University of Southern California (USC)

EE599 – Computing Principles for Electrical Engineering

Final Project Report/README

Group 2: Runtime Terror

Name: Kunal Sheth (3656712856), Rohit Singh (6689387841)

• Abstract

This project focuses on Graph algorithms & is based on area near University of Southern California (USC), Los Angeles.

• What we changed

There are 4 functions provided: AutoComplete, Get the position, Shortest Path, Travelling Trojan.

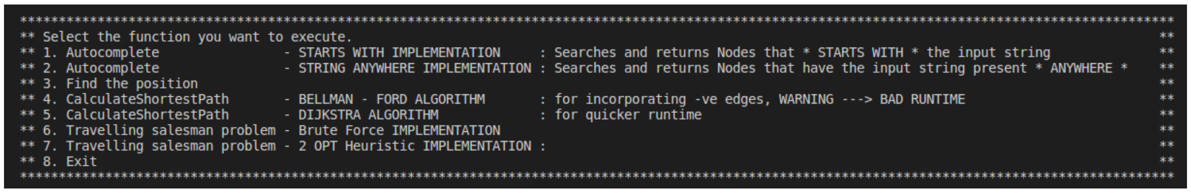
1. Main Menu: We modified the "CreateGraphfromCSVfile" - menu to incorporate various Helper functions.
2. AutoComplete: Added case insensitivity and corner cases for Testing boundary conditions.
3. Get the position: Added corner cases for Testing boundary conditions.
4. Shortest Path Dijkstra: Added corner cases for Testing boundary conditions.
5. Shortest Path Bellman Ford: Incorporated Bellman Ford algorithm.
6. Travelling Trojan: For this as there were 2 methods needed to be implemented: Brute-Force & 2_opt, we modified the function call in the "PrintMenu" function of "trojanmap.cc" file. We also added "Travellingtrojan_2opt" in "trojanmap.cc" & "trojanmap.h" files.

• Individual Contributions

1.	Main Menu Alterations	: Rohit
2.	GetName, GetID, Getdistance, Getpath code	: Kunal
3.	GetName, GetID, Getdistance, Getpath Gtest	: Rohit
4.	AutoComplete and GTest	: Rohit
5.	AutoComplete_2 and GTest	: Rohit
6.	Get the position and GTest	: Rohit
7.	Shortest Path Dijkstra and GTest	: Rohit
8.	Shortest Path Bellman Ford and GTest	: Rohit
9.	Travelling Trojan Brute-Force and GTest	: Kunal
10.	Travelling Trojan 2_opt and GTest	: Kunal

- **How we implemented**

a. MENU:

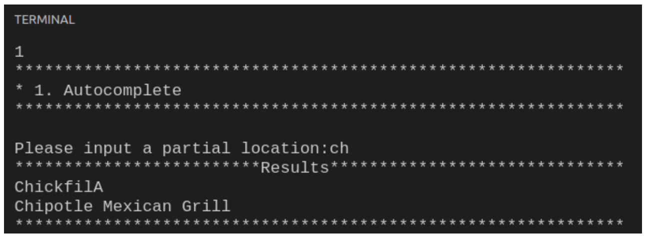


```
*******************************************************************************************************************************************************
** Select the function you want to execute.                                                                                                         **
** 1. Autocomplete                - STARTS WITH IMPLEMENTATION     : Searches and returns Nodes that * STARTS WITH * the input string                **
** 2. Autocomplete                - STRING ANYWHERE IMPLEMENTATION : Searches and returns Nodes that have the input string present * ANYWHERE *    **
** 3. Find the position                                                                                                                              **
** 4. CalculateShortestPath       - BELLMAN - FORD ALGORITHM       : for incorporating -ve edges, WARNING ---> BAD RUNTIME                           **
** 5. CalculateShortestPath       - DIJKSTRA ALGORITHM             : for quicker runtime                                                             **
** 6. Travelling salesman problem - Brute Force IMPLEMENTATION                                                                                       **
** 7. Travelling salesman problem - 2 OPT Heuristic IMPLEMENTATION :                                                                                 **
** 8. Exit                                                                                                                                           **
*******************************************************************************************************************************************************
```

