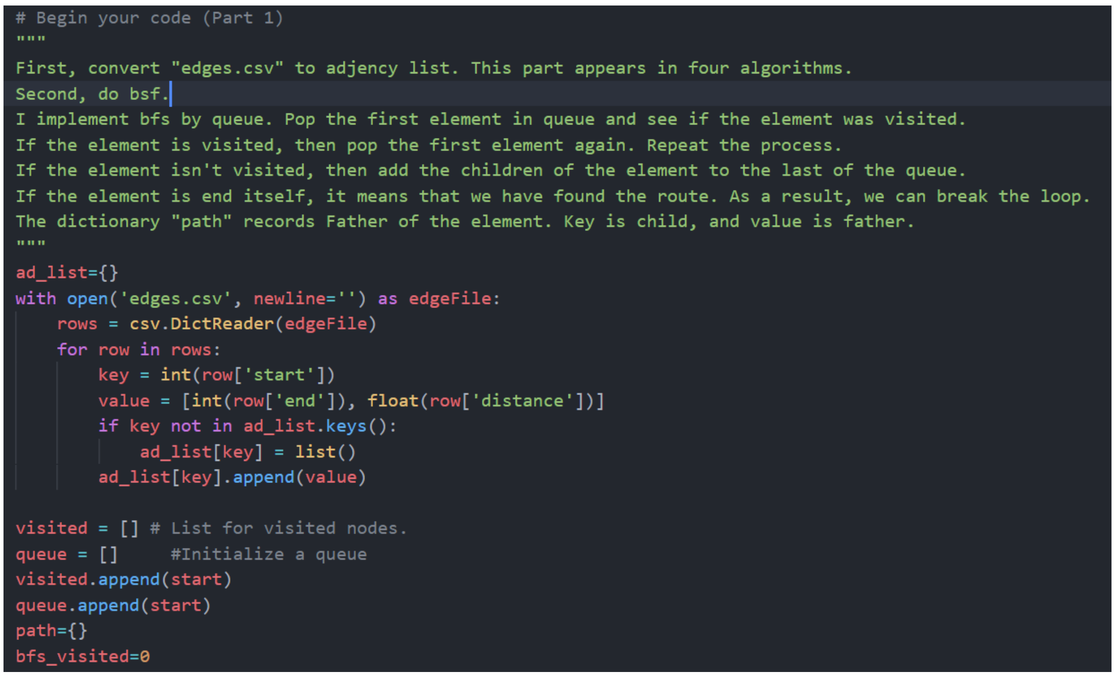
Homework 2: Route Finding Report Template

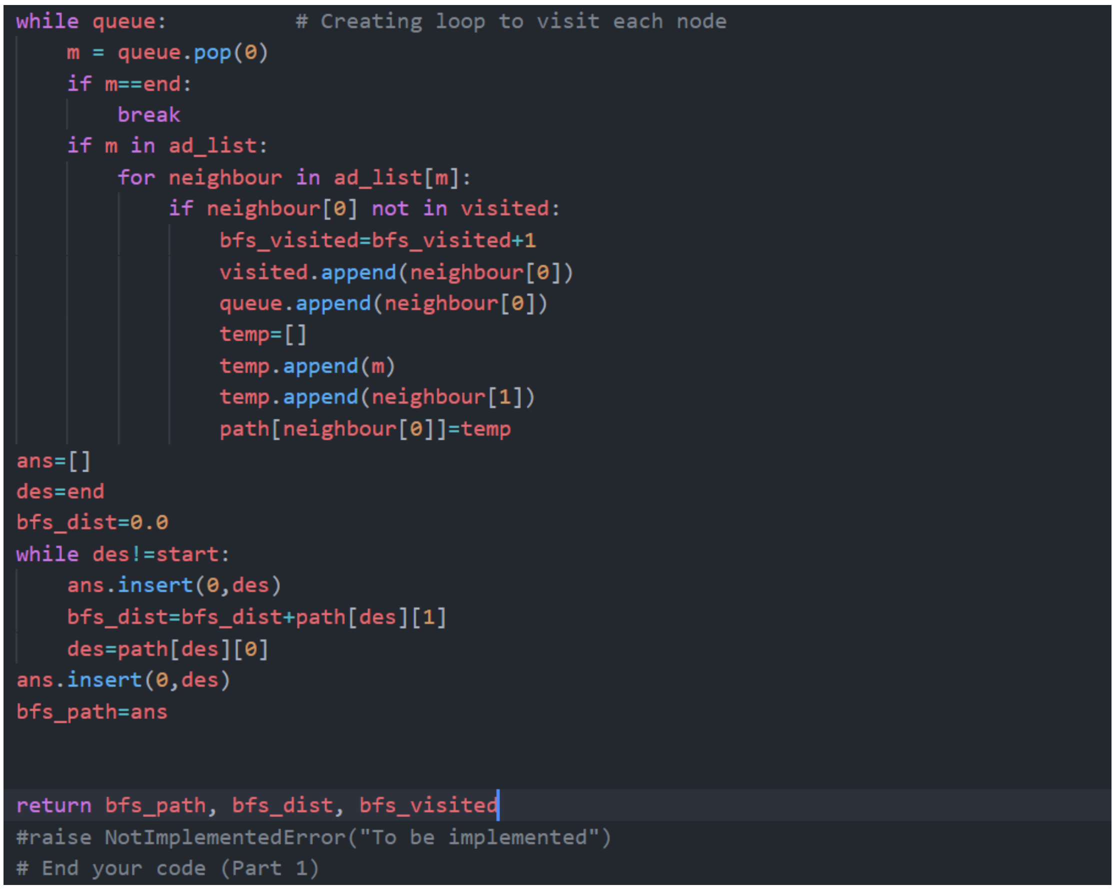
109550136 邱弘竣

Part I. Implementation (6%):



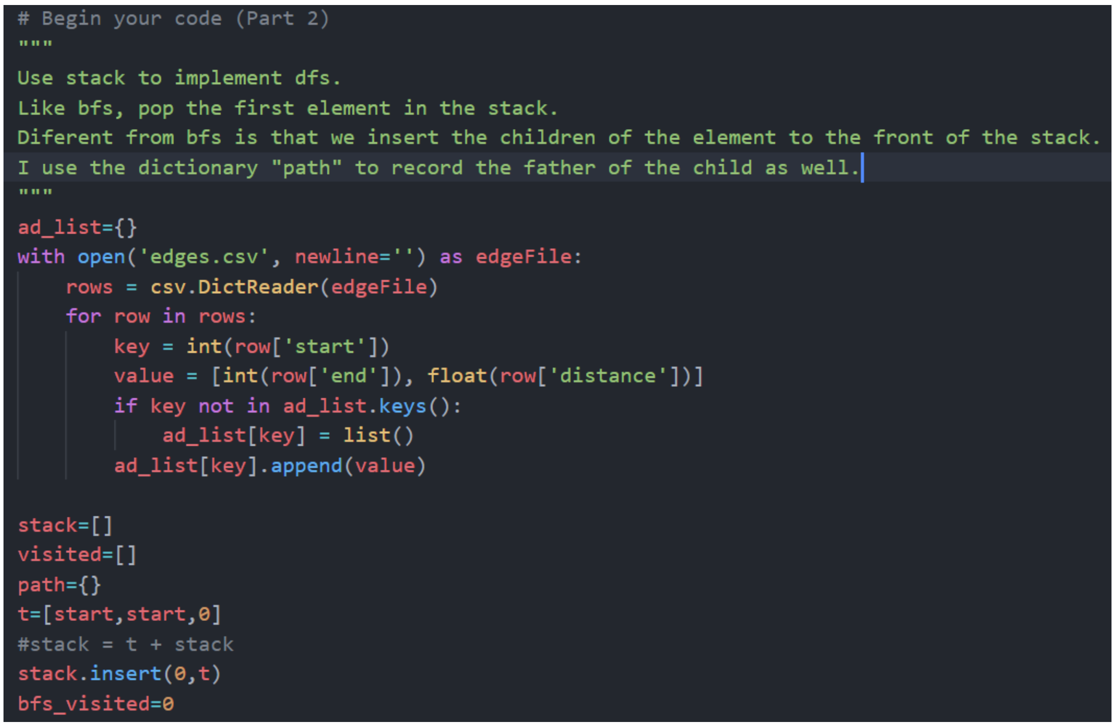
```python
# Begin your code (Part 1)
"""
First, convert "edges.csv" to adjency list. This part appears in four algorithms.
Second, do bsf.
I implement bfs by queue. Pop the first element in queue and see if the element was visited.
If the element is visited, then pop the first element again. Repeat the process.
If the element isn't visited, then add the children of the element to the last of the queue.
If the element is end itself, it means that we have found the route. As a result, we can break the loop.
The dictionary "path" records Father of the element. Key is child, and value is father.
"""
ad_list={}
with open('edges.csv', newline='') as edgeFile:
    rows = csv.DictReader(edgeFile)
    for row in rows:
        key = int(row['start'])
        value = [int(row['end']), float(row['distance'])]
        if key not in ad_list.keys():
            ad_list[key] = list()
        ad_list[key].append(value)

visited = [] # List for visited nodes.
queue = []     #Initialize a queue
visited.append(start)
queue.append(start)
path={}
bfs_visited=0
```



