EE599 Final Project
Trojan Map

A small map application based on the C++ programming

Tian Xia, txia0604@usc.edu

11/22/2020

1. Abstract

This project focuses on using data structures and graph search algorithm to build a mapping application. In this application, it has four main functions: searching location included in the map, searching the position of a given location, calculating the shortest path between two locations and calculating Traveling Salesman Problem for randomly selected locations. In this project, I will use different methods and algorithms to construct these functions, and make them work well in the program. I will also research their time complexity and their behavior in the test case.

2. Basic Functions of The Trojan Map

This project is aimed to building a mapping application by using data structures and graph search algorithms. This project has four main functions related to this field:

Autocomplete: using insensitive partial name to search a list of location

Find the position: using sensitive name to search the position of the location

CalculateShortestPath: giving the shortest path and its length between two location

Traveling salesman problem: randomly select a number of locations and give you the shortest path to reach these locations and return to the start location

3. Small modification in the PrintMenu function

Because I have applied three methods at the TSP function, so I make a small change at the PrintMenu() function to let people be able to select a method to do the TSP function. After inputting the number of points, the programming will print following menu:

```
************************************************************
* 1. Brute Force
* 2. 2-opt
* 3. 3-opt
************************************************************
```

You can input the number of method to let the system choose the selected method to solve TSP. If your input is an invalid input(e.g. 4, "I want to use 3-opt method", ...). The system will remind you that it is an invalid input and use the default method(Brute Force) to solve TSP.

4. Functions in this project

4.1. Simple functions which return related parameter

In this project, it also has some simple functions which return related parameters based on the input location ID as following:

```
double GetLat(std::string id);

double GetLon(std::string id);

std::string GetName(std::string id);

std::vector<std::string> GetNeighborIDs(std::string id);
```

For these functions, I use the private variable, data, in this project. It is a map variable whose key is the id of locations and value is the node to the id. Thus, I can use this private variable to find the node by the inputted id and return the related parameter of this node.

4.2. Helper functions to calculate the distance between locations and the length of a path

Because of the last two main functions for the menu item, it is necessary to have two functions which can calculate the distance for different situations as following:

```
double CalculateDistance(const Node &a, const Node &b);

double CalculatePathLength(const std::vector<std::string> &path);
```

The first function is to calculate the distance between two locations, and the other one is to calculate the length of a vector of locations. Let's explain the first function at first.

In the function **CalculateDistance**(), I use the Haversine Formula to calculate the distance between two locations based on their latitudes and longitudes.

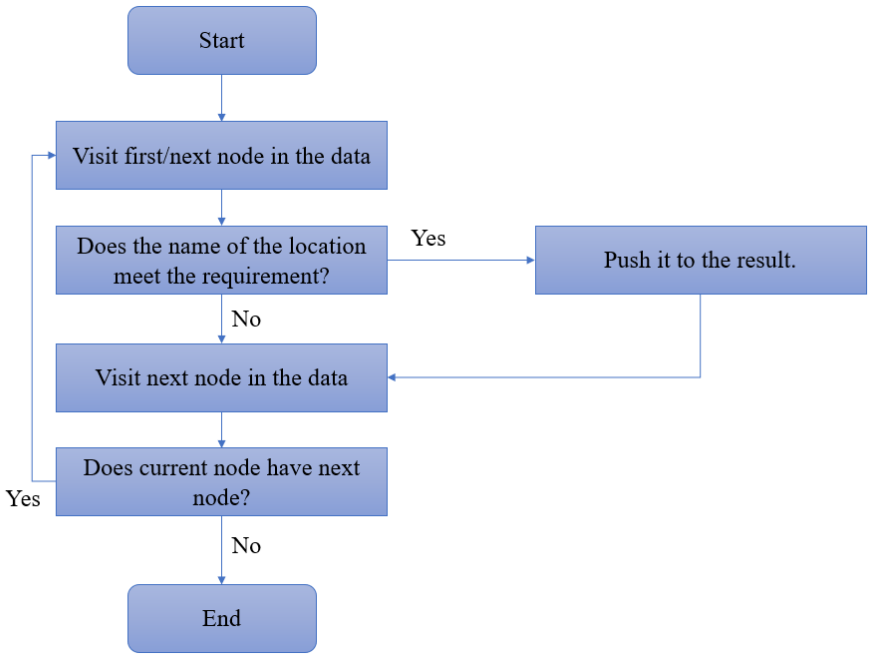
In the function **CalculatePathLength**(), I calculate the distances between two adjacent locations in the vector and accumulate them as the sum length of this path.

4.3. First function for the menu: Autocomplete

The first main function for the menu in this project is to search a place by a unsensitive partial name as prefix. Here is the header of this function:

```
std::vector<std::string> Autocomplete(std::string name);
```

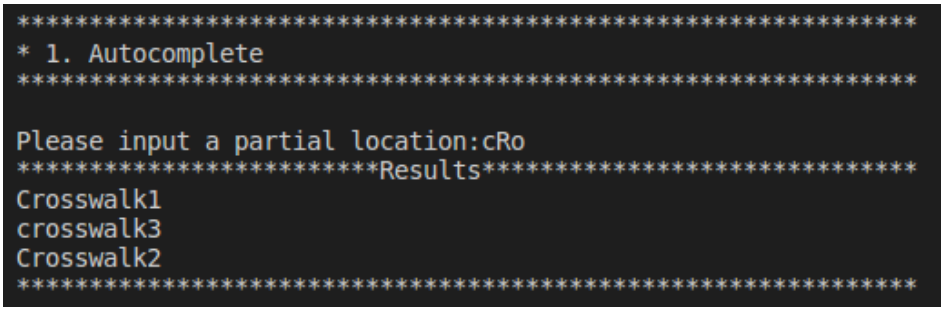
In this function, I use the function **equal**() from STL to find name in the map which match the requirement in a loop. If it doesn't exist the required name, it will return an empty result. Here is the code diagram of this function:

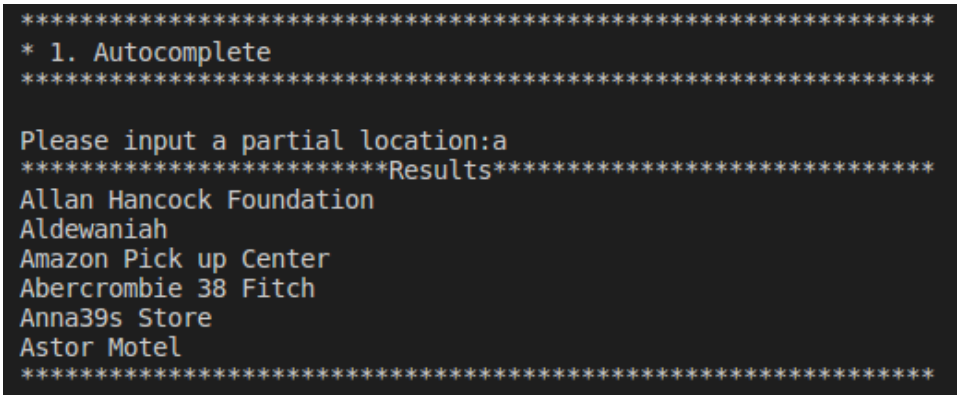


The time complexity of this function is O(n).

