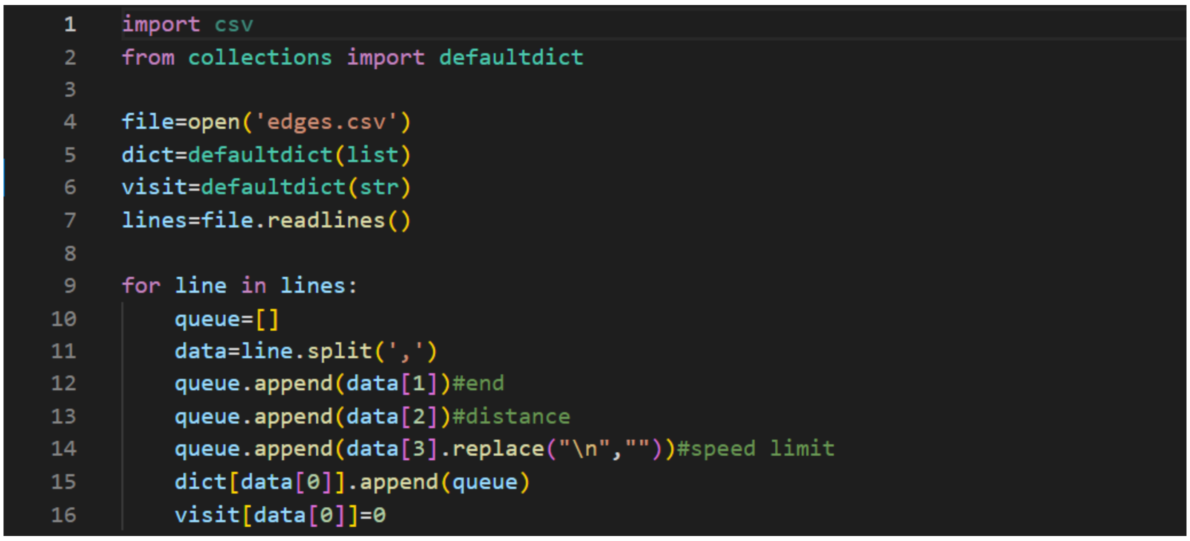
Homework 2: Route Finding Report Template

Please keep the title of each section and delete examples. Note that please keep the questions listed in Part III.

Part I. Implementation (6%):

- **Please screenshot your code snippets of Part 1 ~ Part 4, and explain your implementation. For example,**



```python
import csv
from collections import defaultdict

file=open('edges.csv')
dict=defaultdict(list)
visit=defaultdict(str)
lines=file.readlines()

for line in lines:
    queue=[]
    data=line.split(',')
    queue.append(data[1])#end
    queue.append(data[2])#distance
    queue.append(data[3].replace("\n",""))#speed limit
    dict[data[0]].append(queue)
    visit[data[0]]=0
```

Before programming the BFS algorithm, first I prepare for the data set.

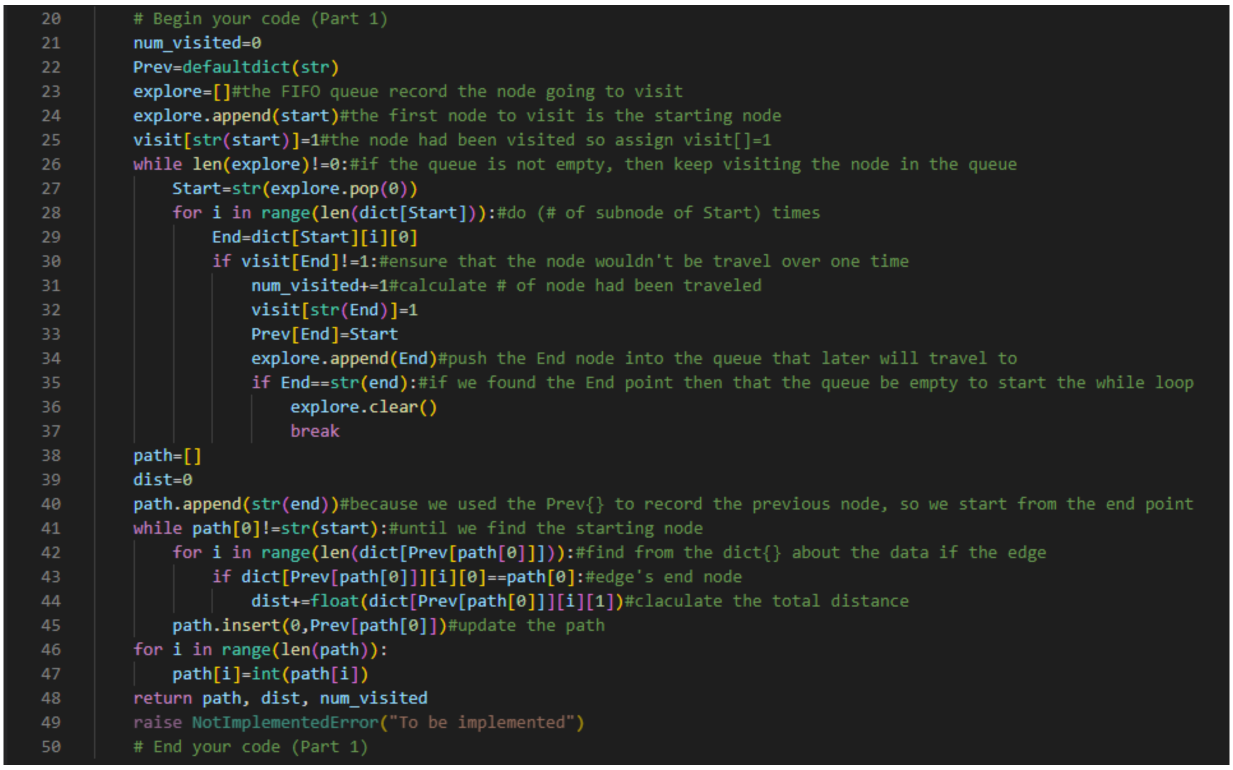
dict{ }: construct a dictionary to record the data of each edge.

