# CS150 Project 3 Report

Oanh Doan

December 16, 2017

## 1 Introduction

This program allows users to generate a tour agenda that fits in their time budget but still maximizes the number of sites visited [3]. To create an agenda, the user needs to specify a starting point, time constraint, and all desired tourist sites.

The program has a map of all hotels, tourist sites, traveling time between places, and suggested time to spend at each attraction. Based on this map and user's input, the program will generate the best agenda.

- In case of no site preferences, at every location, the program looks for the nearest site that can be reached. Once a site is found, the program check two conditions to decide whether or not that site should be added to the tour:
  - Can the site be reached within the remaining time?
  - Do we still have enough time to return to the starting point after we have traveled there?

  If both of these conditions are met, the program includes this site in the tour. Otherwise, we shall return to the starting point.
- If there are some desired destinations, from the starting point, we will first go to as many of them as possible until time runs out. Along the way, as we reach any sites, we still have to check the two conditions above. The only difference is that instead of looking for the nearest site in general, we look for the nearest site in the specified list of favorite places. If there is still some time left after we have visited all of the desired places, we shall continue to travel according to the rules in the first case.

This is done by using the Dijkstra algorithm to find the shortest path between any two points. As we arrive at a site, we mark the site as visited so that we will not consider revisiting it as a tourist attraction. However, at some point, on the way home, we can still pass by the site if it is required to achieve the shortest path, but we would not spend time at the site.

# 2 Approach

## 2.1 Program design

The design of the graph in this program is based on Lab 7 [8]. The program contains the following classes:

- generic Node: stores the key of the node and outgoing egdes. For each nodes, we can find its adjacent nodes by following its outgoing edges to the other ends.
- generic Edge: stores two nodes at two ends and the weight of the edge.
- generic Hotel: extends the Node class.
- generic Site: extends the Node class and stores the suggested amout of time to spend at this site.
- generic DirectedGraph: contains lists of nodes, hotels, and sites. A node in this graph can be a hotel, a site, or just a transitive node. Even though the lists of hotels and sites are subsets of the list of all nodes, we will need them to make it easier to search for or gain access to a site/a hotel among hundreds of other nodes in the graph.
- Path: stores a starting node, an ending node, along with intermediate nodes needed to get to the destination. It also contains a graph to which its nodes belong to. Adding the weights of edges between all intermediate nodes, we can find the length of the path.
- Tour: stores a graph, start node, and time limit.
- Controller: creates map by reading input files and create tours based on the user's input.

There are also 4 unit test classes:

- NodeTest
- DirectedGraphTest
- PathTest
- TourTest

There are no testing classes for the other classes because they contain very simple get/set methods or are nest-tested in the four classes above. In particular, the implementation and testing of methods in Node, Edge, and DirectedGraph classes come from Lab 7 [2].

### 2.2 Choice of data structure and algorithms

- ArrayList [7]: Along with HashMap, ArrayList is one the two most commonly used data structures in this program. Often, the list is used to store graph nodes. For example, the DirectedGraph class has 3 node lists; the Tour class has a list of intermediate nodes that represents the itinerary of a tour, and so does the Path class.
- HashMap [4]: HashMap is used locally within methods of classes. Often, the key of the map is node and the value is a boolean whether or not the node has been visited, or a path whose destination is the node itself. For example, from a starting point, to search for the next nearest site, we create a hashmap in which the keys are all sites and the values are distances from starting point to the site.
- Comparable interface [6]: is used to make the keys of nodes comparable.
- Dijkstra algorithm [1]: This is the key algorithm of this program. It is used to find the shortest path between any two nodes in the class. It returns null if the two nodes do not exist in the graph, or returns a path of infinite length if one node is unreachable from the other. The algorithm is that we keep visiting the nearest non-visited node until we have arrived at the specified destination.
- Scanner [5] is used to read input files.

## 3 Methods

We first run the program using a small file of 13 nodes with 1 hotels and 5 sites to verify the correctness of our program. Then we will use a bigger file of 199 nodes with 13 hotels and 5 sites. With each of these file, we will attempt to:

- For a fixed time limit and starting node, generate tours with and without preferences,then compare the total time spent and maximum number of sites visited.
- For a fixed time limit and preferences, change the starting node and observe the differences.
- For the same tour, change the order in which desired sites are specified.

## 4 Data and analysis

### 4.1 Experiment with small dataset

Figure 1 represents the graph of the small input file we use to test program correctness. In this graph, orange nodes are sites with suggested visiting time to be highlighted in red. Green nodes are hotels, and blue nodes are transitive nodes.