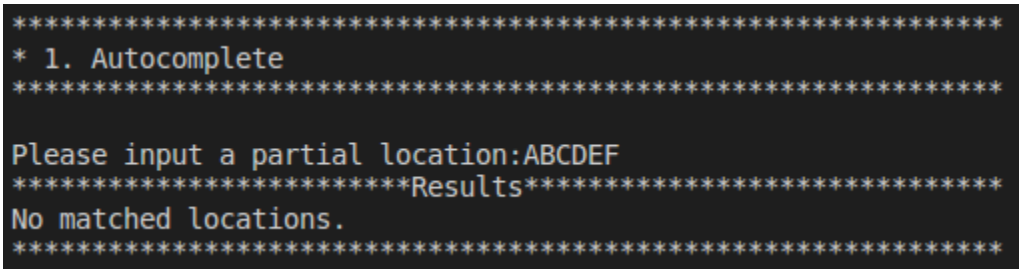
Here are some results of the live demo:

```
************************************************************
* 1. Autocomplete
************************************************************

Please input a partial location:cH
**************************Results***************************
ChickfilA
Chipotle Mexican Grill
************************************************************
```





When the project can't find any location which requires your input, its result will be:



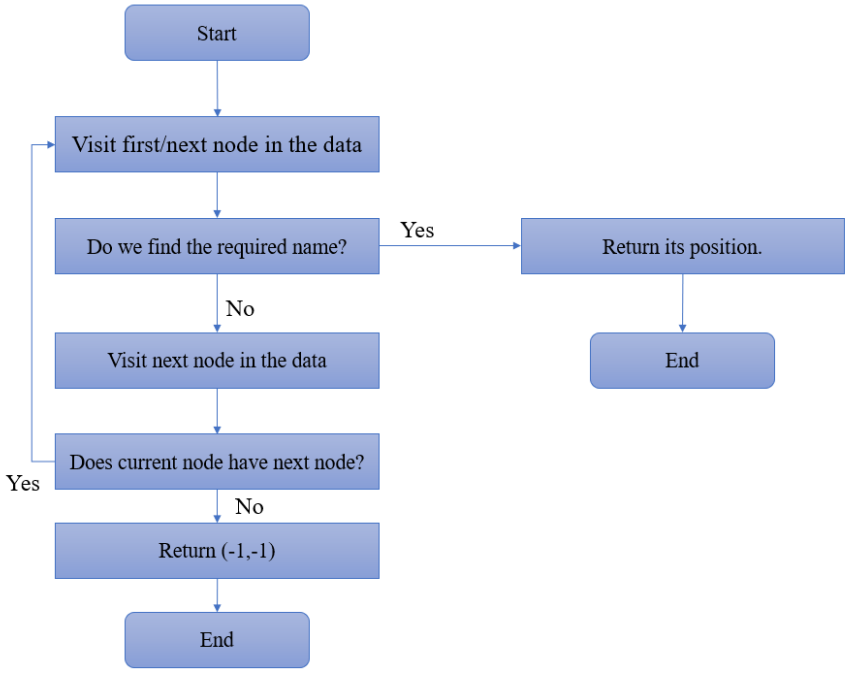
4.4. Find the position of a location

This function can return the position of a given location name or (-1,-1) if there isn't a location has the given name. Here is the header of this function:

```
std::pair<double, double> GetPosition(std::string name);
```

This function can easily achieve it by using a loop for data variable to find a location which meet the requirement. The time complexity of this function is O(n).

Here is the code diagram of this function:

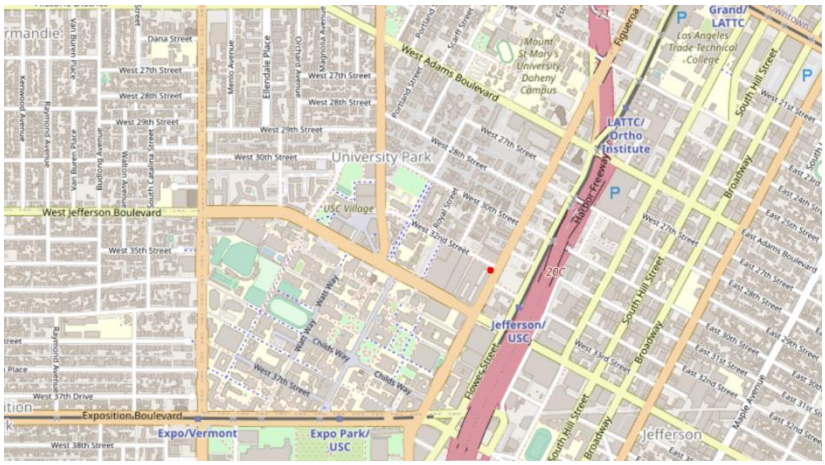


The time complexity of this function is O(n).