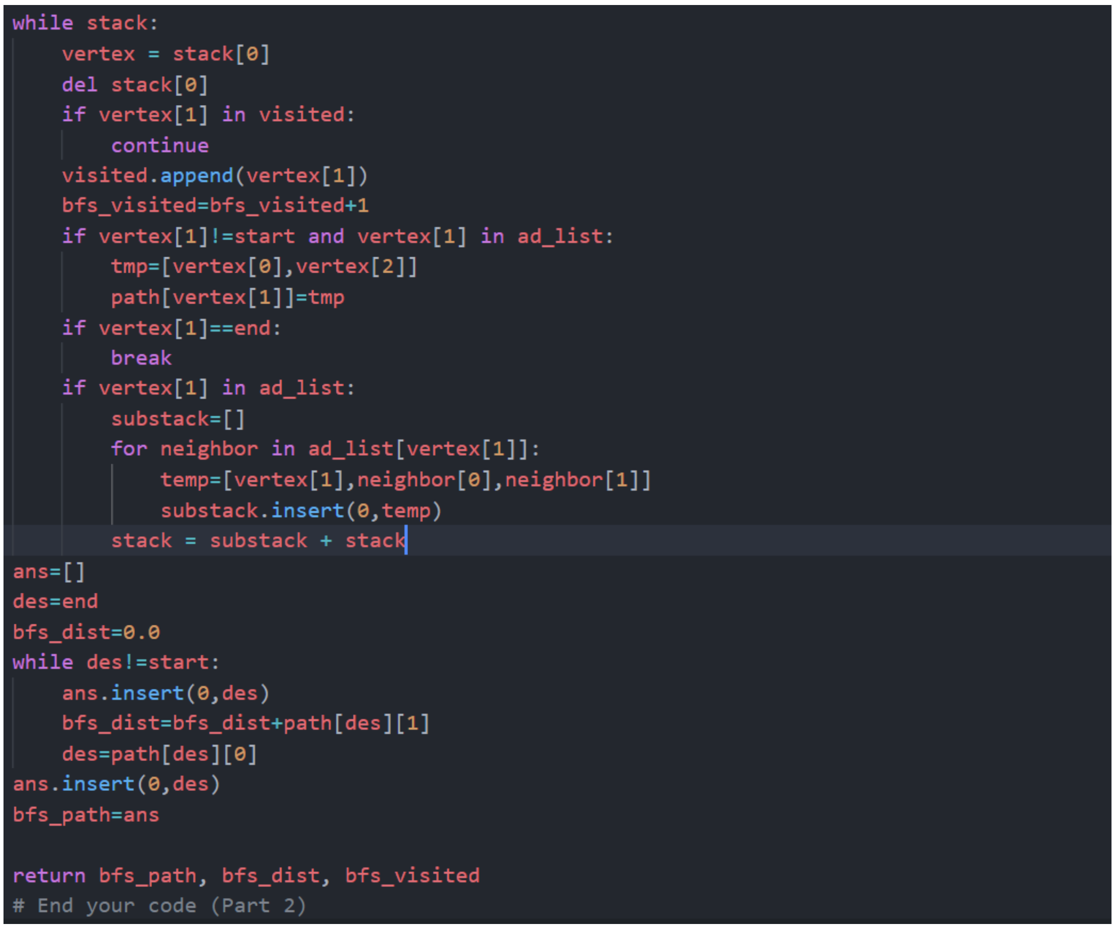
```python
while queue:          # Creating loop to visit each node
    m = queue.pop(0)
    if m==end:
        break
    if m in ad_list:
        for neighbour in ad_list[m]:
            if neighbour[0] not in visited:
                bfs_visited=bfs_visited+1
                visited.append(neighbour[0])
                queue.append(neighbour[0])
                temp=[]
                temp.append(m)
                temp.append(neighbour[1])
                path[neighbour[0]]=temp
ans=[]
des=end
bfs_dist=0.0
while des!=start:
    ans.insert(0,des)
    bfs_dist=bfs_dist+path[des][1]
    des=path[des][0]
ans.insert(0,des)
bfs_path=ans


return bfs_path, bfs_dist, bfs_visited
#raise NotImplementedError("To be implemented")
# End your code (Part 1)
```



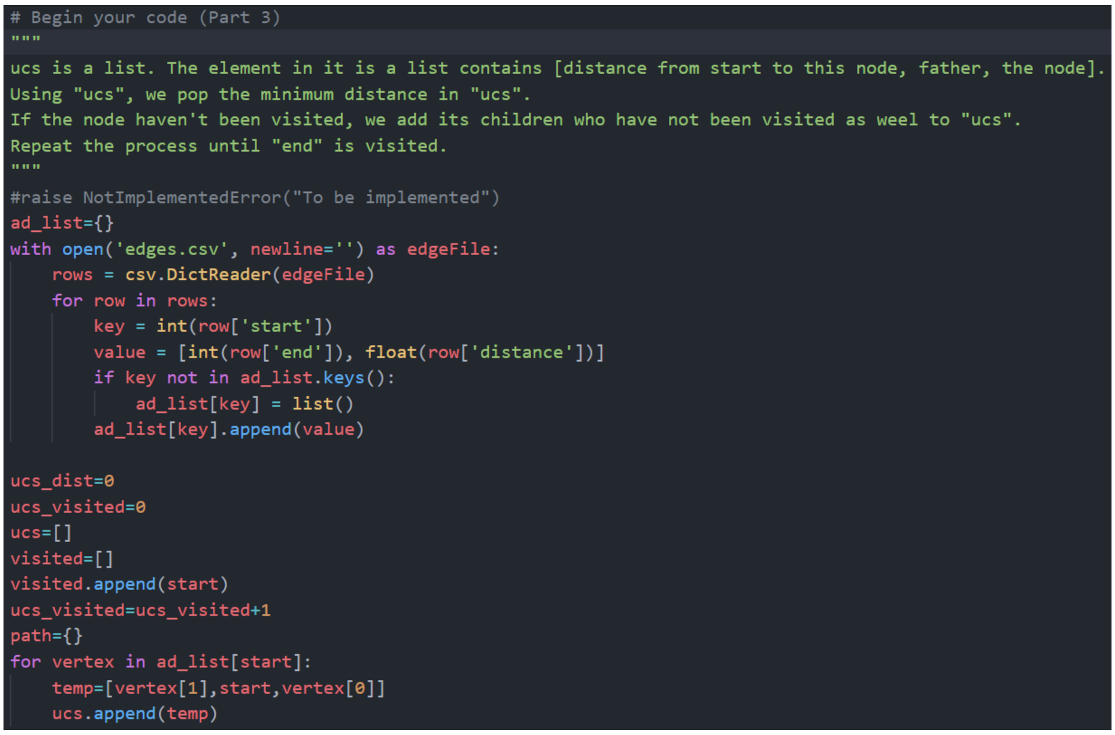
```python
# Begin your code (Part 2)
"""
Use stack to implement dfs.
Like bfs, pop the first element in the stack.
Diferent from bfs is that we insert the children of the element to the front of the stack.
I use the dictionary "path" to record the father of the child as well.
"""
ad_list={}
with open('edges.csv', newline='') as edgeFile:
    rows = csv.DictReader(edgeFile)
    for row in rows:
        key = int(row['start'])
        value = [int(row['end']), float(row['distance'])]
        if key not in ad_list.keys():
            ad_list[key] = list()
        ad_list[key].append(value)

stack=[]
visited=[]
path={}
t=[start,start,0]
#stack = t + stack
stack.insert(0,t)
bfs_visited=0
```