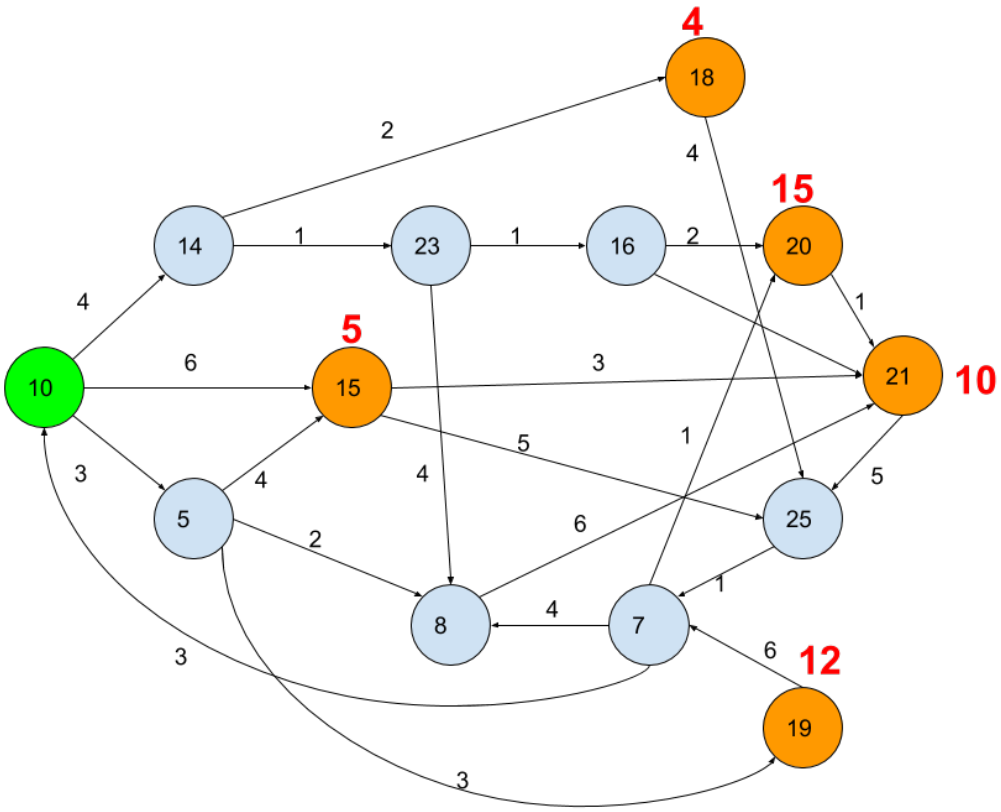


Figure 1. Graph of small input file

```
Specify hotel, time limit, and favorite sites:
10 50
Generated tour: 10 14 18 25 7 20 21 25 7 10
Total traveling and visiting time: 43
```

Figure 2. No desired destination

Figure 2 shows the path of the tour in 50 minutes, starting at node 10. The tour visits 18 and 20 before returning to 10. On its way back, even though the tour passes by 20, it does not stop to visit the site. Therefore, the total traveling and visiting time is 43.

```
Specify hotel, time limit, and favorite sites:
10 50 19 18
Generated tour: 10 5 19 7 10 14 18 25 7 10
Total traveling and visiting time: 45
```

Figure 3. Two desired destinations

Figure 3 shows how the program prioritizes desired destinations and visit them first before considering other places. However, in this case, since we run out of time after visiting node 18, we cannot visit any other sites. This gives us the motivation to increase the time limit to see how many more sites we can visit with the same location preference.

```
Specify hotel, time limit, and favorite sites:
10 80 18 19
Generated tour: 10 14 18 25 7 10 5 19 7 20 21 25 7 10
Total traveling and visiting time: 80
```

Figure 4. Two desired destinations with more time

Indeed, with 20 more minutes, we actually have enough time to visit one more site, which is node 20. Also, since 18 and 19 are equally distanced from 10, thus the program bases its decision to pick which one to visit on the order in which the desired sites are specified.

## 4.2 Experiment with big dataset

Next, we examine the results using the big input file.