b. <u>AutoComplete:</u>

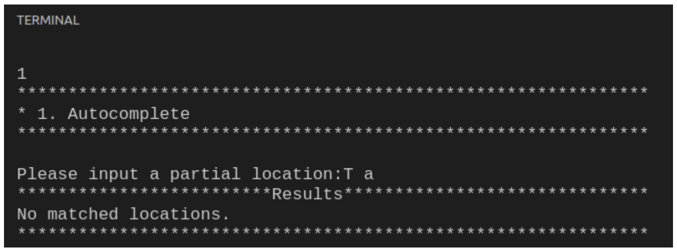
We incorporated Case Insensitivity and considered all the corner test cases. We created a vector to store all the data into it. We utilized the vector to operate the AutoComplete function. To optimize the Auto Complete Function, we added a new function – that test whether the input string is anywhere in the Nodes field.

<u>Time Complexity</u>: O(n) as we are working with vectors.



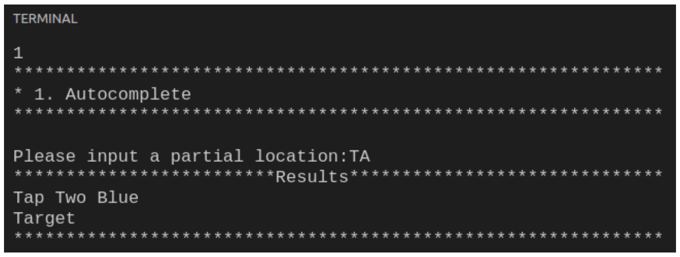
```
TERMINAL

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

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

Fig: 1.1



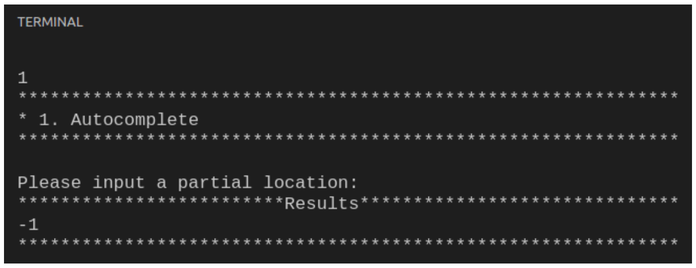
```
TERMINAL


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

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

Fig: 1.2



```
TERMINAL

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

Please input a partial location:TA
************************Results*****************************
Tap Two Blue
Target
**************************************************************
```

Fig: 1.3

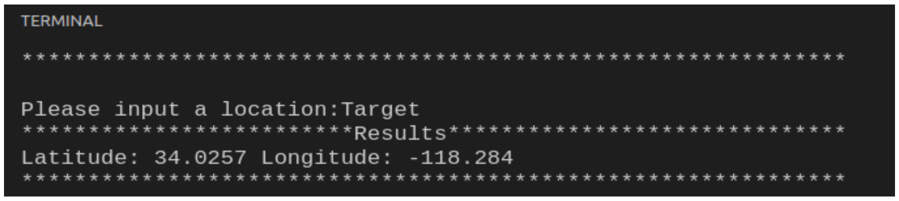


```
TERMINAL


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

Please input a partial location:
************************Results*****************************
-1
**************************************************************
```

Fig: 1.4

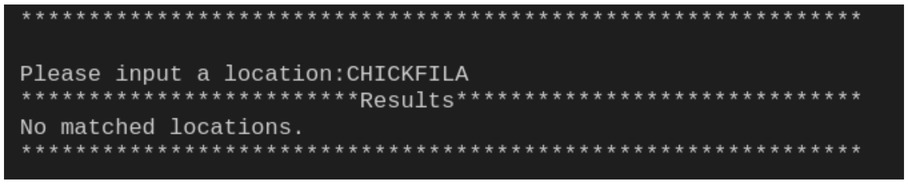
c. **<u>Get the position</u>**: We return the latitude and longitude for the given input string.