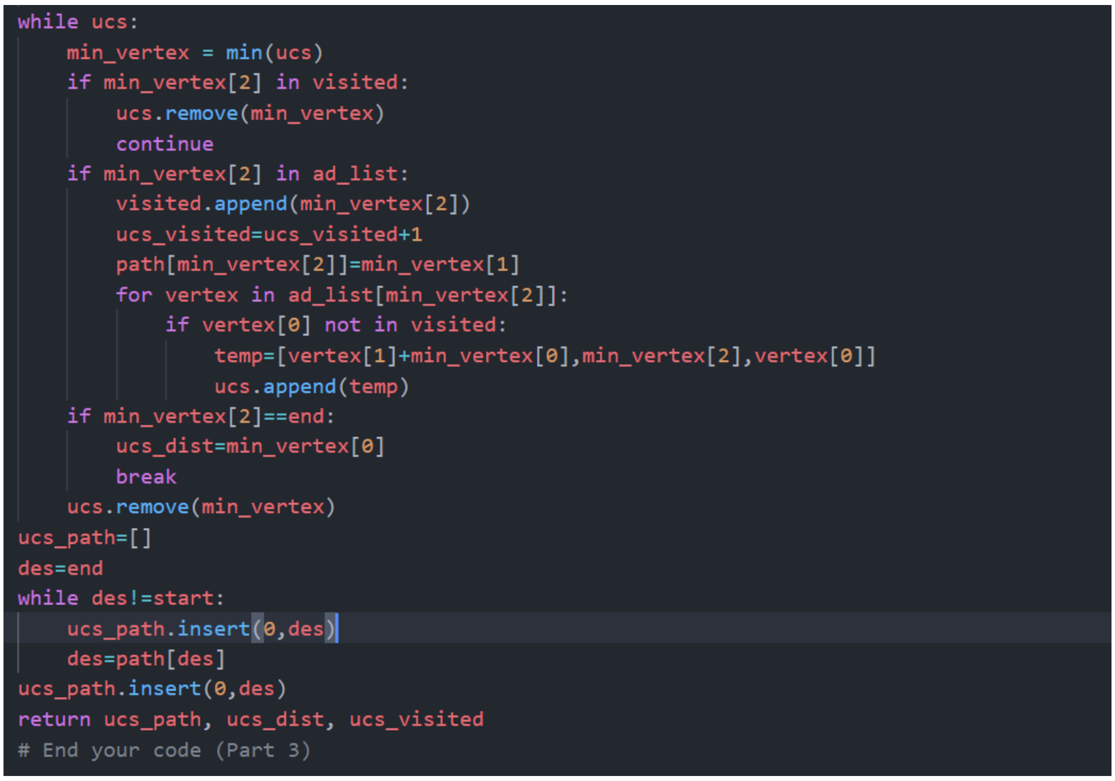
```python
while stack:
    vertex = stack[0]
    del stack[0]
    if vertex[1] in visited:
        continue
    visited.append(vertex[1])
    bfs_visited=bfs_visited+1
    if vertex[1]!=start and vertex[1] in ad_list:
        tmp=[vertex[0],vertex[2]]
        path[vertex[1]]=tmp
    if vertex[1]==end:
        break
    if vertex[1] in ad_list:
        substack=[]
        for neighbor in ad_list[vertex[1]]:
            temp=[vertex[1],neighbor[0],neighbor[1]]
            substack.insert(0,temp)
        stack = substack + stack
ans=[]
des=end
bfs_dist=0.0
while des!=start:
    ans.insert(0,des)
    bfs_dist=bfs_dist+path[des][1]
    des=path[des][0]
ans.insert(0,des)
bfs_path=ans

return bfs_path, bfs_dist, bfs_visited
# End your code (Part 2)
```

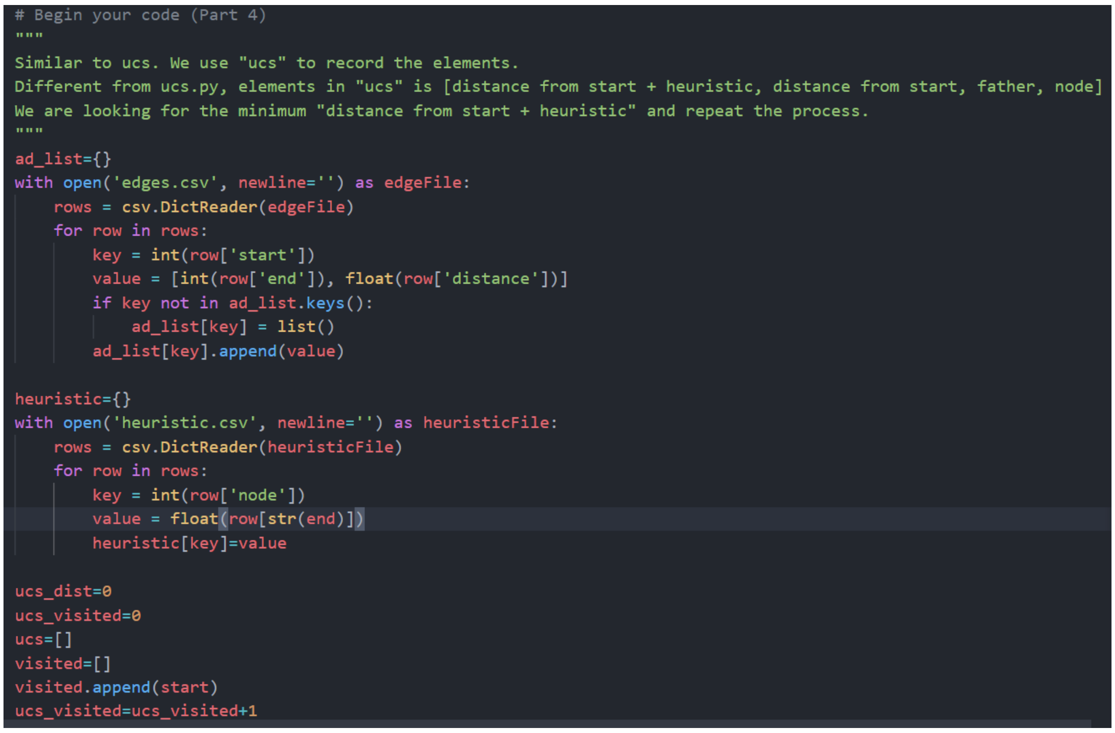


```python
# Begin your code (Part 3)
"""
ucs is a list. The element in it is a list contains [distance from start to this node, father, the node].
Using "ucs", we pop the minimum distance in "ucs".
If the node haven't been visited, we add its children who have not been visited as weel to "ucs".
Repeat the process until "end" is visited.
"""
#raise NotImplementedError("To be implemented")
ad_list={}
with open('edges.csv', newline='') as edgeFile:
    rows = csv.DictReader(edgeFile)
    for row in rows:
        key = int(row['start'])
        value = [int(row['end']), float(row['distance'])]
        if key not in ad_list.keys():
            ad_list[key] = list()
        ad_list[key].append(value)

ucs_dist=0
ucs_visited=0
ucs=[]
visited=[]
visited.append(start)
ucs_visited=ucs_visited+1
path={}
for vertex in ad_list[start]:
    temp=[vertex[1],start,vertex[0]]
    ucs.append(temp)
```