dict[starting node ID]={ end node ID, distance, speed limit }

visit{ }: use to record if the node had been travel before, set the default as 0 to represent not yet to travel.

visit[node]=0(not yet)/1(traveled)

Part1:



```python
# Begin your code (Part 1)
num_visited=0
Prev=defaultdict(str)
explore=[]#the FIFO queue record the node going to visit
explore.append(start)#the first node to visit is the starting node
visit[str(start)]=1#the node had been visited so assign visit[]=1
while len(explore)!=0:#if the queue is not empty, then keep visiting the node in the queue
    Start=str(explore.pop(0))
    for i in range(len(dict[Start])):#do (# of subnode of Start) times
        End=dict[Start][i][0]
        if visit[End]!=1:#ensure that the node wouldn't be travel over one time
            num_visited+=1#calculate # of node had been traveled
            visit[str(End)]=1
            Prev[End]=Start
            explore.append(End)#push the End node into the queue that later will travel to
            if End==str(end):#if we found the End point then that the queue be empty to start the while loop
                explore.clear()
                break
path=[]
dist=0
path.append(str(end))#because we used the Prev{} to record the previous node, so we start from the end point
while path[0]!=str(start):#until we find the starting node
    for i in range(len(dict[Prev[path[0]]])):#find from the dict{} about the data if the edge
        if dict[Prev[path[0]]][i][0]==path[0]:#edge's end node
            dist+=float(dict[Prev[path[0]]][i][1])#claculate the total distance
    path.insert(0,Prev[path[0]])#update the path
for i in range(len(path)):
    path[i]=int(path[i])
return path, dist, num_visited
raise NotImplementedError("To be implemented")
# End your code (Part 1)
```

About the BFS algorithm, the main idea is that find from the starting node, then each step travel all its subnode and push them in to a queue, what the queue store is the node we are going to travel. (Notice that for BFS the stack should be FIFO first in first out)