<u>Time Complexity</u>: O(logn) as we are using the map data.



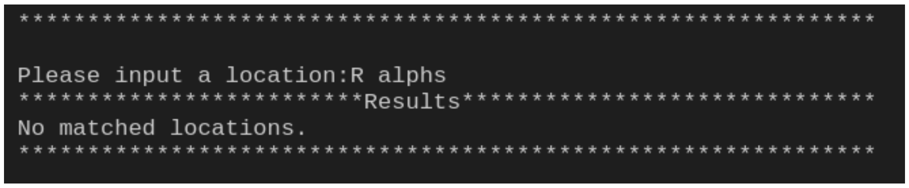
```
TERMINAL

**************************************************************

Please input a location:Target
*************************Results******************************
Latitude: 34.0257 Longitude: -118.284
**************************************************************
```

Fig: 2.1



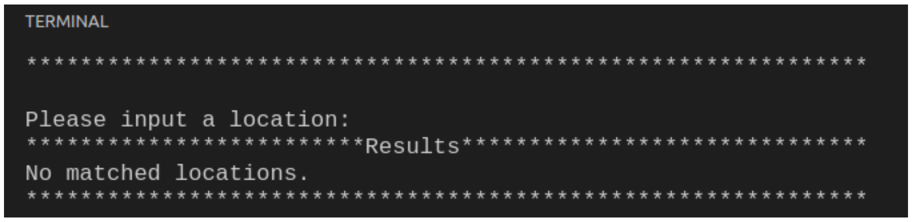
```
***************************************************************

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

Fig: 2.2



```
****************************************************************

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

Fig: 2.3



```
TERMINAL

****************************************************************

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

Fig: 2.4

d. **<u>Shortest Path</u>**:

1. **Dijkstra Algorithm:** In this algorithm we are interesting from the starting position and greedily deciding the edges, according to the min. distance from the source to the child of the current node. The min heap accepts the nodes and it's corresponding distance from the source. As we insert nodes into the heap, it automatically sorts it, and prepares the min. distance node at the top of the heap. This saves the runtime for calculating the min. distance from source. Additionally, we compare the weights every time we work will all the children of the current node.

<u>Time Complexity</u>: O (m + n log n)

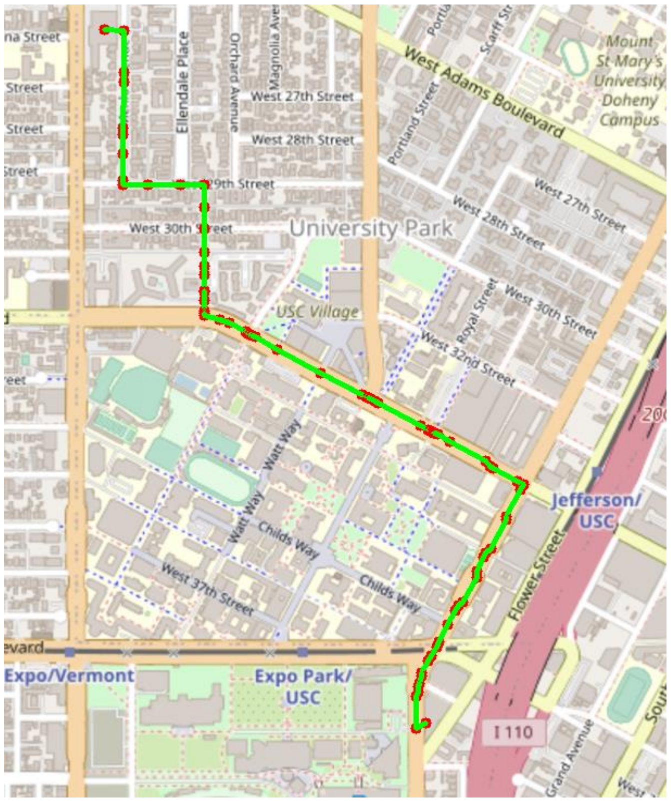
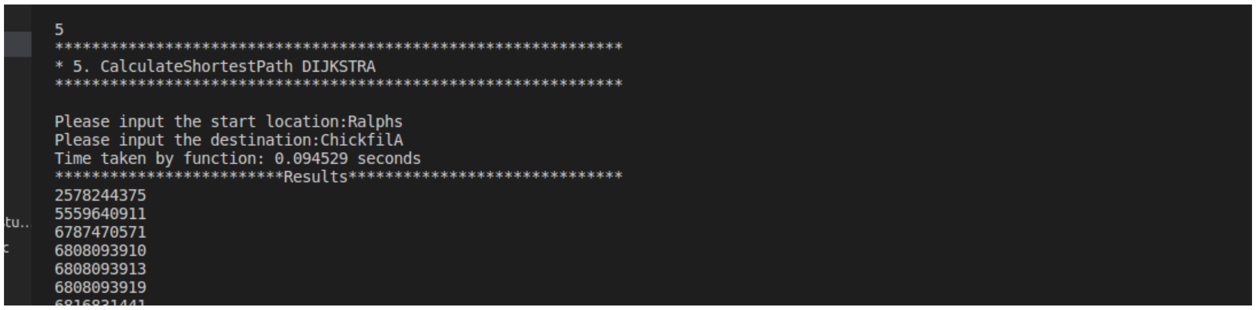


Fig: 3.1.1



