graph itself should do this task. By knowing which task should belong to which class and being able to make modular code, we're extremely benefit by working in an abstracted way when the project is developed at a higher-level and save us so much time. Likewise, when others looking at the code, they wouldn't have to worry or spend much time about this low-level details.

- Inheritance:

The specific Graph representations has a lot of unique implementation because of its unique data structure, but in the end they still have a lot of things in common. For example, once the basic operations of a Graph are defined, building or loading a Graph is the same for every specific Graph representations. Also, since every thing is encapsulated, all concrete classes will have the basic set/get methods and for specific different Graph representation or algorithm, they are exactly the same. Therefore having a super class like abstract classes we mentioned earlier has the inheritance purpose by keeping track of the common attributes and behaviours of the more specific subclasses. As the result, the amount of code is significantly reduced which increases readability, and more importantly the subclass don't have to keep track of these common details and can rather focus on it own unique implementation only. Both specific Graph representations and algorithms don't have many attributes or methods in their own class, their constructor to initialize is very simple, even though they have to rigorously follow the Interface, thanks to the super classes have handled the common tasks already such as loading or resetting graph. From the Interface, in the end each concrete subclass only have to implement half of those specified methods, and there's no duplication since every class has a unique role and this unique code.

This help in reducing amount of time extending the project further, since when we follow this project structure and adding new things in such as algorithms and graph data structure, we will only need to focus on the specific implementation and not need to keep track of general things as you will inherit these details from the abstract super class above already.

- Polymorphism:

The Interfaces specified are fairly basic and intuitive. For every graph data structure, they should at least follow these basic operations of a Graph ADT; namely adding edge, getting all the neighbor edge, getting edge weight, loading/resetting the entire graph, etc. … For algorithms, we specified earlier that they both should responsible for choosing a graph they want to use and process it, a long with solving the problem.

The Interfaces are to ensure scalability and maintainability of the project when the application grows much more complex or errors occur, because we know that new objects added such as new graph data structures or algorithms must follow the rules we specified by implementing these Interface at least.

By having objects following the Interface, we have now written a polymorphic code had the opportunity to make use of Polymorphism. This is a crucial part of the project since not only it helps extend the application, it also deals with memory optimization. For example, we have the 2 commonly use graph data structure: adjacency matrix and adjacency list, each with its own advantage and disadvantage and they both implement Graph Interface. Adjacency matrix is simply a matrix implemented as 2D array, each entry storing the weight of the corresponding vertices at the indexing. The memory it takes to store this is $O(E = V^2)$, where V is the number of vertices. The only reason when we use this data structure is when we have a very densely connected graph, i.e. when having a complete graph in our case. If we have a sparsely connected graph, much of the entry in the graph will be left empty since there're not many edges and it'd be a waste. In this case, we need to switch to adjacency list implemented using Hash Map with each key is the vertex/city and its value is a Linked List storing all the neighbour vertices. This way, the graph data structure dynamically expand as we need. Nevertheless, we're treating these 2 graph representation objects the same (by making them follow an Interface), as we'll be able to switch these 2 graphs easily depending on circumstances, since they both have the same methods.

Polymorphism is used a lot through the project, for example when the algorithms work with graph. It has the freedom to choose which graph data structure to use, but after initializes it, it will be treated as a general Graph (as shown throughout both algorithms when the graph data structure is passed through the constructor to when different helper methods within an algorithm easily switched between 2 different graph data structures). For instance, in the 2-Approximation algorithm, we initially chose to work with adjacency matrix since we had to first build a complete graph. After that, we're able to reduce it to a tree. Tree is basically a subgraph, in graph theory it is connected acyclic graph, meaning in general a tree is still a graph. But a tree is much sparser since it's total number of edges is now $O(E) = O(V)$. We now have to switch back to adjacency list representation, and fortunately, our algorithm (or any algorithm in the application) deals with graph as a general Graph rather than only a specific graph data structure, so we can easily switch without compromising the entire algorithm.

Likewise, for algorithms (instances of Solver), we want to further expand the project by developing many more algorithms and we're able treat them the same by making the algorithms following the Solver Interface. It makes sense for every algorithm to do this because in the end we're only interest in calling the algorithm's solve() method, and by treating them all the same, i.e. an instance of Solver, we're able to group them together and easily switch around to see their performance on the same dataset.