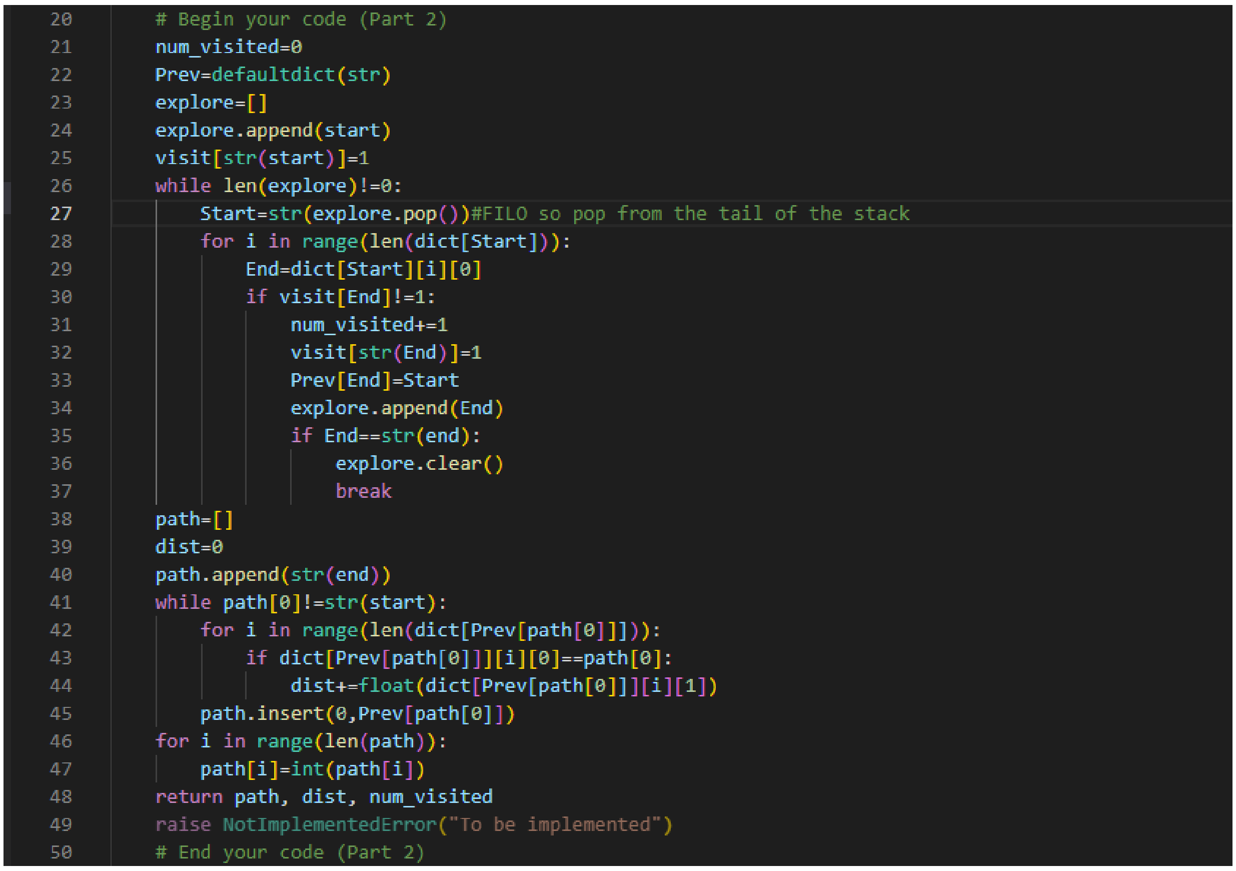
Prev{ }: dictionary that use to record the previous node for each node, so that we can
find the final path easily.

explore[ ]: the queue of BFS

visit[ ]: record if the node had been travel or not, so that we won't repeat to travel same
node.

path[ ]: record the final path from start node to end node, and return.

Part2:

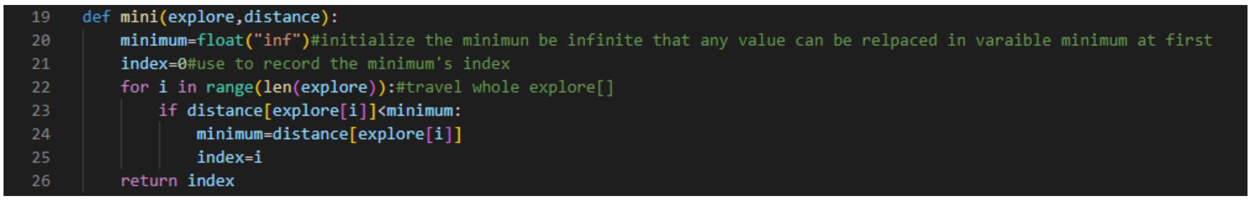


```
20      # Begin your code (Part 2)
21      num_visited=0
22      Prev=defaultdict(str)
23      explore=[]
24      explore.append(start)
25      visit[str(start)]=1
26      while len(explore)!=0:
27          Start=str(explore.pop())#FILO so pop from the tail of the stack
28          for i in range(len(dict[Start])):
29              End=dict[Start][i][0]
30              if visit[End]!=1:
31                  num_visited+=1
32                  visit[str(End)]=1
33                  Prev[End]=Start
34                  explore.append(End)
35                  if End==str(end):
36                      explore.clear()
37                      break
38      path=[]
39      dist=0
40      path.append(str(end))
41      while path[0]!=str(start):
42          for i in range(len(dict[Prev[path[0]]])):
43              if dict[Prev[path[0]]][i][0]==path[0]:
44                  dist+=float(dict[Prev[path[0]]][i][1])
45          path.insert(0,Prev[path[0]])
46      for i in range(len(path)):
47          path[i]=int(path[i])
48      return path, dist, num_visited
49      raise NotImplementedError("To be implemented")
50      # End your code (Part 2)
```

The only different between BFS and DFS is, BFS use FIFO queue and DFS use FILO stack. In that case, just change the code in line 27 into pop() then it will turn to FILO.