```
5
**************************************************************
* 5. CalculateShortestPath DIJKSTRA
**************************************************************

Please input the start location:Ralphs
Please input the destination:ChickfilA
Time taken by function: 0.094529 seconds
*************************Results*****************************
2578244375
5559640911
6787470571
6808093910
6808093913
6808093919
```

Fig: 3.1.2

2. **Bellman Ford Algorithm:** In this algorithm we are traversing all nodes present. Like Dijkstra's Algorithm, we compare the min. distance from the source to the child of the current node. We lack the advantage of the priority que, in bellman ford because we are not using that. We traverse between all nodes and its children, comparing the weights every time we work will all the children of the current node.

   Time Complexity: O (m*n)

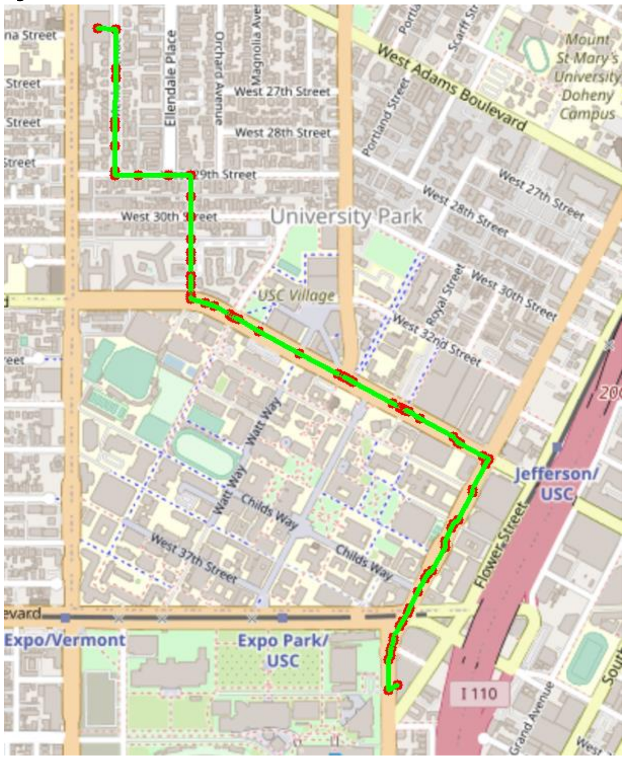
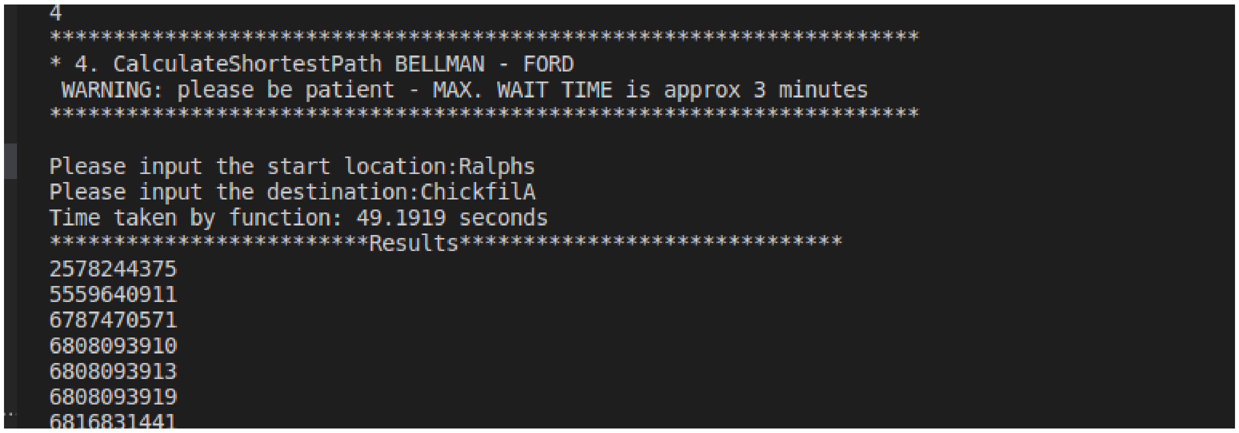


Fig: 3.2.1