The project makes great use of Polymorphism that if you switch the graph data structure in each algorithm, it would still perform well.

ii. Algorithm Design

- 2 – Approximation Algorithm:

Designing this algorithm is fairly intuitive. We first approach the TSP problem by noticing that we need to connect all the vertices somehow with minimum weight (it doesn't have to be a closed tour yet). This is the minimum spanning tree problem which can be solved in polynomial time. MST is a special type of graph, and it is a tree that connects all the vertices (spanning property) with least amount of edge weight (minimum property) from a graph.

The algorithm used to compute MST is Prim's algorithm which is a greedy algorithm. It maintain 2 sets, the 1st set store the vertices already inside the MST and the 2nd set is the remaining vertices we'll continue to choose from the original graph. Initially the 1st set has only 1 element which is some random vertex chose as the root. We look at its neighbors only and see which one is the closest (the neighbour vertex connects to it with least weight edge). We continue to do so with every other vertices until the 2nd set is empty. While doing so, we also have to continuously keep track of and update the parent vertex which is closest vertex it just comes from. As soon as we find a new vertex that leads to a lesser weight edge then we have to update it. This way Prim's algorithm works and runs in $O(V^2)$.

We now need a way to construct a path out of this MST and a simple way is to use basic graph traversal algorithm. BFS is not particularly helpful since it visits all the neighbor vertices at once before moving next so its ambiguous which vertex to choose. This leaves us with DFS algorithm to use and in this implementation we use Stack ADT instead of the more common recursion algorithm. This is because there're thousands of vertices to visit and hence that many recursive call which can lead to stack overflow. Stack ADT's mechanism of last-in first-out work just like a recursive calls to function in the stack memory. The path that DFS goes through we'll only add the vertices we haven't seen before to the result list. Finally as DFS finishes traversing we return the result list which is the tour we're looking to solve. The graph traversal DFS algorithm runs in $O(V+E) = O(V)$.

Prove the result tour is no more than twice the optimal tour: Denote OPT as the cost (distance) of optimal tour and MST is the cost of the minimum spanning tree. It's intuitive that $MST \leq OPT$. The minimum spanning tree is only a tree and doesn't have to rigorously go around while covering all the vertices with minimum weight. When we do a DFS on this MST, it hits every edge of the tree twice (we first explore that edge, and later backtrack through that edge as well). Therefore the cost is $\leq 2 \times MST \leq 2 \times OPT \quad \blacksquare$.

The heaviest work of the algorithm is when we compute the MST and the total runtime of this entire algorithm is $O(V^2)$.

- Nearest-Neighbour Algorithm:

This is also a sort of greedy algorithm that is very simple but performs worse than the previous algorithm in terms of finding optimal tour. For every vertex in the complete graph, look at all of its neighbour and find the closest vertex, add it to the tour and make sure there's no vertex duplication. This algorithm is heuristic and does not guarantee anything, but it runs in polynomial time $O(V^2)$.

III. Conclusion

- In terms of OOP design, the project has made great use of all the principles: abstraction and encapsulation with the appropriate access modifiers private and protected on methods and attributes. Inheritance is also a part of this project and polymorphism is the most important aspect. Along with MVC paradigm, I think the project has achieved its goal of making the project maintainable and extensible in terms of adding new algorithms solvers or new graph data structure and data file.

- In terms of algorithm the runtime is reasonable given that the TSP problem is NP-hard and the complexity of the algorithm itself in this project has been well demonstrated. Currently the project is not scalable given the resource constraint which is memory space. Building up a complete graph is very costly and in fact, the loadGraph() is the most expensive algorithm in this entire application.
- Possible extension:
    - The obvious one is to implement more algorithms that can solve the TSP problem even better. There're a lot of algorithm out there and the 2 – Approximation algorithm can be reduced to 3/2 – Approximation
    - I need to do more research on optimizing loadGraph() algorithm, maybe I don't need to build a complete graph but just enough of it, that way the project can scale up a bit more with larger text file. Additionally, there can be other graph data structures that are more efficient at this and adding it to the application will be just as easy as adding new algorithm thanks to Polymorphism