Here are some results of the live demo:

```
**********************************************************************
* 2. Find the position
**********************************************************************

Please input a location:CVS
**************************Results*********************************
Latitude: 34.0235 Longitude: -118.279
**********************************************************************
```

In the main programming, it will also plot the location on the map like following picture:



In following case, although Target is included in the map, the input should be case-sensitive. Therefore, when we input "target", it won't return the position of Target as following:

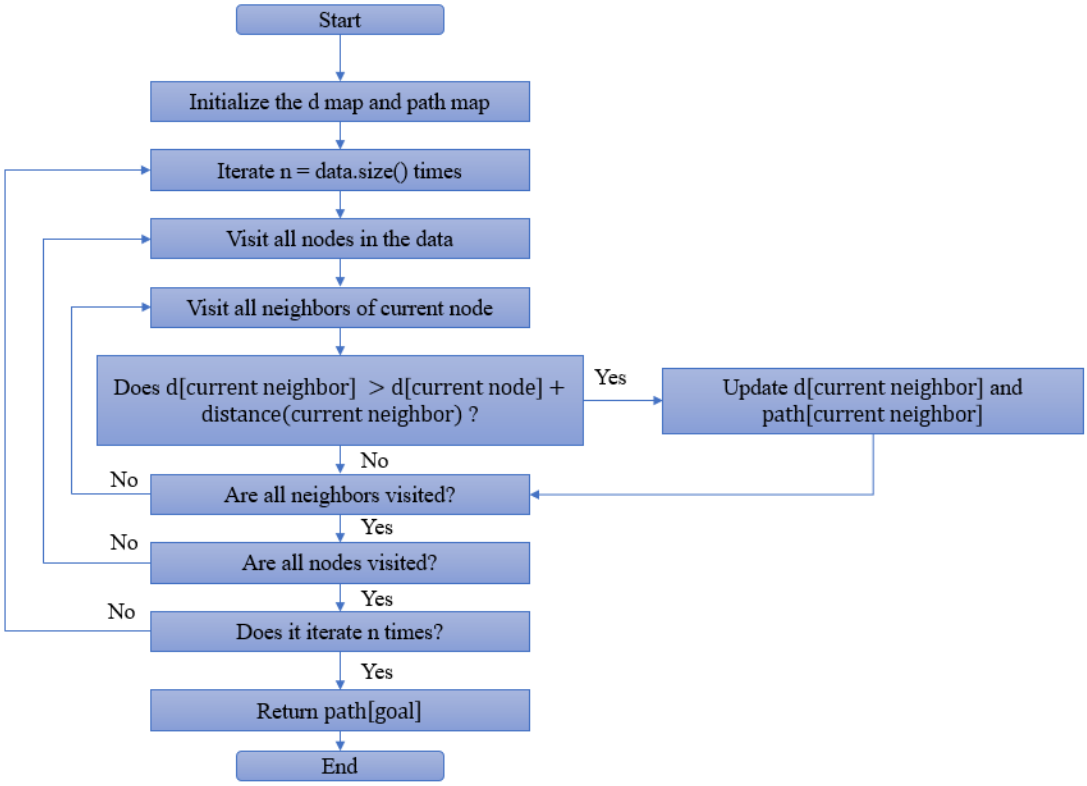
```
**********************************************************************
* 2. Find the position
**********************************************************************

Please input a location:target
************************Results*******************************
No matched locations.
**********************************************************************
```

4.5. Calculate Shortest Path

Here is the header file of the function:

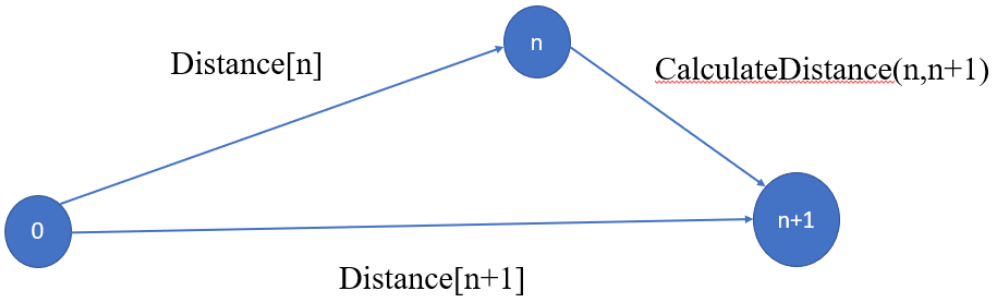
```
std::vector<std::string> CalculateShortestPath(std::string location1_name,
                                               std::string location2_name);
```

It will calculate and return the shortest path between location 1 and location 2. In this function, I use the Bellman-Ford algorithm, and here is the code diagram:



According to this diagram, the time complexity of this function is $O(n * m)$, which m is a specific number related to the neighbor number of each location.

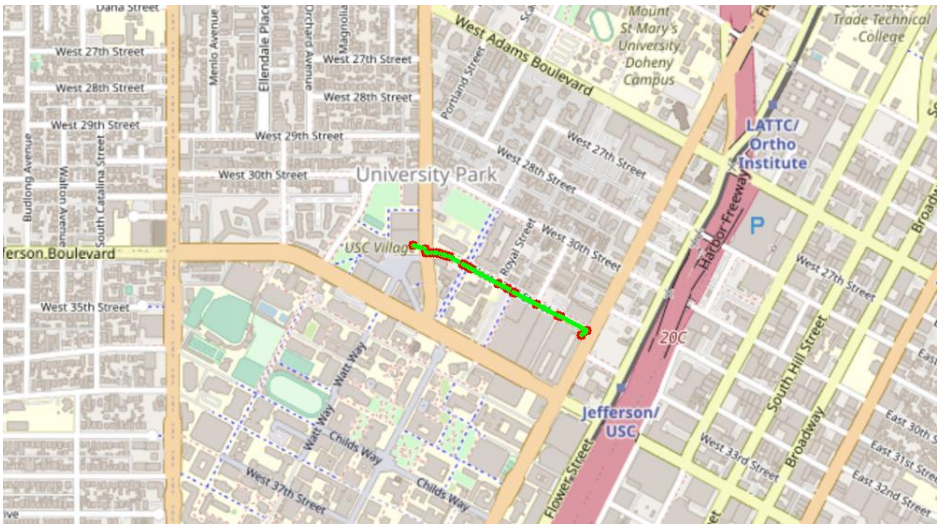
Here is a single process of the process of update the d map. if we start from node 0, and we found that the current distance[n+1] is greater than the sum of distance[n] and the distance between node n and node n+1, we will use the sum of distance[n] and the distance between node n and node n+1 to update the value of distance[n+1].