```
4
*********************************************************************
* 4. CalculateShortestPath BELLMAN - FORD
 WARNING: please be patient - MAX. WAIT TIME is approx 3 minutes
*********************************************************************

Please input the start location:Ralphs
Please input the destination:ChickfilA
Time taken by function: 49.1919 seconds
*************************Results******************************
2578244375
5559640911
6787470571
6808093910
6808093913
6808093919
6816831441
```

Fig: 3.2.1

e. Travelling Trojan

1. **Brute Force Method**: Over here on this method, whenever the current path length is larger than the current optimal result we will just return. The input to the function was a "<vector <string>>" and the output was a "pair of <double, vector<vector<string>>>". Here we firstly built a graph. Then as the graph is a cyclic one, random location was chosen out of all the provided one's as a starting and ending location. We then generated all (n-1)! of the locations & calculated the current__path weight of every possible permutation keeping the track of the minimum weight permutation & finally returned the minimum cost out of all. This method is the best possible method for the Travelling salesmen as every possible scenario will be covered. The time complexity for the brute force method is mentioned below. The output of the brute force method for 9 location is shown below. However, as the number of input location increases, this method is too slow, and we switch to different method. One such method is 2_opt discussed next.

Time complexity: O(n!)

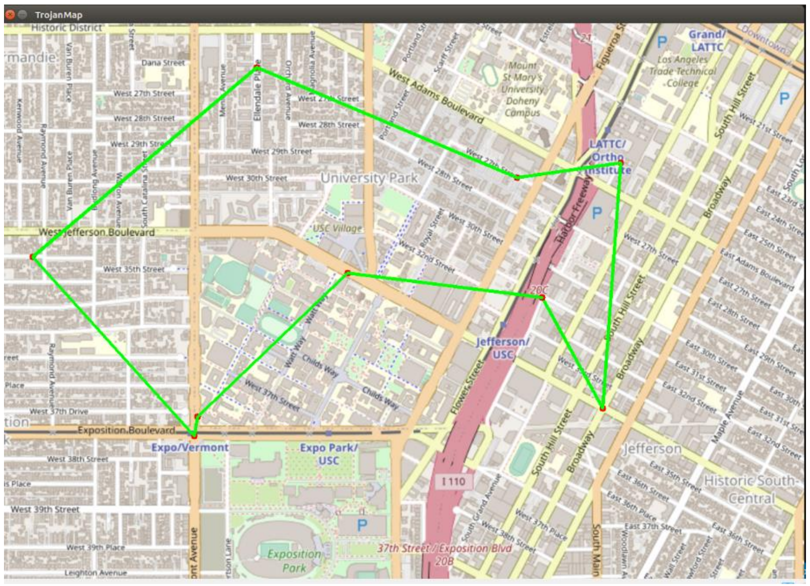
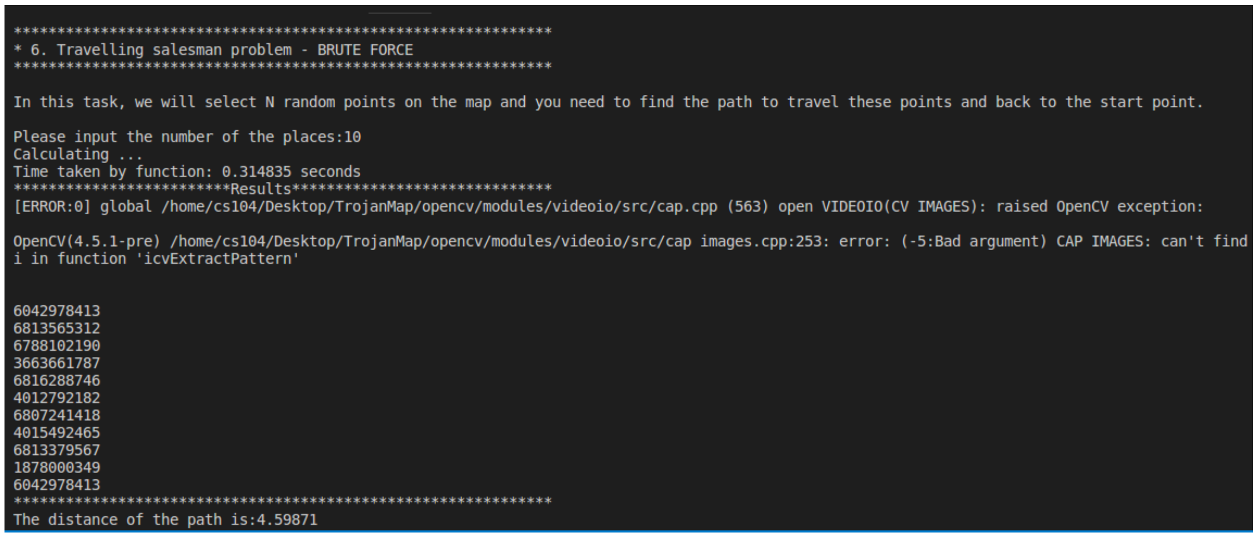


Fig: 4.1.1



```
**************************************************************
* 6. Travelling salesman problem - BRUTE FORCE
**************************************************************

In this task, we will select N random points on the map and you need to find the path to travel these points and back to the start point.

Please input the number of the places:10
Calculating ...
Time taken by function: 0.314835 seconds
*************************Results*****************************
[ERROR:0] global /home/cs104/Desktop/TrojanMap/opencv/modules/videoio/src/cap.cpp (563) open VIDEOIO(CV IMAGES): raised OpenCV exception:

OpenCV(4.5.1-pre) /home/cs104/Desktop/TrojanMap/opencv/modules/videoio/src/cap images.cpp:253: error: (-5:Bad argument) CAP IMAGES: can't find
i in function 'icvExtractPattern'