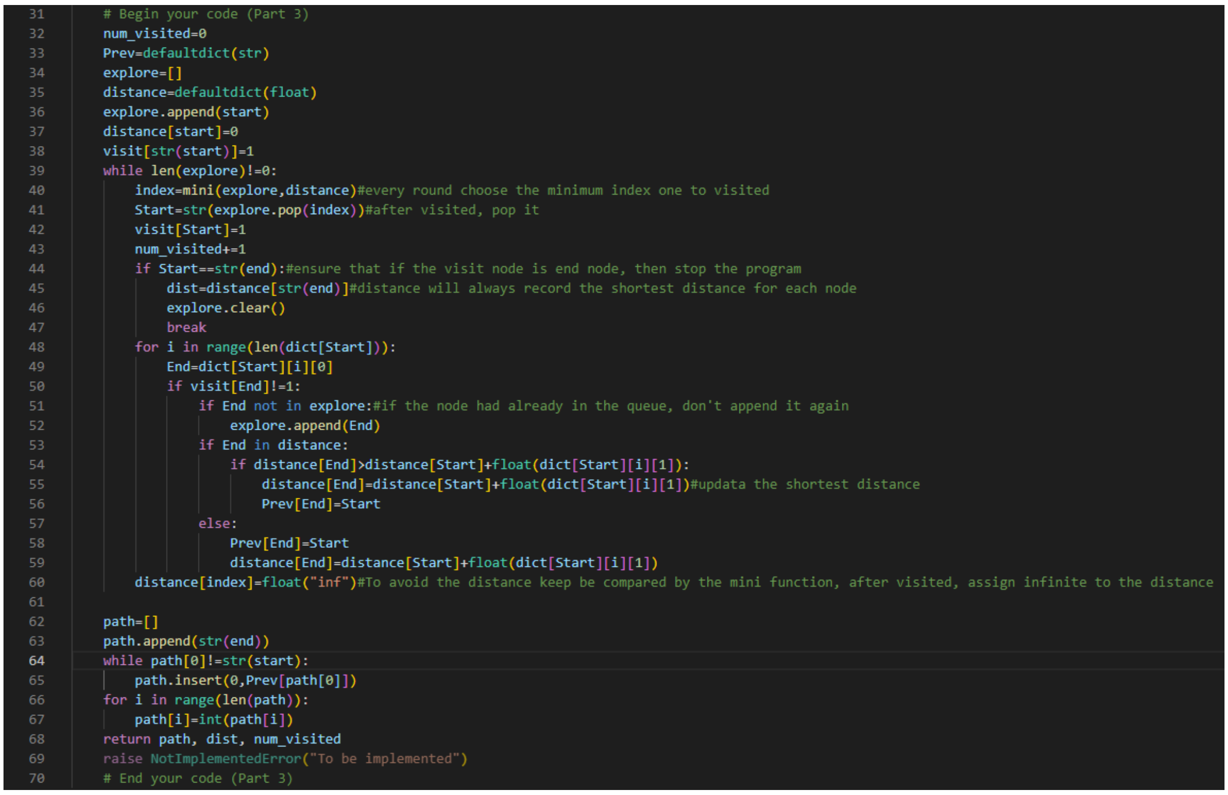
Part3:

The main idea for UCS algorithm is choose the shortest distance in each round.



```python
19  def mini(explore,distance):
20      minimum=float("inf")#initialize the minimun be infinite that any value can be relpaced in varaible minimum at first
21      index=0#use to record the minimum's index
22      for i in range(len(explore)):#travel whole explore[]
23          if distance[explore[i]]<minimum:
24              minimum=distance[explore[i]]
25              index=i
26      return index
```

mini function: return the minimum distance value's index.