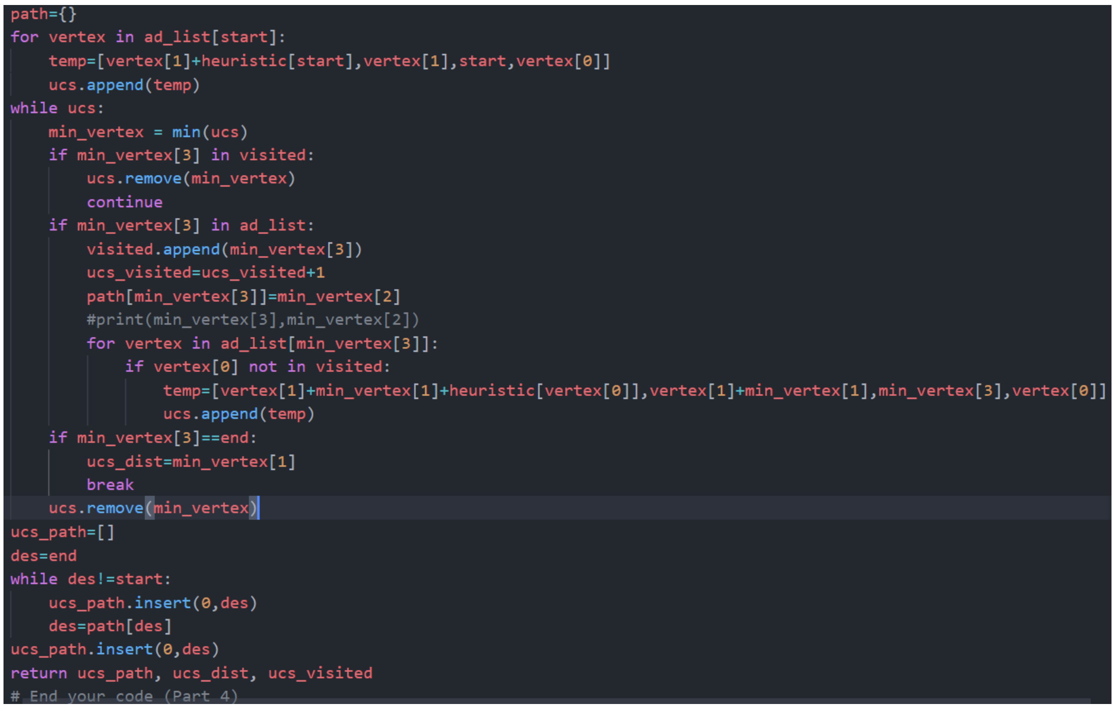
```python
while ucs:
    min_vertex = min(ucs)
    if min_vertex[2] in visited:
        ucs.remove(min_vertex)
        continue
    if min_vertex[2] in ad_list:
        visited.append(min_vertex[2])
        ucs_visited=ucs_visited+1
        path[min_vertex[2]]=min_vertex[1]
        for vertex in ad_list[min_vertex[2]]:
            if vertex[0] not in visited:
                temp=[vertex[1]+min_vertex[0],min_vertex[2],vertex[0]]
                ucs.append(temp)
    if min_vertex[2]==end:
        ucs_dist=min_vertex[0]
        break
    ucs.remove(min_vertex)
ucs_path=[]
des=end
while des!=start:
    ucs_path.insert(0,des)
    des=path[des]
ucs_path.insert(0,des)
return ucs_path, ucs_dist, ucs_visited
# End your code (Part 3)
```



```python
# Begin your code (Part 4)
"""
Similar to ucs. We use "ucs" to record the elements.
Different from ucs.py, elements in "ucs" is [distance from start + heuristic, distance from start, father, node]
We are looking for the minimum "distance from start + heuristic" and repeat the process.
"""
ad_list={}
with open('edges.csv', newline='') as edgeFile:
    rows = csv.DictReader(edgeFile)
    for row in rows:
        key = int(row['start'])
        value = [int(row['end']), float(row['distance'])]
        if key not in ad_list.keys():
            ad_list[key] = list()
        ad_list[key].append(value)

heuristic={}
with open('heuristic.csv', newline='') as heuristicFile:
    rows = csv.DictReader(heuristicFile)
    for row in rows:
        key = int(row['node'])
        value = float(row[str(end)])
        heuristic[key]=value

ucs_dist=0
ucs_visited=0
ucs=[]
visited=[]
visited.append(start)
ucs_visited=ucs_visited+1
```

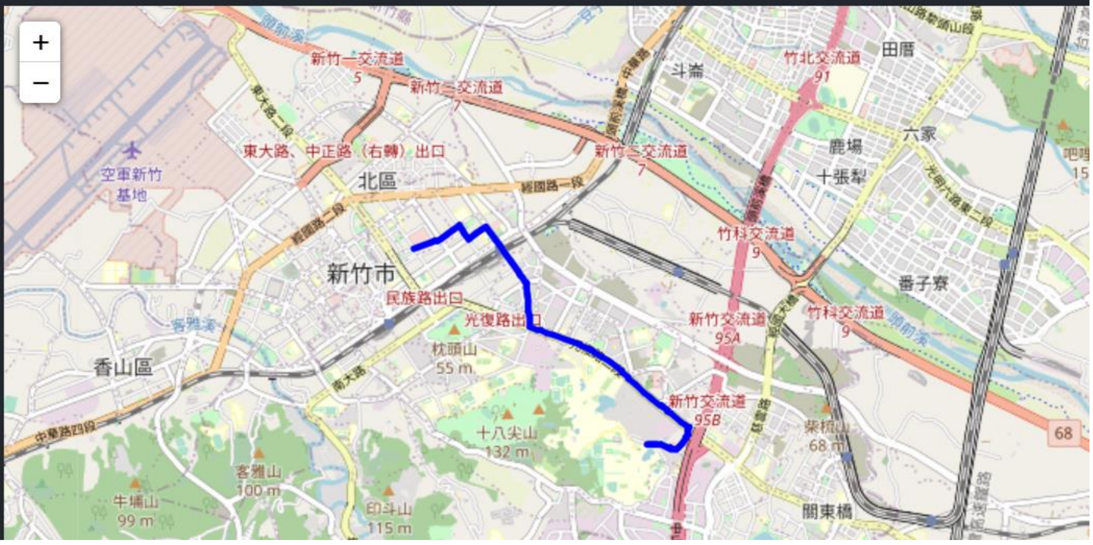


```python
path={}
for vertex in ad_list[start]:
    temp=[vertex[1]+heuristic[start],vertex[1],start,vertex[0]]
    ucs.append(temp)
while ucs:
    min_vertex = min(ucs)
    if min_vertex[3] in visited:
        ucs.remove(min_vertex)
        continue
    if min_vertex[3] in ad_list:
        visited.append(min_vertex[3])
        ucs_visited=ucs_visited+1
        path[min_vertex[3]]=min_vertex[2]
        #print(min_vertex[3],min_vertex[2])
        for vertex in ad_list[min_vertex[3]]:
            if vertex[0] not in visited:
                temp=[vertex[1]+min_vertex[1]+heuristic[vertex[0]],vertex[1]+min_vertex[1],min_vertex[3],vertex[0]]
                ucs.append(temp)
    if min_vertex[3]==end:
        ucs_dist=min_vertex[1]
        break
    ucs.remove(min_vertex)
ucs_path=[]
des=end
while des!=start:
    ucs_path.insert(0,des)
    des=path[des]
ucs_path.insert(0,des)
return ucs_path, ucs_dist, ucs_visited
# End your code (Part 4)
```

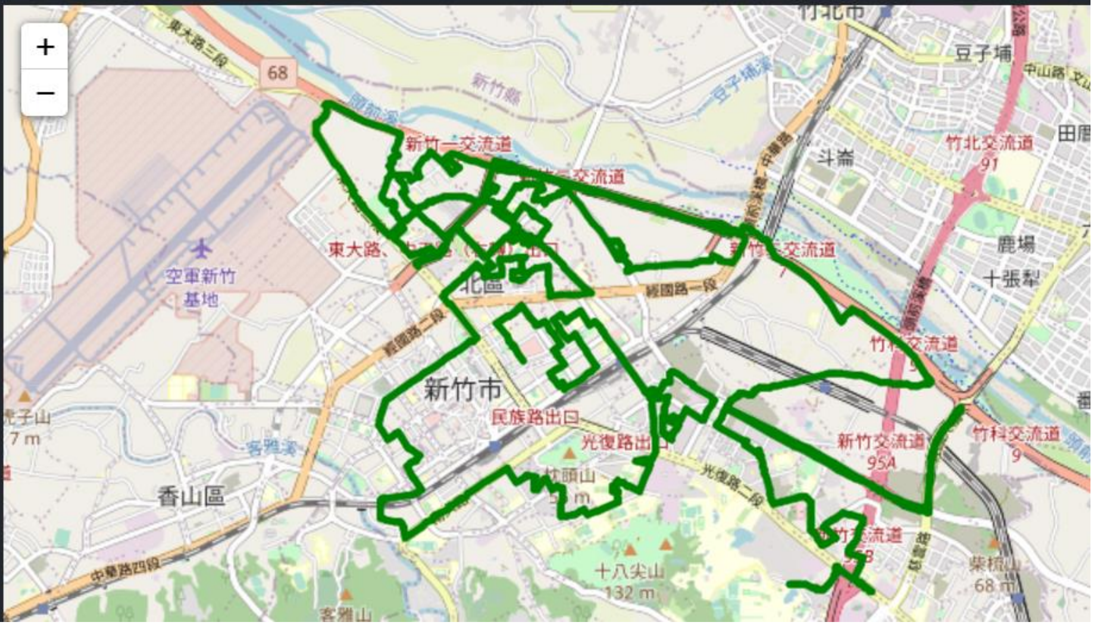
Part II. Results & Analysis (12%):