6042978413
6813565312
6788102190
3663661787
6816288746
4012792182
6807241418
4015492465
6813379567
1878000349
6042978413
**************************************************************
The distance of the path is:4.59871
```

2. **2_opt Heuristic Method**: Over here, the input as well as output remain the same as that of the brute force method. This method is a heuristic one where we keep swapping the nodes till the time there is no improvement. Sometimes it can find local optimal results instead of a global one. One such picture of the concept is shown below:

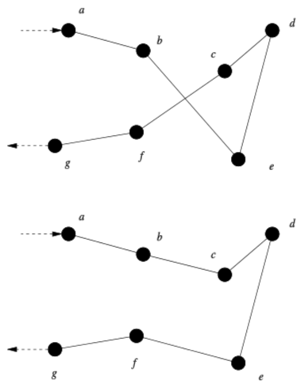


Fig: 4.2.1

The important thing to notice over here is that if we start or end at a particular node, then we must pop this from the search as an eligible candidate for swapping, as reversing the order will cause an invalid path. Here the time taken of large sets of input location is very less compared to that of the brute force method. The time complexity and the output from our implementation are mentioned & shown below for 2_opt. This concept can also be extended to other heuristic method called "3_opt", but that is currently out of scope at this point of time.

Below is also the comparison & visualization between the Brute force & 2_opt method.

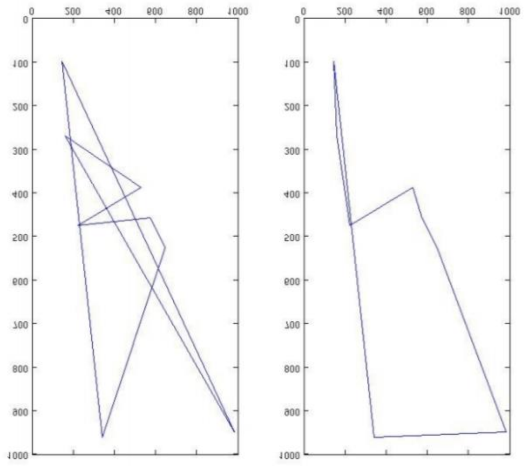


Fig: 4.2.2(left one: brute force, right one: 2_opt)

Time Complexity: O(n^2)

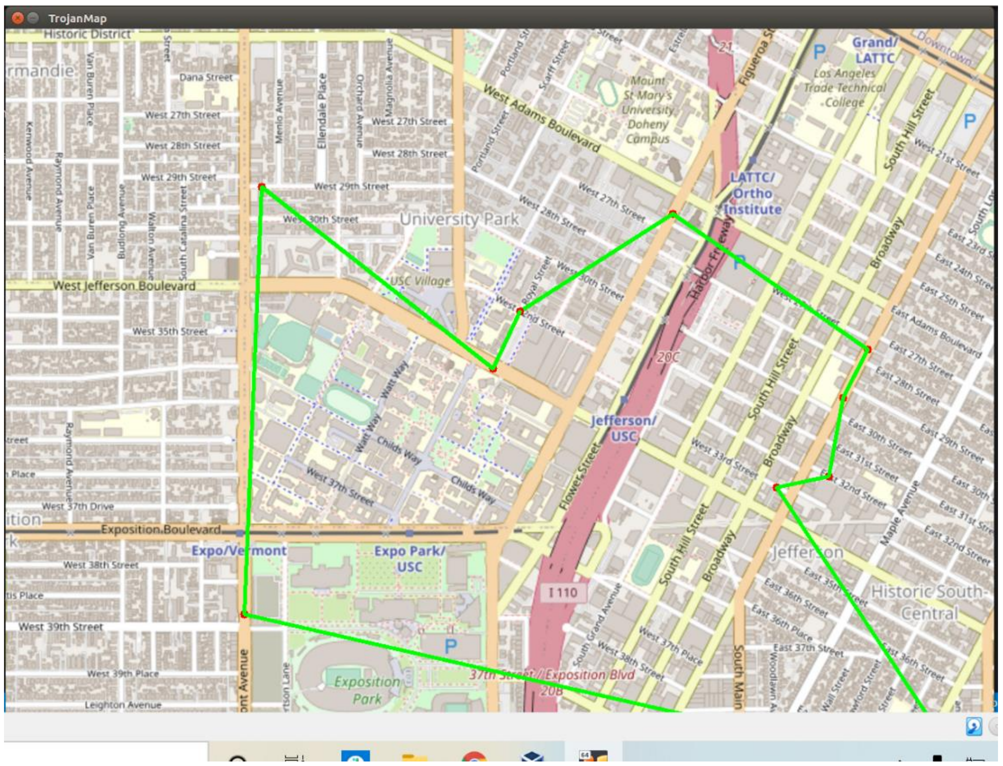
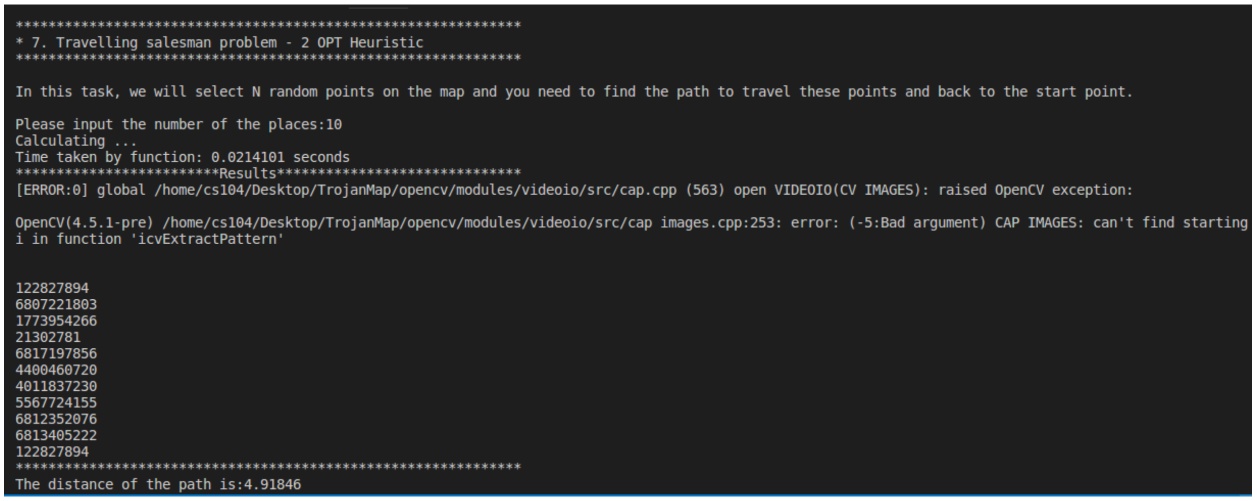


Fig: 4.2.3