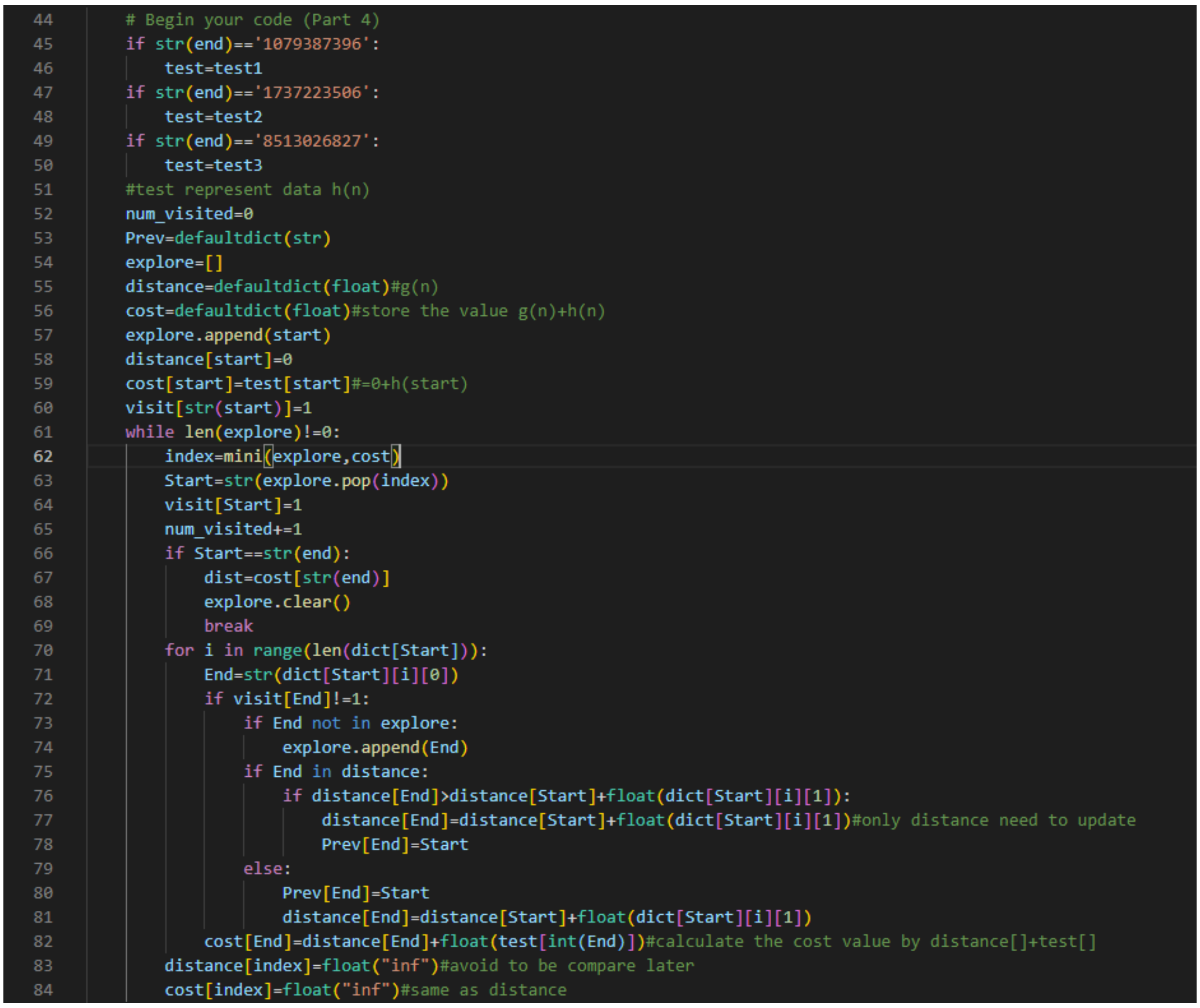


```python
31      # Begin your code (Part 3)
32      num_visited=0
33      Prev=defaultdict(str)
34      explore=[]
35      distance=defaultdict(float)
36      explore.append(start)
37      distance[start]=0
38      visit[str(start)]=1
39      while len(explore)!=0:
40          index=mini(explore,distance)#every round choose the minimum index one to visited
41          Start=str(explore.pop(index))#after visited, pop it
42          visit[Start]=1
43          num_visited+=1
44          if Start==str(end):#ensure that if the visit node is end node, then stop the program
45              dist=distance[str(end)]#distance will always record the shortest distance for each node
46              explore.clear()
47              break
48          for i in range(len(dict[Start])):
49              End=dict[Start][i][0]
50              if visit[End]!=1:
51                  if End not in explore:#if the node had already in the queue, don't append it again
52                      explore.append(End)
53                  if End in distance:
54                      if distance[End]>distance[Start]+float(dict[Start][i][1]):
55                          distance[End]=distance[Start]+float(dict[Start][i][1])#updata the shortest distance
56                          Prev[End]=Start
57                  else:
58                      Prev[End]=Start
59                      distance[End]=distance[Start]+float(dict[Start][i][1])
60          distance[index]=float("inf")#To avoid the distance keep be compared by the mini function, after visited, assign infinite to the distance
61
62      path=[]
63      path.append(str(end))
64      while path[0]!=str(start):
65          path.insert(0,Prev[path[0]])
66      for i in range(len(path)):
67          path[i]=int(path[i])
68      return path, dist, num_visited
69      raise NotImplementedError("To be implemented")
70      # End your code (Part 3)
```

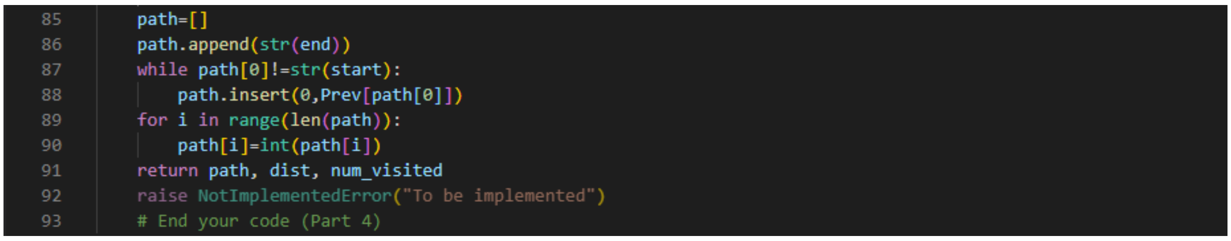
There are some change between UCS and BFS/DFS in line 63-68.