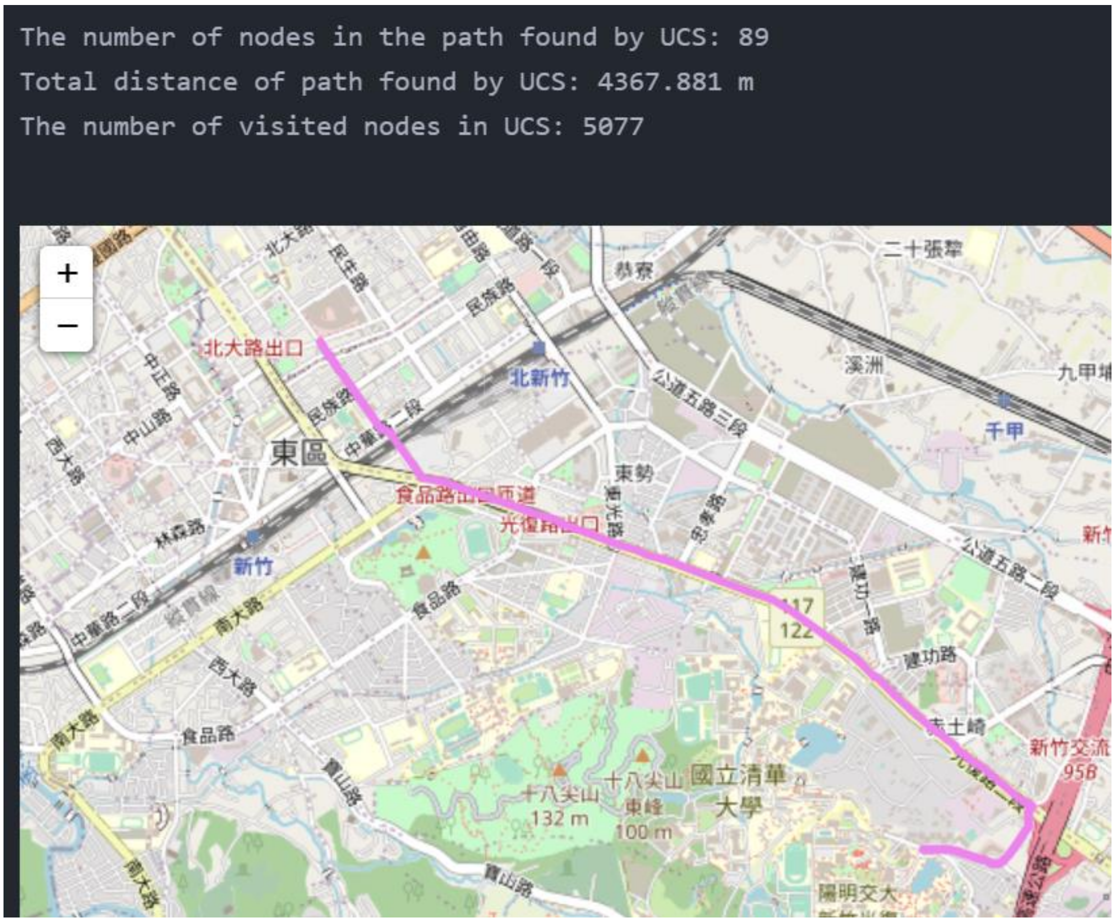
Test1

```
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: 4402
```

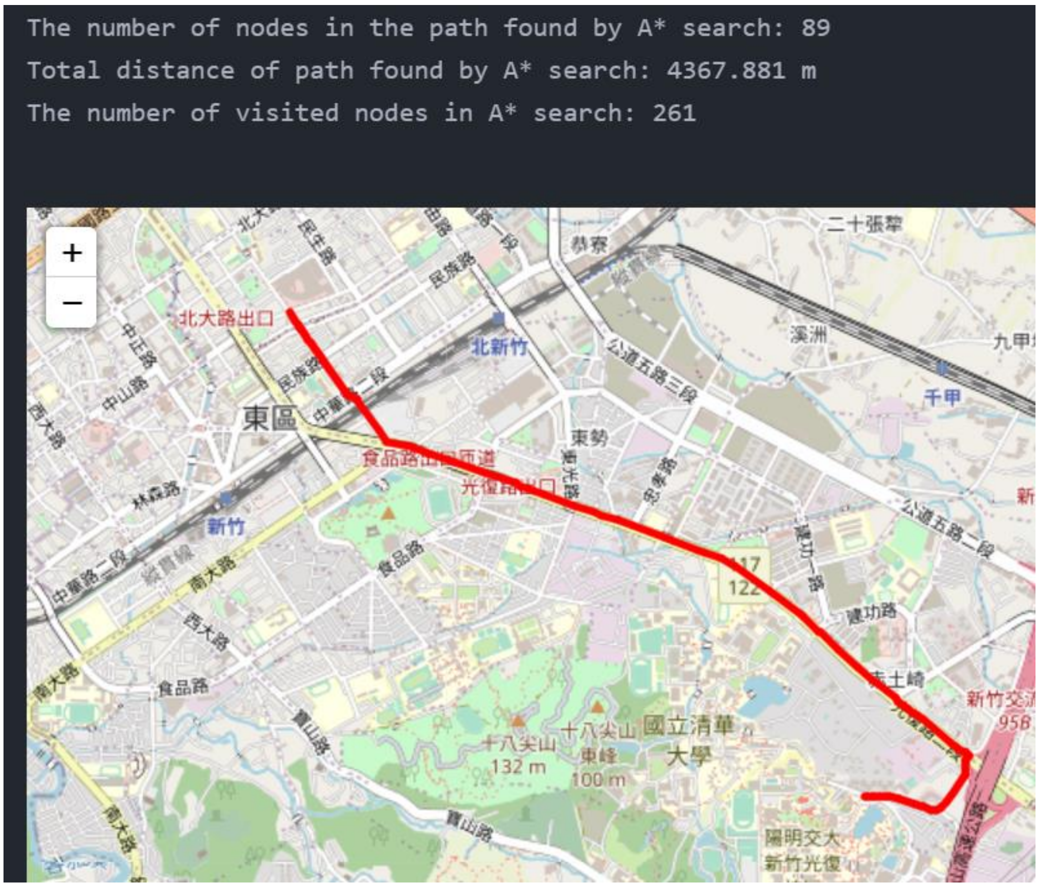


```
The number of nodes in the path found by DFS: 1232
Total distance of path found by DFS: 57208.987 m
The number of visited nodes in DFS: 4211
```

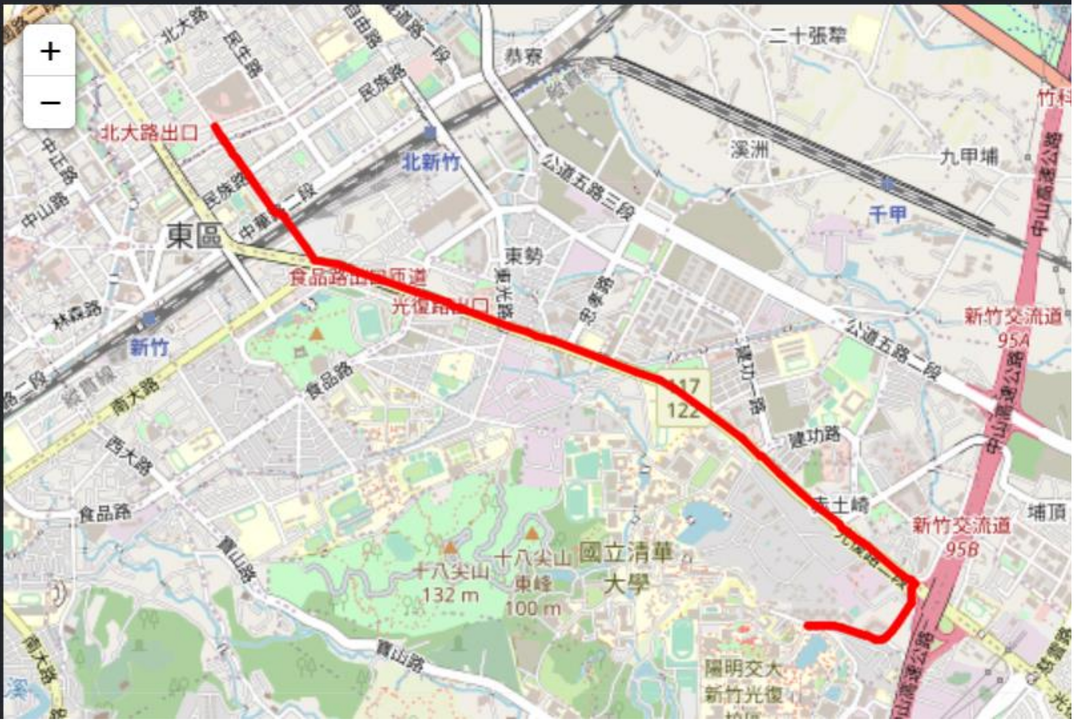




```
The number of nodes in the path found by UCS: 89
Total distance of path found by UCS: 4367.881 m
The number of visited nodes in UCS: 5077
```