```
**************************************************************
* 7. Travelling salesman problem - 2 OPT Heuristic
**************************************************************

In this task, we will select N random points on the map and you need to find the path to travel these points and back to the start point.

Please input the number of the places:10
Calculating ...
Time taken by function: 0.0214101 seconds
*************************Results******************************
[ERROR:0] global /home/cs104/Desktop/TrojanMap/opencv/modules/videoio/src/cap.cpp (563) open VIDEOIO(CV IMAGES): raised OpenCV exception:

OpenCV(4.5.1-pre) /home/cs104/Desktop/TrojanMap/opencv/modules/videoio/src/cap images.cpp:253: error: (-5:Bad argument) CAP IMAGES: can't find starting
i in function 'icvExtractPattern'


122827894
6807221803
1773954266
21302781
6817197856
4400460720
4011837230
5567724155
6812352076
6813405222
122827894
**************************************************************
The distance of the path is:4.91846
```

- ## What did we learn?

1. The most important thing we learned is “think and implement” & not “implement & think”.
2. In depth understanding of the usage of majorly used data structures and algorithms.
3. Debugging skills, we faced a lot of challenges & thus debugging skills for us improved a lot over our implementation as after that we were able to visualize about some corner cases that we missed.
4. Presentation skills. How we present ourselves is the most important thing which we learnt from this project.
5. Got to learn about the practical scenario & real-world implementation for shortest path algorithms beyond our project like for IP routing to find the open shortest path first, robotic path etc.

- ## References

1. https://www.geeksforgeeks.org/traveling-salesman-problem-tsp-implementation/
2. https://en.wikipedia.org/wiki/2-opt
3. http://cs.indstate.edu/~zeeshan/aman.pdf