For UCS the total distance had already been record in distance[ ], so don't need to calculate again in the while loop.n

Part4:



```python
    # Begin your code (Part 4)
    if str(end)=='1079387396':
        test=test1
    if str(end)=='1737223506':
        test=test2
    if str(end)=='8513026827':
        test=test3
    #test represent data h(n)
    num_visited=0
    Prev=defaultdict(str)
    explore=[]
    distance=defaultdict(float)#g(n)
    cost=defaultdict(float)#store the value g(n)+h(n)
    explore.append(start)
    distance[start]=0
    cost[start]=test[start]#=0+h(start)
    visit[str(start)]=1
    while len(explore)!=0:
        index=mini(explore,cost)
        Start=str(explore.pop(index))
        visit[Start]=1
        num_visited+=1
        if Start==str(end):
            dist=cost[str(end)]
            explore.clear()
            break
        for i in range(len(dict[Start])):
            End=str(dict[Start][i][0])
            if visit[End]!=1:
                if End not in explore:
                    explore.append(End)
                if End in distance:
                    if distance[End]>distance[Start]+float(dict[Start][i][1]):
                        distance[End]=distance[Start]+float(dict[Start][i][1])#only distance need to update
                        Prev[End]=Start
                else:
                    Prev[End]=Start
                    distance[End]=distance[Start]+float(dict[Start][i][1])
            cost[End]=distance[End]+float(test[int(End)])#calculate the cost value by distance[]+test[]
        distance[index]=float("inf")#avoid to be compare later
        cost[index]=float("inf")#same as distance
```



```python
    path=[]
    path.append(str(end))
    while path[0]!=str(start):
        path.insert(0,Prev[path[0]])
    for i in range(len(path)):
        path[i]=int(path[i])
    return path, dist, num_visited
    raise NotImplementedError("To be implemented")
    # End your code (Part 4)
```

The main idea about astar algorithm is when choosing the minimum is choose the data of (path cost + goal proximity).

test[ ]:the data of goal proximity

distance[ ]:the minimum path cost for each path from start node to it

Part II. Results & Analysis (12%):

- **Please screenshot the results. For instance,**

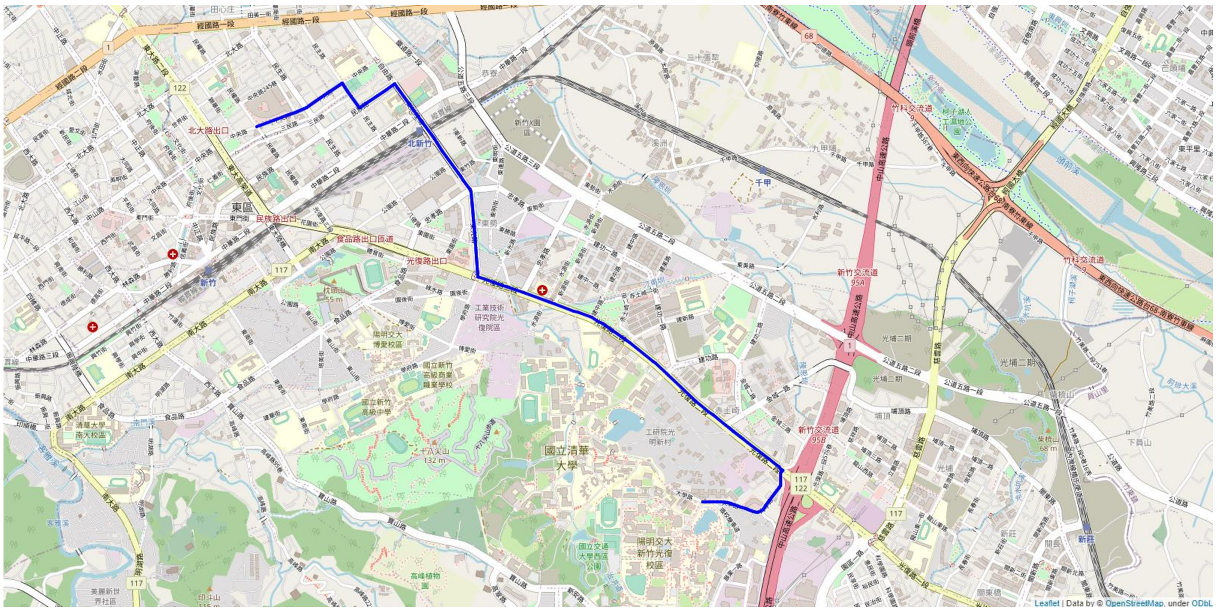
  Test1: from National Yang Ming Chiao Tung University (ID: 2270143902) to Big City Shopping Mall (ID: 1079387396)

BFS:

The number of nodes in the path found by BFS: 88

Total distance of path found by BFS: 4978.881999999998 m

The number of visited nodes in BFS: 4273

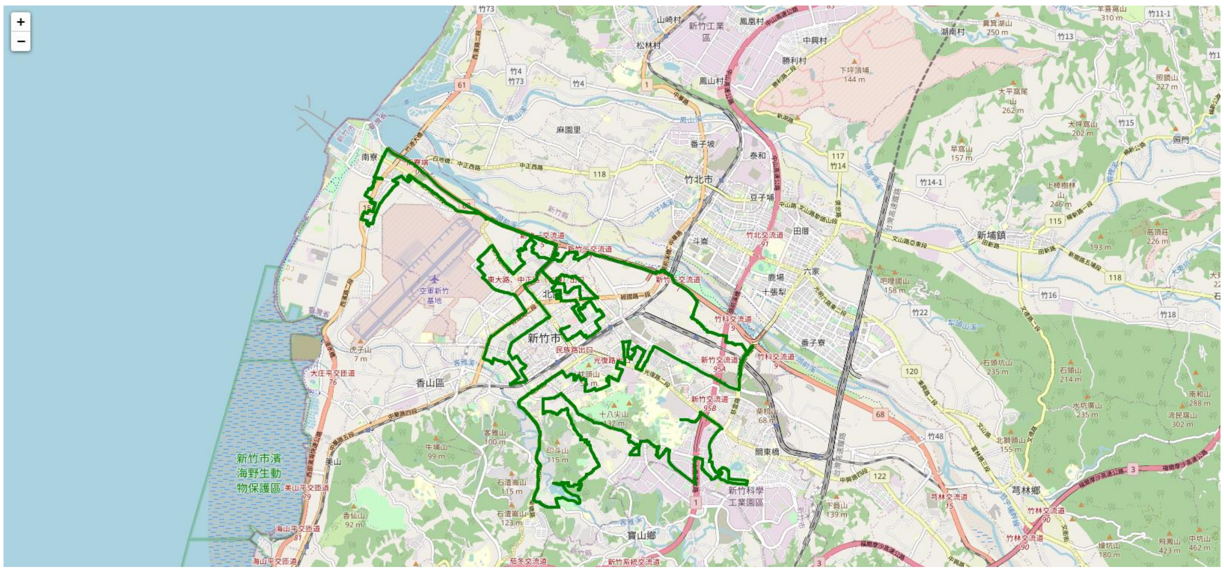


DFS (stack):

The number of nodes in the path found by DFS: 1718

Total distance of path found by DFS: 75504.3150000001 m

The number of visited nodes in DFS: 5235

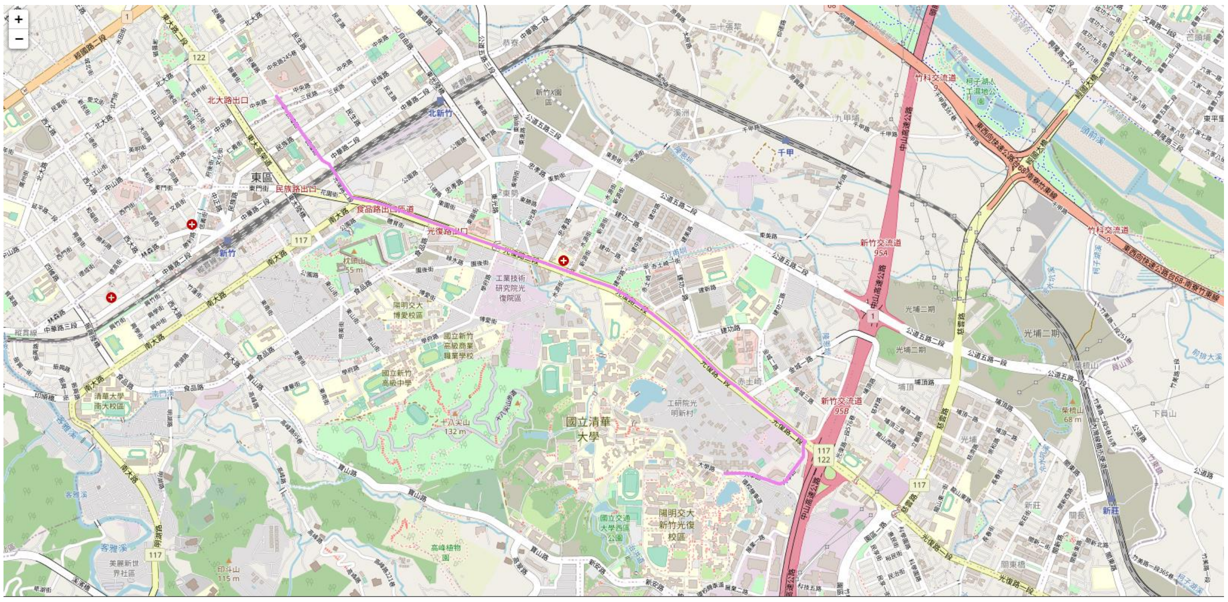


UCS:

The number of nodes in the path found by UCS: 89

Total distance of path found by UCS: 4367.881 m

The number of visited nodes in UCS: 5086

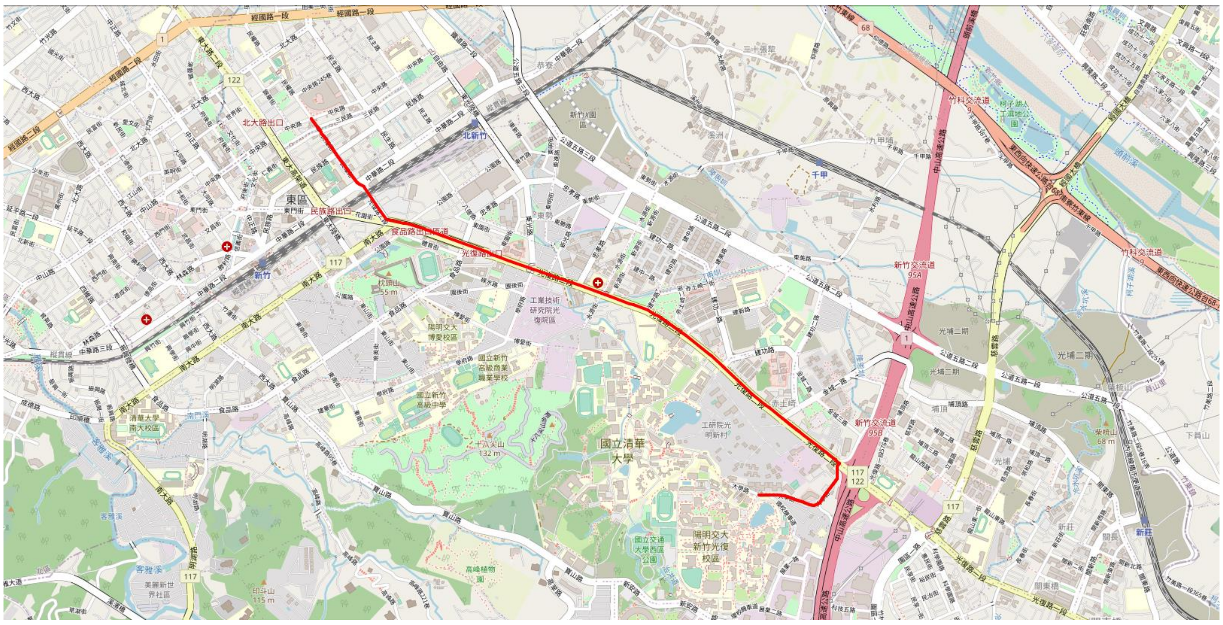


ASTAR:

The number of nodes in the path found by A* search: 89

Total distance of path found by A* search: 4367.881 m

The number of visited nodes in A* search: 261



Part III. Question Answering (12%):

1. **Please describe a problem you encountered and how you solved it.**
   The biggest problem I met in this homework is use correct type to call the variable, for example, when I calculate the total distance of the path, the type I initial store the distance is string, so to do the operation add, notice that we should turn the type to floated.

2. **Besides speed limit and distance, could you please come up with another attribute that is essential for route finding in the real world? Please explain therationale.**
   Another essential attribute in the real world can be the traffic flow in different street. In the real world, especially the streets in the city center, the large amount of traffic flow may cause traffic jam, which cause the time spend increase. To calculate this attribute, we may get the statistic data that what time the car usually stuck in traffic, for example commute time, then we can adjust the average speed for the road.

3. **As mentioned in the introduction, a navigation system involves mapping, localization, and route finding. Please suggest possible solutions for mappingand localization components?**

   the process of concurrently building up a map of the environment and using this map to obtain improved estimates of the location of the vehicle. We can define the interfaces for the components, provide implementations, and connect them in a data flow or dependency graph. The final implementation presented supports, in theory, particle filter localization and can benefit from automated parallelization.

4. **The estimated time of arrival (ETA) is one of the features of Uber Eats. To provide accurate estimates for users, Uber Eats needs to dynamically update based on other attributes. Please define a dynamic heuristic function for ETA. Please explain the rationale of your design.**

   We propose a restricted dynamic programming heuristic for constructing the vehicle routes, and an efficient heuristic for optimizing the vehicle’s departure times for each (partial) vehicle route, such that the complete solution algorithm runs in polynomial time. we apply it to the extended node set concerned with the giant-tour representation of vehicle routing solutions. When a state is expanded with a vehicle route-end node, then the associated vehicle route is completed. In the next stage, we consider the route-start node of the successive vehicle as the only feasible expansion, such that a new vehicle route is started. In order to obtain feasible vehicle routes, we add state dimensions that indicate. When we expand a state, we perform feasibility checks to ensure that vehicle capacities are not exceeded.