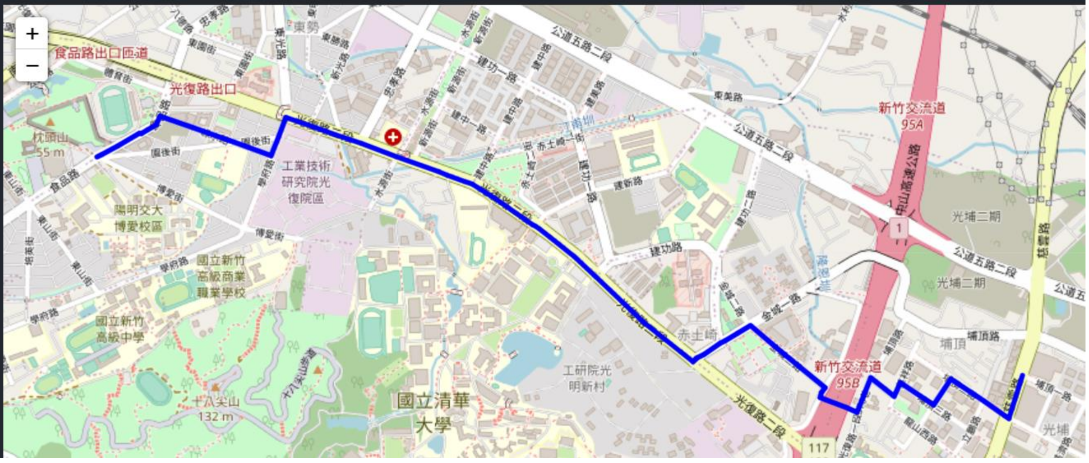


```
The number of nodes in the path found by A* search: 89
Total distance of path found by A* search: 4367.881 m
The number of visited nodes in A* search: 261
The number of nodes in the path found by A* search: 89
Total second of path found by A* search: 320.87823163083164 s
The number of visited nodes in A* search: 814
```

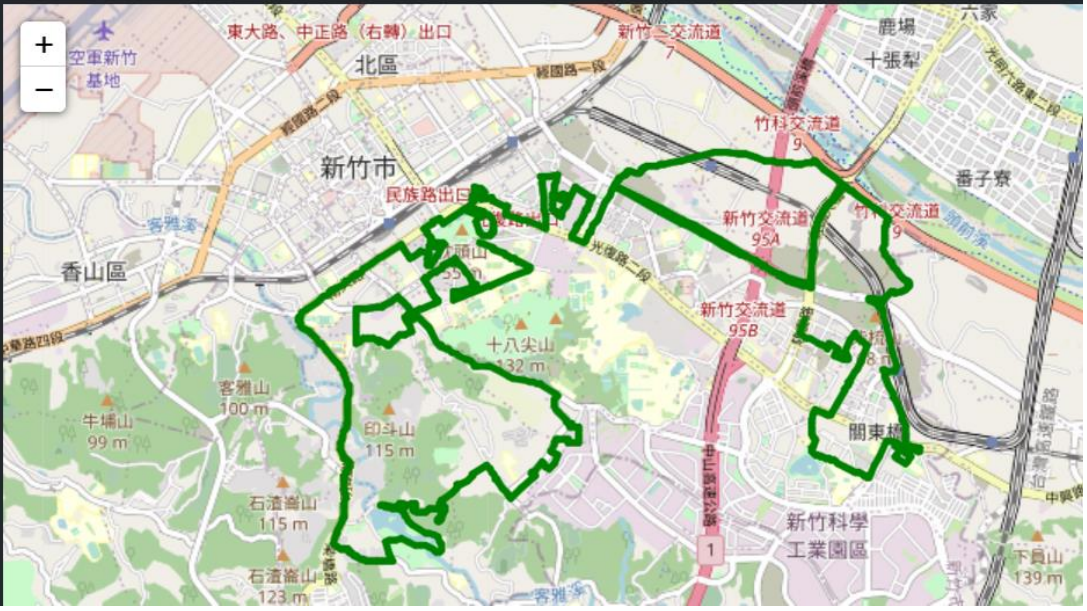


Test2

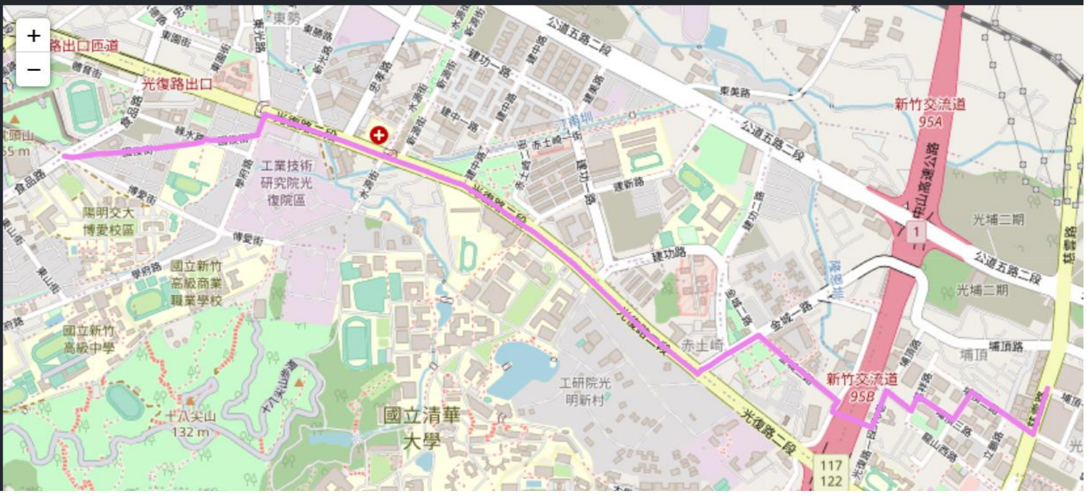
```
The number of nodes in the path found by BFS: 60
Total distance of path found by BFS: 4215.521000000001 m
The number of visited nodes in BFS: 4751
```

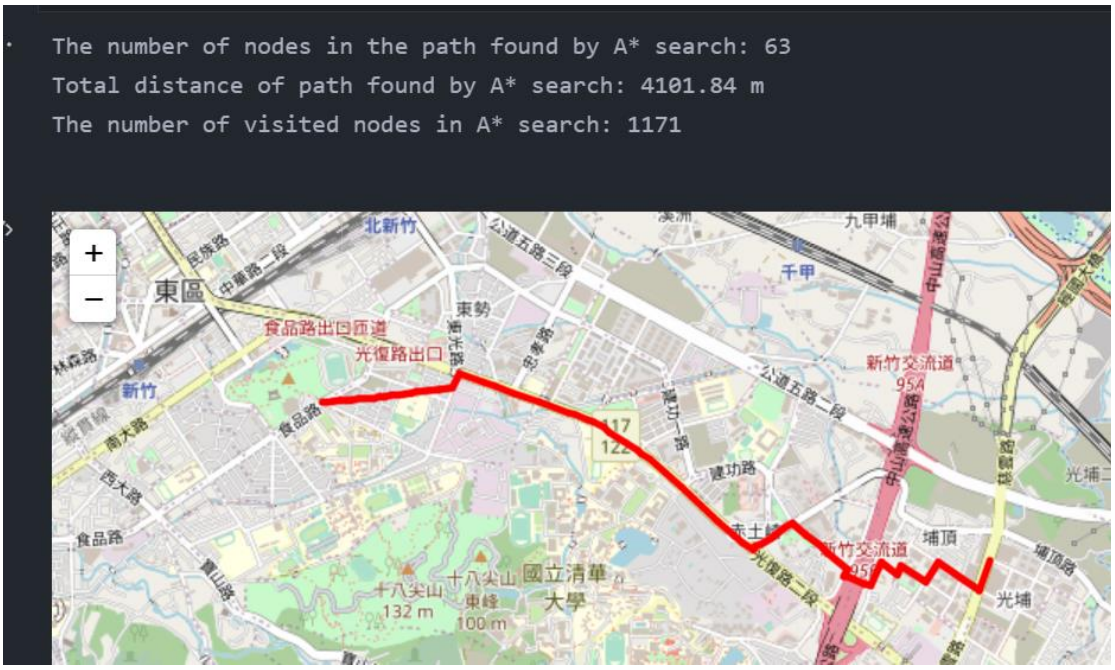


```
The number of nodes in the path found by DFS: 998
Total distance of path found by DFS: 41094.657999999916 m
The number of visited nodes in DFS: 8031
```