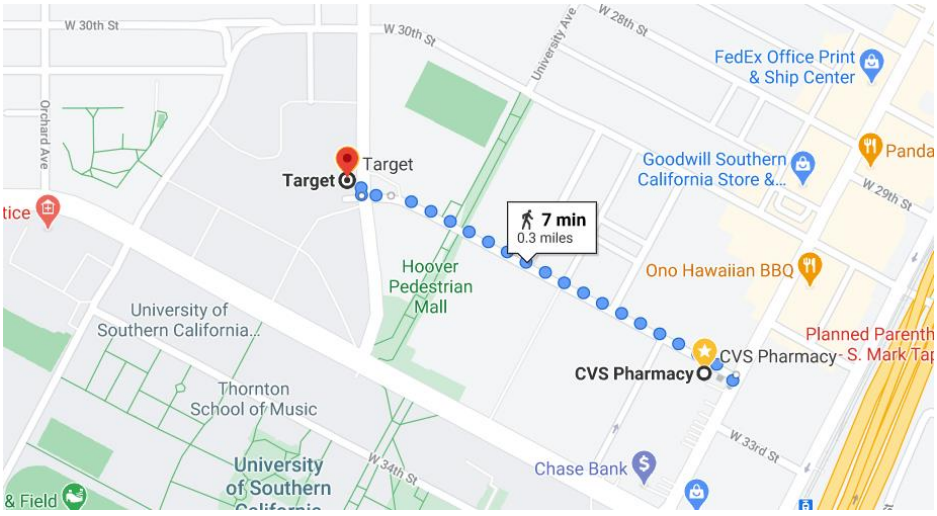
Here are some results of this function. Because it is also a case-sensitive function, you will get following result if you still using "target" as one of the inputs:

```
**********************************************************************
* 3. CalculateShortestPath
**********************************************************************

Please input the start location:target
Please input the destination:CVS
************************Results*******************************
No route from the start point to the destination.
**********************************************************************
```

Because the list of paths may be a bit harder to see in the programming, I use the plot of the path as the result. Here is the plot of the shortest path between Target and CVS:

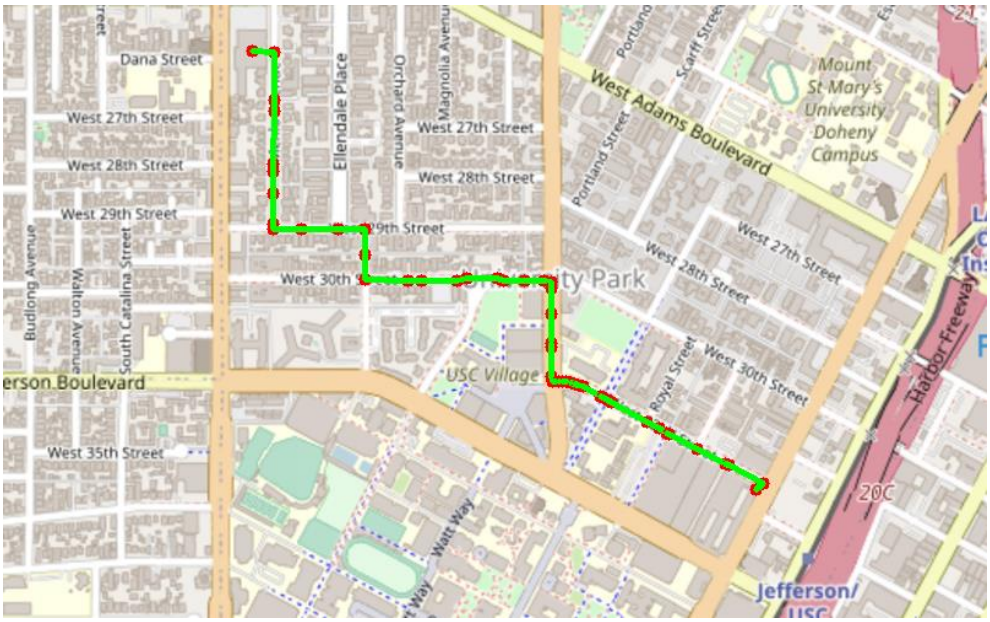


and we compare it with the recommend route on the Google map:

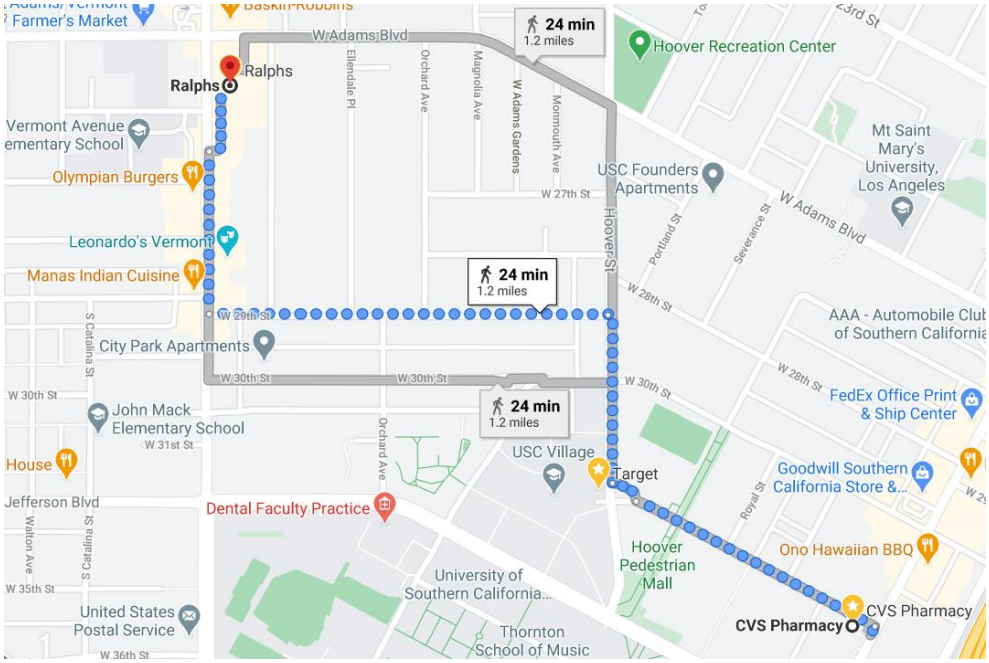


We can see that they have similar result.