| | Start Node | Time limit | Desired Sites | Total Time (travel + visit) | Visiting time | Visited Sites | Route |
|---|---|---|---|---|---|---|---|
| 1 | 130 | 50 | N/A | 29 | 25 | 103 | 130 103 16 19 130 |
| 2 | 130 | 100 | N/A | 100 | 89 | 103 72 157 165 | 130 103 72 64 157 51 153 165 16 19 130 |
| 3 | 130 | 150 | N/A | 159 | 139 | 103 72 157 165 100 | 130 103 72 64 157 51 153 165 134 100 26 130 |
| 4 | 130 | 500 | N/A | 336 | 331 | 103 72 157 165 100 26 112 68 190 140 150 177 | 130 103 72 64 157 51 153 165 134 100 26 112 4 68 65 190 189 140 19 63 150 54 177 181 179 80 26 130 |
| 5 | 130 | 50 | 157 | 39 | 29 | 157 165 | 130 103 151 157 51 153 165 16 19 130 |
| 6 | 130 | 100 | 157 | 89 | 79 | 157 165 100 | 130 103 151 157 51 153 165 134 100 26 130 |
| 7 | 130 | 150 | 112 103 26 150 | 109 | 98 | 103 26 112 150 | 130 103 11 26 112 4 137 150 100 26 130 |
| 8 | 130 | 150 | 26 150 103 112 | 111 | 98 | 103 112 26 150 | 130 103 16 112 125 26 4 137 150 100 26 130 |
| 9 | 130 | 100 | 100 179 | 75 | 70 | 100 26 | 130 31 168 100 26 130 |
| 10 | 130 | 100 | 179 100 | 80 | 71 | 179 190 | 130 103 179 190 189 140 19 130 |
| 11 | 130 | 150 | 100 179 | 134 | 121 | 100 179 190 | 130 31 168 100 126 179 190 189 140 19 130 |
| 12 | 130 | 150 | 179 100 | 121 | 111 | 179 100 26 | 130 103 179 80 26 4 100 26 130 |
| 13 | 181 | 50 | N/A | 35 | 29 | 157 165 | 181 157 51 153 165 66 181 |
| 14 | 181 | 100 | N/A | 88 | 79 | 157 165 100 | 181 157 51 153 165 66 181 |
| 15 | 181 | 100 | 100 179 | 77 | 70 | 100 26 | 181 131 63 100 26 4 57 181 |
| 16 | 181 | 100 | 179 100 | 99 | 90 | 179 190 165 | 181 179 190 165 66 181 |

Table 1. Tours generated using a graph of 199 nodes

- In the first 4 observations, starting at the same node (130), without any site preferences, the number of sites and total time increase as the time limit gradually increases. However, once the time limit goes up to 500, the tour actually has enough time to visit all sites in the graph, as evidenced in row 4. Indeed, the tour only needs 331 minutes to visit all sites and return to its starting point.

- In the $5^{th}$ and $6^{th}$ observations, we add one site preference while keeping the starting point unchanged and increasing time limit gradually. In both cases, the tour prioritizes visiting the desired location. After the desired site is visited, it considers visiting other sites.

- In the next 2 observations, we add a few more desired places and switch the order in which these sites are specified. We learn that this order matters because different orders generate different routes and traveling time. For example, the path in $8^{th}$ is actually longer and takes more time to finish. This is indeed a shortcoming of this program. The program should generate the same path regardless of the order in which the desired sites are specified. Otherwise, it is not the shortest path because it is possible to generate a shorter path by switching the order.

- In the next 4 observations, we pick the two most time-consuming sites as our desired sites, which together will take 91 minutes to visit. Thus, within 100 minutes, we could not afford to travel to both places and return. In fact, the program chooses to visit the first one to be specified along with another regular site, which depends on the choice of the first one. If we increase the time limit by 50, we could visit both sites and even an additional one.

- We choose to repeat some experiments with a different starting point (181). Even though the paths are completely different, some patterns we have observed remain unchanged. For example, desired sites are always prioritized and the order in which desired sites are listed matters. In addition, because of differences in paths, the maximum number of sites to be visited and total traveling time also differ. For instance, starting from 181, with 100 minutes, we can visit 1 site less than starting from 130.

# 5 Conclusion

- The number of sites visited depends on not only the time limit, but also the start node and whether or not there are any desired attractions.

- The order in which desired sites are listed causes changes to the path, which is a drawback that needs to be improved.

- The program always prioritizes visiting desired places before considering other sites.

# References


[1] Canvas. Dijkstra's algorithm. https://canvas.instructure.com/courses/1171720/assignments/6432426?module_item_id=12945391.

[2] Oanh Doan. Lab 7 testing and implementation. https://drive.google.com/drive/folders/1DGqsx3amctwJTVeZv6bUOfYR6Okvh_J8?usp=sharing.

[3] Chun Wai Liew. Project description. https://goo.gl/JgQJbp.

[4] Orable. Hashmap. https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html.

[5] Orable. Scanner. https://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html.

[6] Oracble. Comparable interface. https://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html.

[7] Oracle. Arraylist (java platform se 8). goo.gl/H11bwDcontent_copyCopyshortURL.

[8] Linda Plotnick. Lab 7 description. https://goo.gl/dBhTyx.