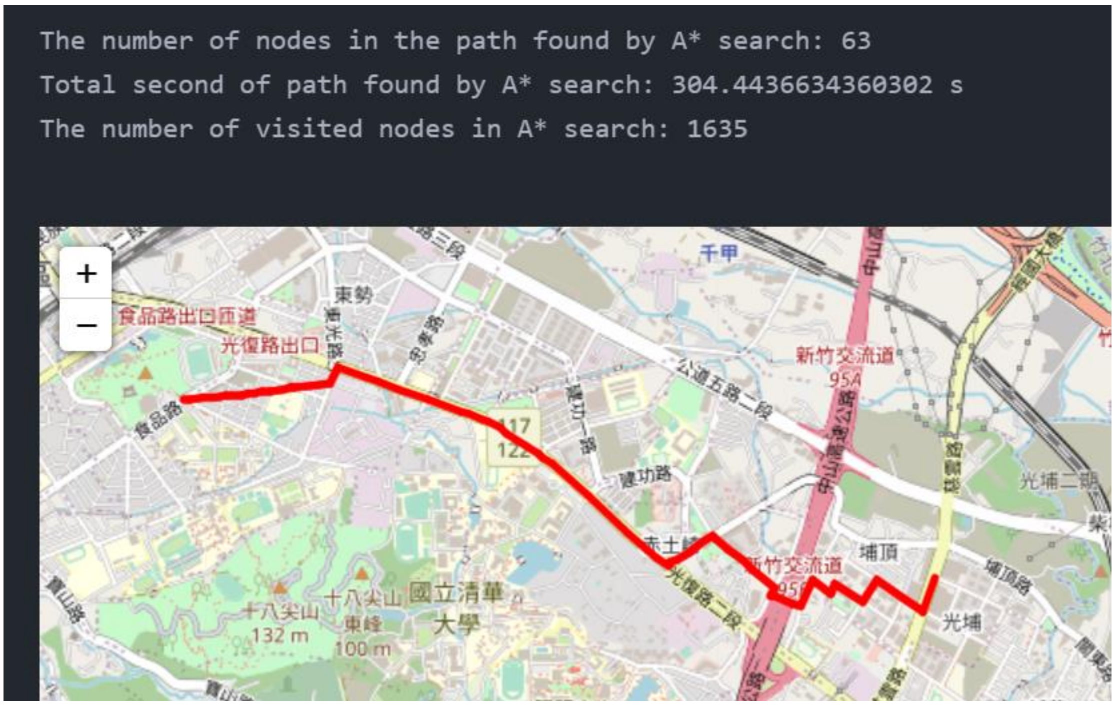


```
The number of nodes in the path found by UCS: 63
Total distance of path found by UCS: 4101.84 m
The number of visited nodes in UCS: 7207
```





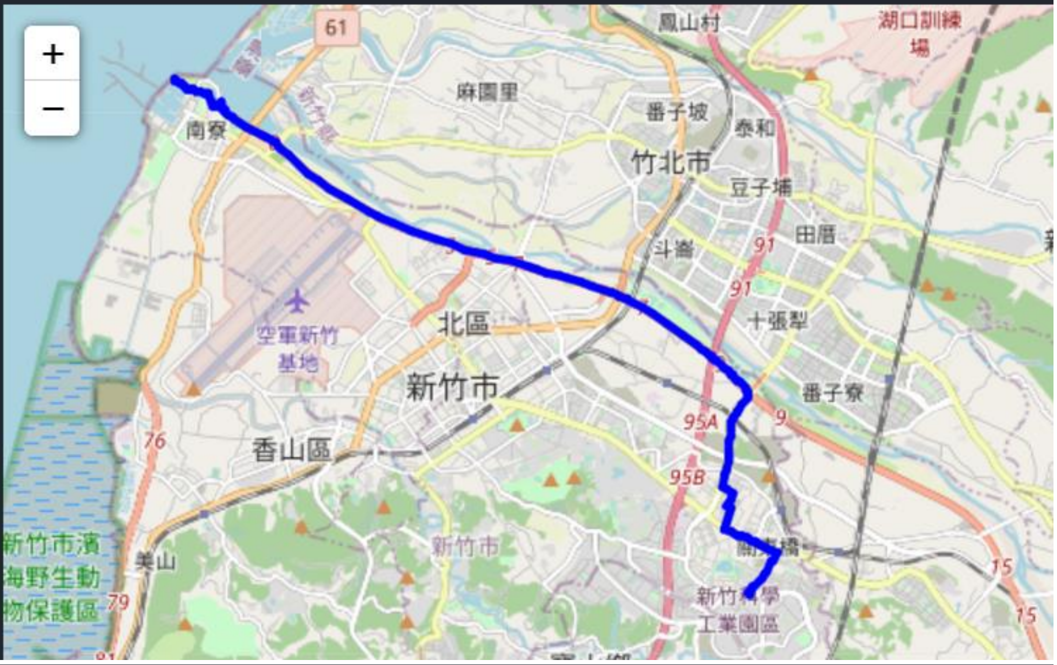
```
The number of nodes in the path found by A* search: 63
Total distance of path found by A* search: 4101.84 m
The number of visited nodes in A* search: 1171
```



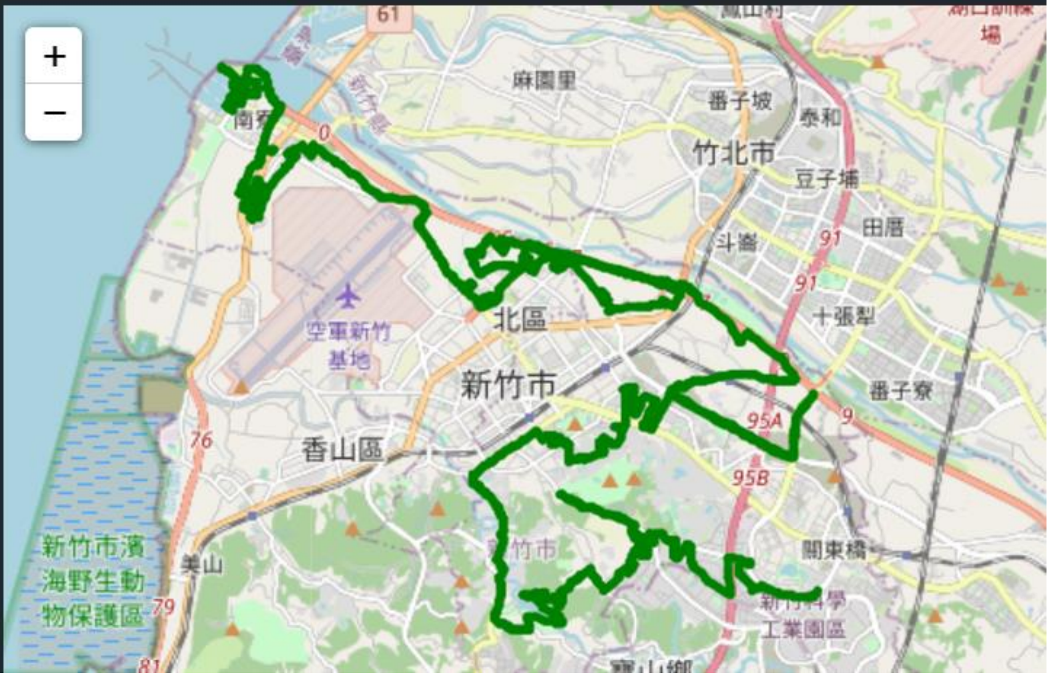
```
The number of nodes in the path found by A* search: 63
Total second of path found by A* search: 304.4436634360302 s
The number of visited nodes in A* search: 1635
```

Test3

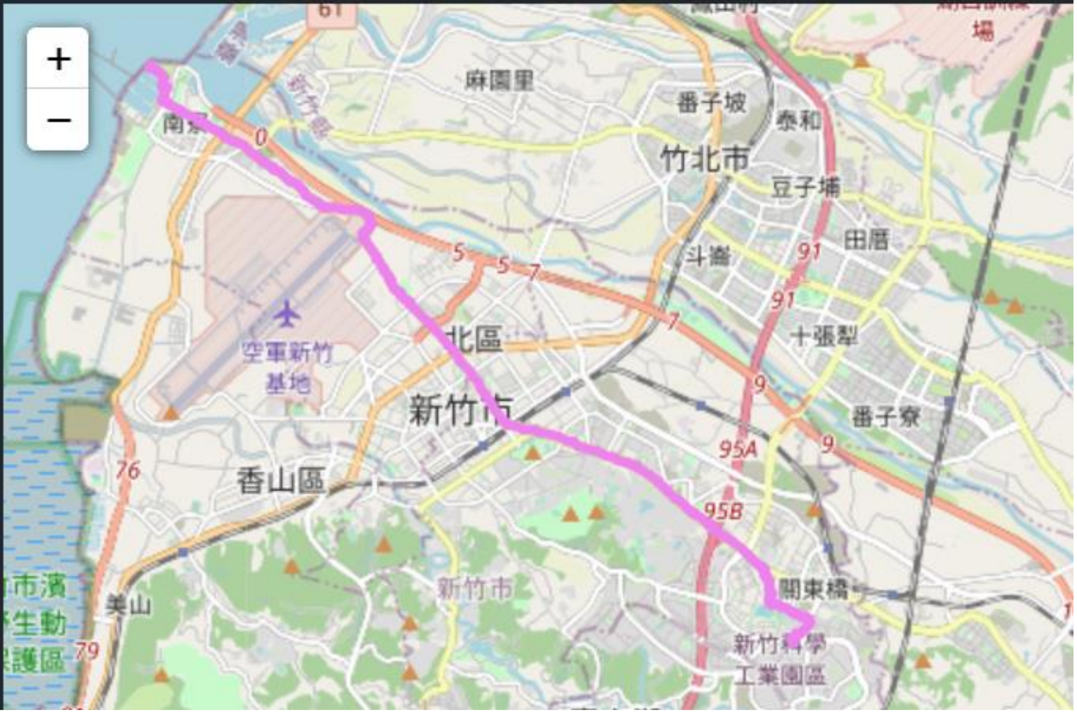
```
The number of nodes in the path found by BFS: 183
Total distance of path found by BFS: 15442.394999999995 m
The number of visited nodes in BFS: 11265
```