Here is the plot of the shortest path between Ralphs and CVS:



and we compare it with the recommend routes on the Google map:



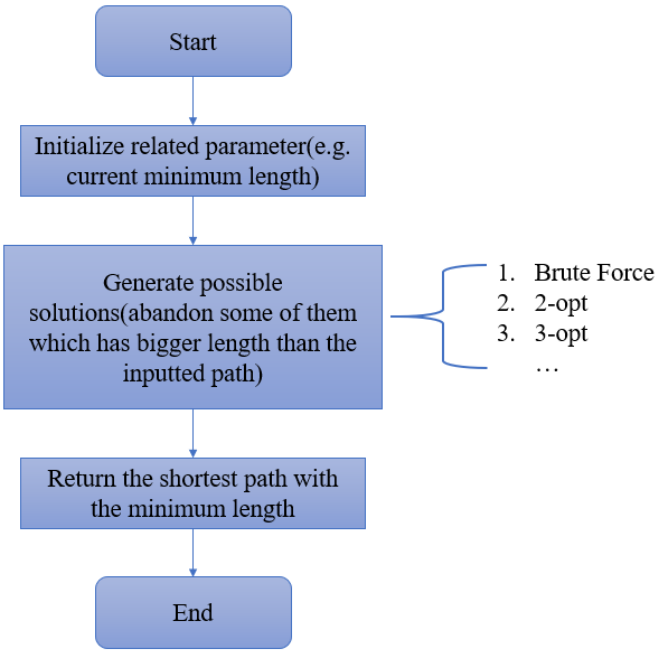
In this case, our result is a bit different from the Google map. I think that one of the reason is that the Google map may not only consider the distance between two location, it will also consider some more elements such as if there exists some roads which is blocked.

4.6. Traveling Trojan based on TSP

Here is the header of this function:

```
std::pair<double, std::vector<std::vector<std::string>>> TravellingTrojan(
      std::vector<std::string> &location_ids);

std::pair<double, std::vector<std::vector<std::string>>> TravellingTrojan_2opt(
      std::vector<std::string> &location_ids);

std::pair<double, std::vector<std::vector<std::string>>> TravellingTrojan_3opt(
      std::vector<std::string> &location_ids);
```

This function will randomly select locations based on your input and find the shortest route which can let you reach these locations and return to the start location. Here is the code diagram of this function:



According to this diagram, we can see that the main idea of this function is to generate new route and find if it has a shorter length. I also use three different algorithms in this part.

(1) Brute Force Method

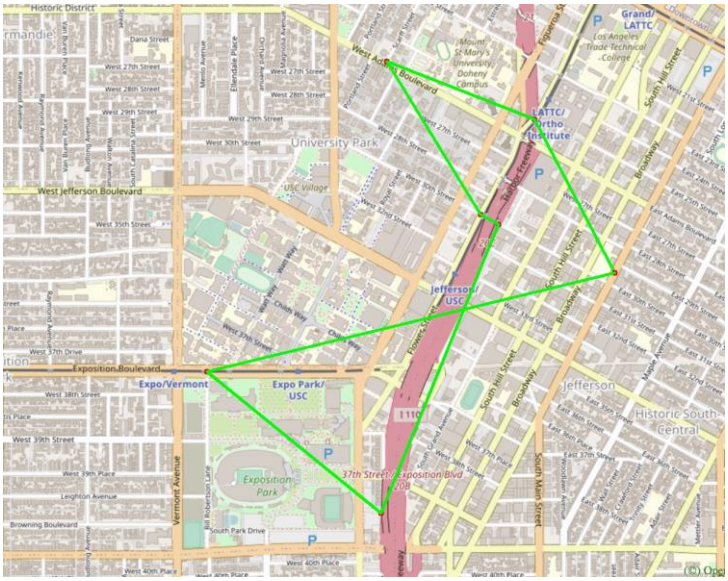
The first algorithm is Brute Force Method, it will generate all possible routes based on the inputted list, and then it will find the one whose length is shortest. I created a recursive helper function to generate all possible routes as the Brute Force Method. The time complexity of this method is O(n!) which is a bad run time in the computer science field. Thus, I apply a little optimization in my code, I use an If statement to let it abandon those routes which has longer distance than the inputted route.

Here is one result of the TSP function based on Brute Force Method:

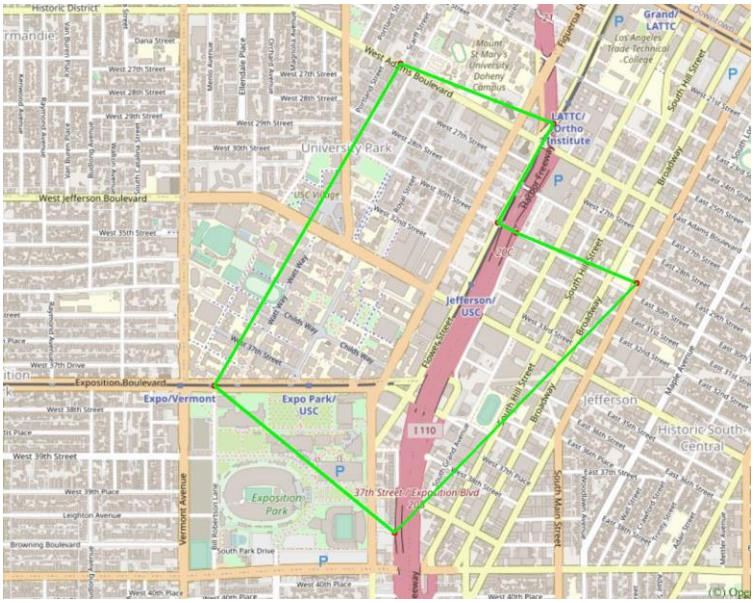
The list of selected points:

```
4012726926
214470964
5680945549
4400460721
6816180157
601390380
348122895
4012726926
************************************************************
The distance of the path is:3.70004
************************************************************
You could find your animation at src/lib/output.avi.
```

The animated plot(You can see the animated plot at the README file):

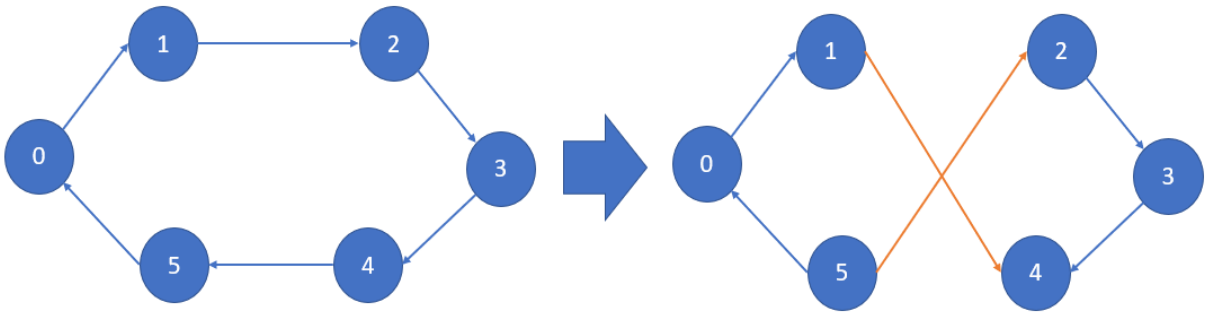


The final TSP route:

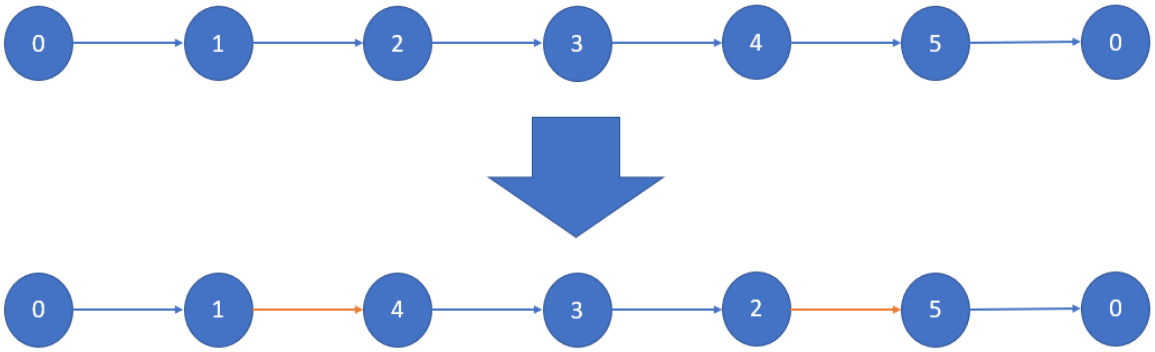


(2). 2-opt algorithm

I also apply the 2-opt method to solve the TSP. The main idea of 2-opt is to delete two edges from original route and reconnect them to generate a new route. Here is an example of a single process of 2-opt method.



The left part is the original route. We delete two edges, $1 \rightarrow 2$ and $4 \rightarrow 5$, and reconnect them to get the new route at the right part. Thus, the transformation from left route to right route is following:



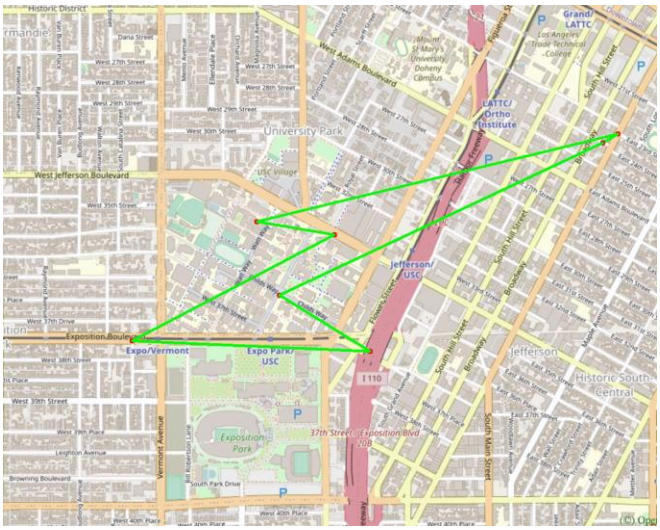
It is clear that the a single process of 2-opt swap can be think as reversing a part in the middle of the original route, so we can use 2-opt swap iteratively to the original route and recursively do the same process to the new routes to generate any possible solution to TSP. The final time complexity of 2-opt method is $O(n^2)$.

Here is one result of the TSP function based on 2-opt Method:

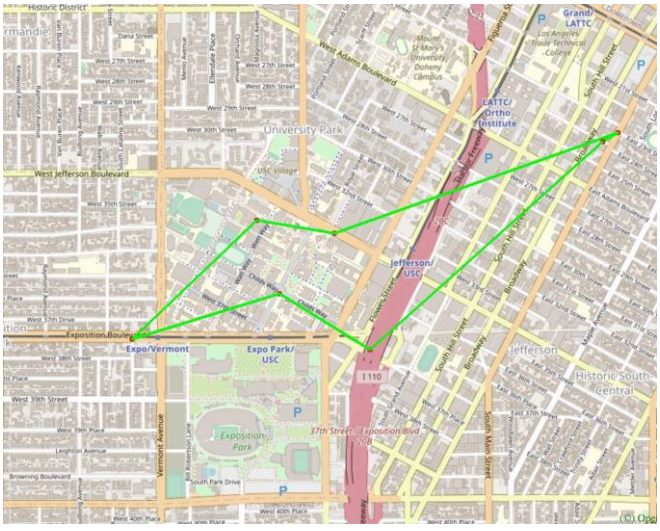
The list of selected points:

```
6816895139
1837212926
4399697646
368172334
123112727
2305853437
6815813015
6816895139
************************************************************
The distance of the path is:3.49839
************************************************************
You could find your animation at src/lib/output.avi.
```

The animated plot(You can see the animated plot at the README file):

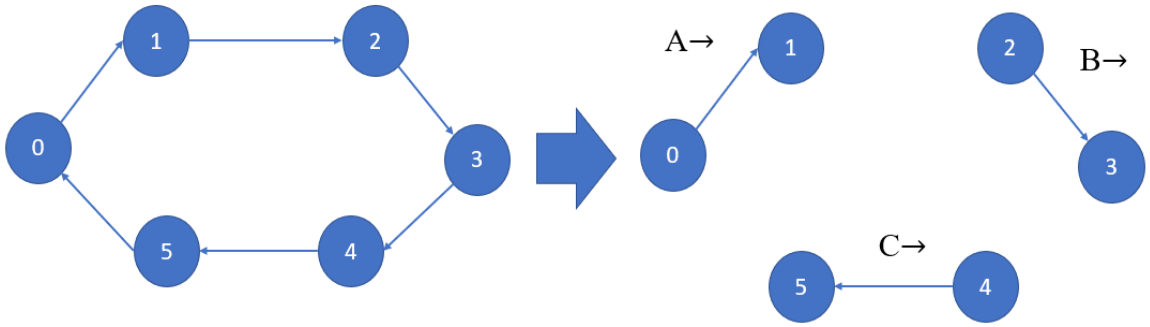


The final TSP route:



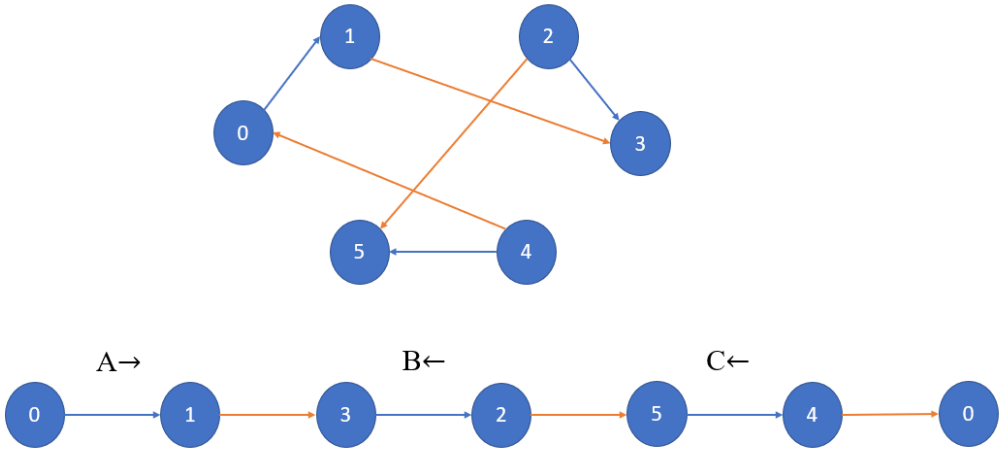
(3). 3-opt algorithm

3-opt method is similar with 2-opt method, it deletes three edges from original route and reconnect them in 7 different styles. Here is an example of a single process of 3-opt method.



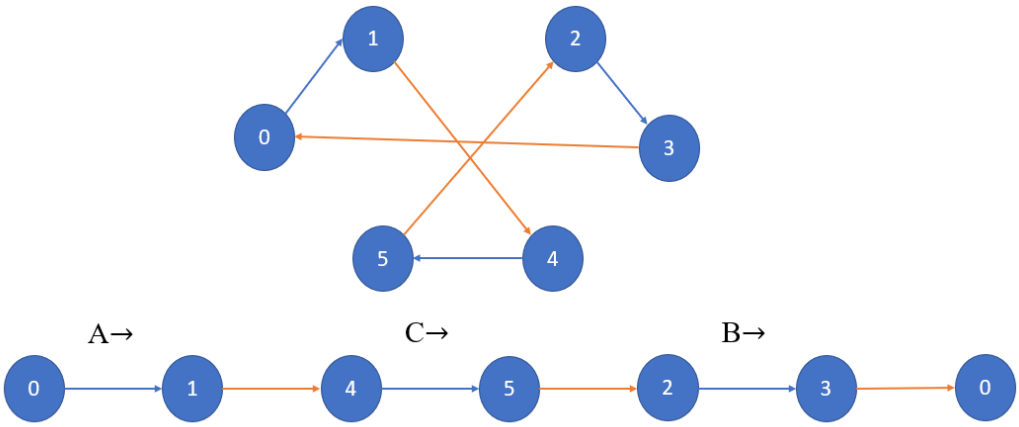
We use the same original route as the 2-opt case, and we delete the three edges, 1 → 2, 3 → 4, and 5 → 0. We will get three different sub routes, 0 → 1, 2 → 3 and 4 → 5. To make it clearer, we suppose that they are A, B and C.

Here is the first style to reconnect:

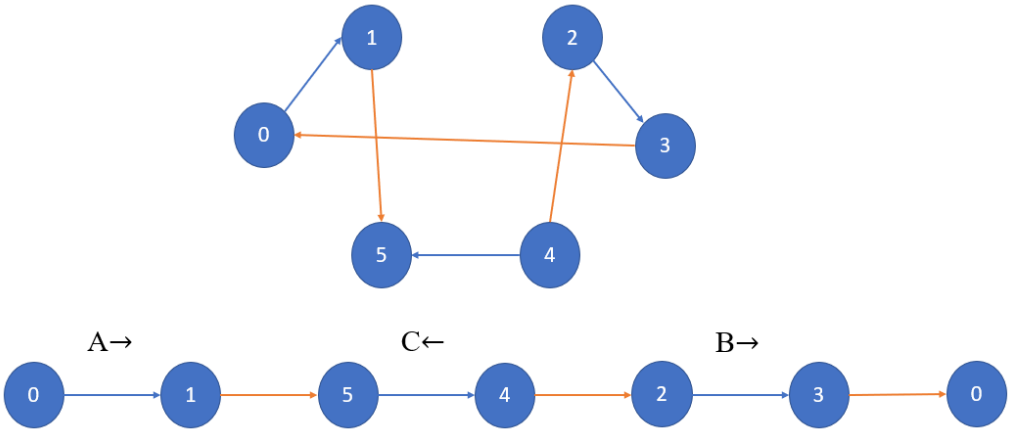


We reconnect the end of A to the end of B, the begin of B to the end of C, and the begin of C to begin of A. To simplify it, we can call the new route as [A→,B<-,C<-].

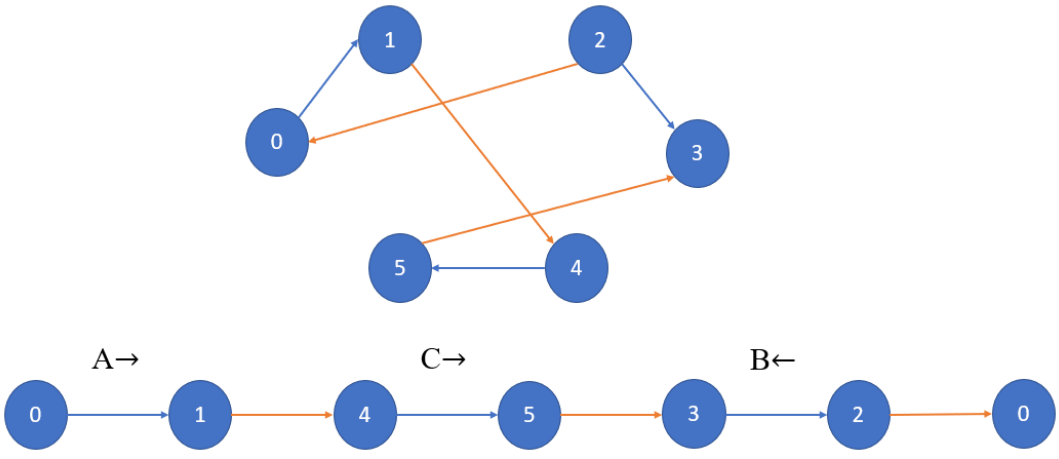
The second style is following, and we call it as [A→,C->,B->]:



The third style is following, it is [A->,C<-,B->]:



The fourth style is following, and it is [A->,C->,B<-]:



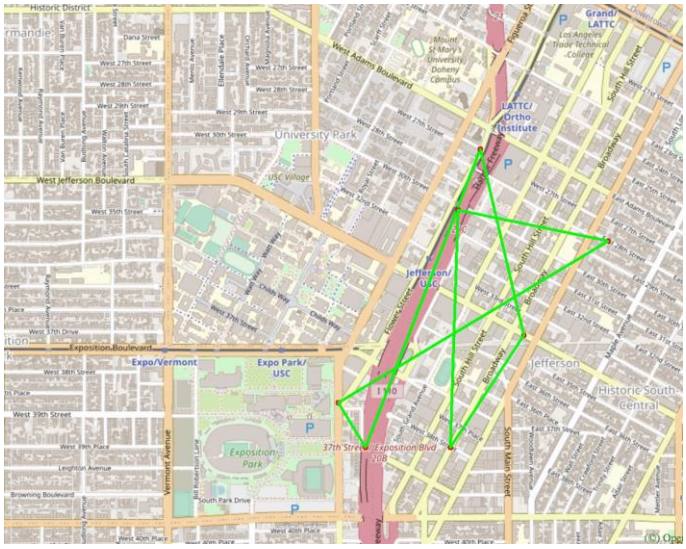
In addition, the other three style is [A->,B<-,C->],[A->,B->,C<-] and [A->,C<-,B<-]. They are also the style of 2-opt, so it is easy to modify 2-opt method code to 3-opt method by

adding the first four 3-opt swap methods in the 2-opt method code. The final time complexity of a single 3-opt method is $O(n^3)$.

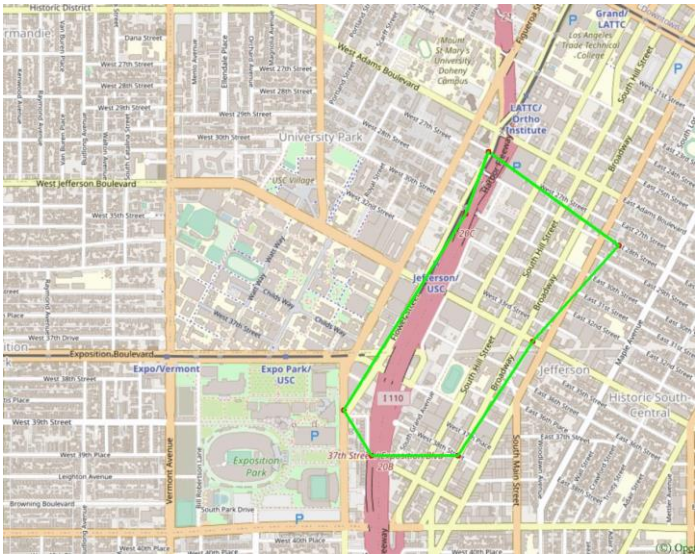
Here is the result of the TSP function based on 3-opt Method:

```
4835549605
6807663995
7424313408
123571748
348123159
4020099365
269633702
4835549605
************************************************************
The distance of the path is:2.49973
************************************************************
You could find your animation at src/lib/output.avi.
```

The animated plot(You can see the animated plot at the README file):



The final TSP route:



5. Time Complexity and behavior in the test case

I also constructed a test file to test the behavior of the four main functions for menu items. For the accuracy, my functions has passed all given unit test, so I will focus on the relationship between their theoretic time complexity and real run time. Here is their run time in one test case:

Autocomplete(): 75ms

FindPosition(): 75ms

CalculateShortestPath(): 185539ms(case 1), 197268ms(case 2)

TravellingTrojan() : 77ms(case 1), 80ms(case 2), 15063ms(case 3)

TravellingTrojan_2opt() : 67ms(case 1), 70ms(case 2), 903ms(case 3)

TravellingTrojan_3opt() : 71ms(case 1), 109ms(case 2), 26901ms(case 3)

For the first two functions, it has short runtime because both of them have a low time complexity(O(n)). The other two functions have some interesting problem which I would like to spend time to analyze.

The first problem is the extraordinary long run time in the function **CalculateShortestPath**(), both these two test cases have nearly 200s run time. I have two hypotheses in this part. The first hypothesis is the limitation of my hardware. I run and test this project on the Ubuntu system which is in a virtual machine. When I set this system, I haven't given this virtual machine a high parameter because I don't want to make it affect other software's behavior on my computer. Thus, I have this hypothesis that the limitation of my coding environment may cause the long run time.

My second hypothesis is the time complexity of the Bellman-Ford algorithm. In my function, I use three loops to iteratively update the d map and path map. In principle, the first loop will run n times, and the other two loops are only checking incoming edges, which makes the time complexity is O(n * (Sum(all incoming edges))) = O(n * m). However, I think that my function may have some unnecessary part that makes the 2nd or 3rd loop run the whole n times. It makes the real time complexity become $O(n^2 * m)$ or even $O(n^3)$ and extraordinary increasing the run time of this function.

The other problem I am focus on is the long run time of TravellingTrojan_3opt() : function in a big number of input points. Although I know the runtime of 3-opt method is O(n^3), which will be a bit longer than 2-opt method, it is abnormal that real run time of 3-opt method is longer than Brute Force method whose time complexity is O(n!). My hypothesis in this problem is that I may create too much branch in the 3-opt method. I use a recursive helper function to generate new routes and find the solution. Because 3-opt method has 7 different reconnecting methods, I create branch for different methods. Although I have combined 3 of them which can be think as 2-opt swap, there are still 5 branches in a single recursive function. I think that it will severely increasing the runtime of the function.

Above all, I think that I have learnt a lot from this project such as how to use c++ to program in a practical field, how to use different algorithms to solve problems. In addition, I also have some future plan related to programming field. For example, one of my main problems in this project is the extraordinary long run time in some function. It is necessary for me to know the complexity in a higher level and know how to optimize the time complexity of a programme.

6. Modified Final Video

Here is the link of the final video of the presentation for this project:

https://youtu.be/yUWVqpwIT1U