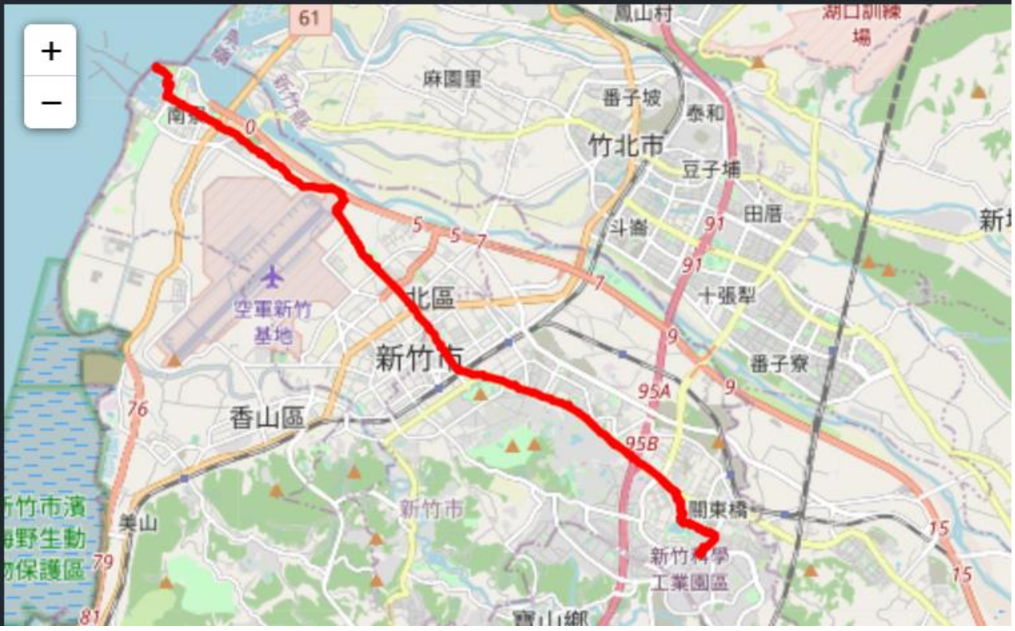
```
The number of nodes in the path found by DFS: 1521
Total distance of path found by DFS: 64821.60399999999 m
The number of visited nodes in DFS: 3292
```

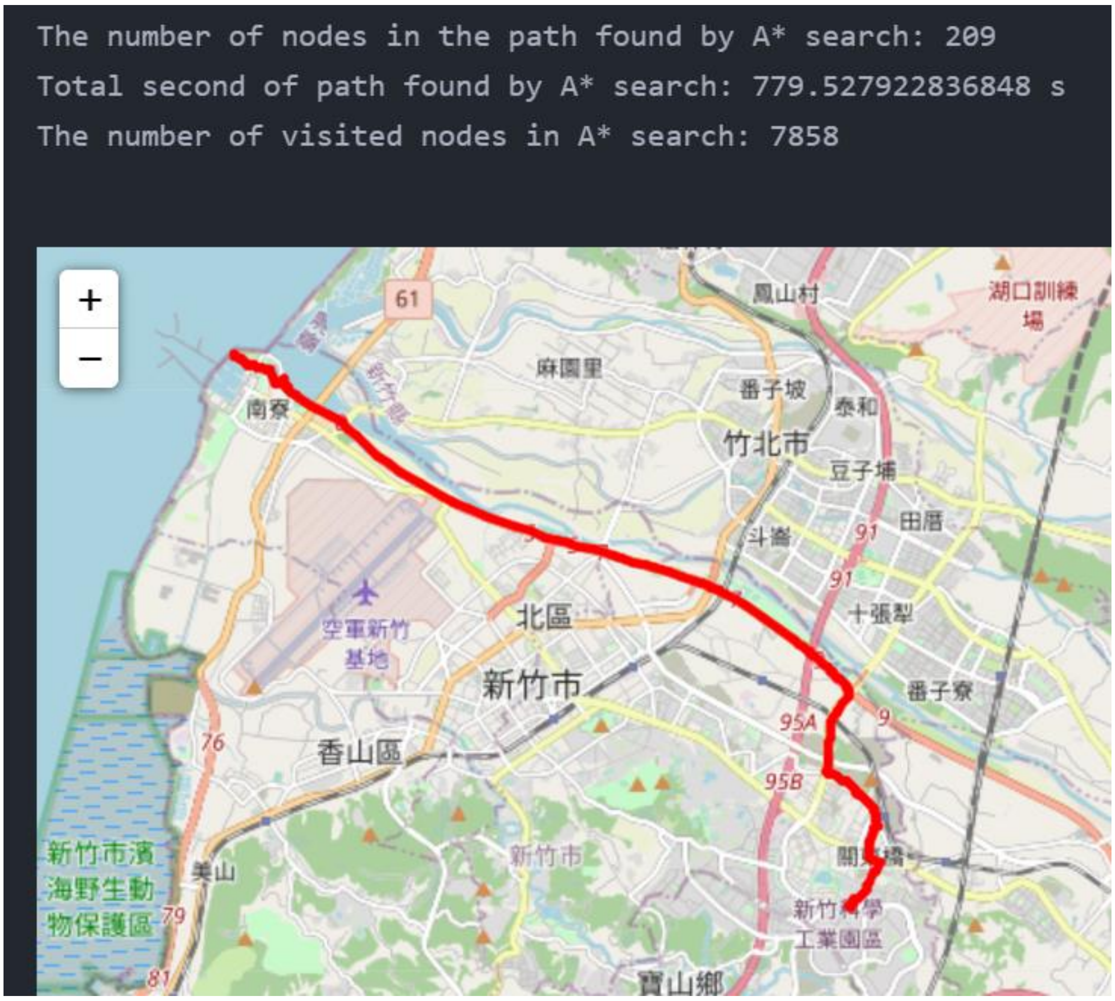


```
The number of nodes in the path found by UCS: 288
Total distance of path found by UCS: 14212.412999999997 m
The number of visited nodes in UCS: 11909
```



```
The number of nodes in the path found by A* search: 288
Total distance of path found by A* search: 14212.412999999997 m
The number of visited nodes in A* search: 7067
```





```
The number of nodes in the path found by A* search: 209
Total second of path found by A* search: 779.527922836848 s
The number of visited nodes in A* search: 7858
```

Bfs：

The goal is to find the fewest node to the destination. However, we didn’t consider the distance of each edge, so the path is not the shortest.

Dfs：

The path found by dfs is the longest because it didn’t consider anything.

Ucs：

The method considers the distance of each edge to find the shortest path, but it didn’t contain other attributes.

A*：

It’s the extension of ucs because it considers the displacement from the present point to the destination.

A* time：

It’s the extension of A* because it considers the speed limit of each edge so that in some cases, traveling on the high speed road may be a better dicision.

Part III. Question Answering (12%):

1. **Please describe a problem you encountered and how you solved it.**
   In the beginning, it’s hard for me to analyze the data in .csv. As a result, I convert the data into adjacency list in python.
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 the rationale.**
   The number of traffic light. If the number of traffic light on the road is high, then it might take much time to arrive at the destination.
3. **As mentioned in the introduction, a navigation system involves mapping, localization, and route finding. Please suggest possible solutions for mapping and localization components?**
   Mapping：using topological mapping. The method is that vital information remains and unnecessary detail has been removed. These maps lack scale, and distance and direction are subject to change and variation, but the relationship between points is maintained.
   Localization：Using GPS. It’s a method that provides geolocation and time information to a GPS receiver anywhere on or near the Earth where there is an unobstructed line of sight to four or more GPS satellites.
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.**
   Using the path length from our location to our destination divides average speed in out future road. Besides, add some time or minus some time depends on how the traffic is jammed at that time. Update the heuristic function every five seconds.