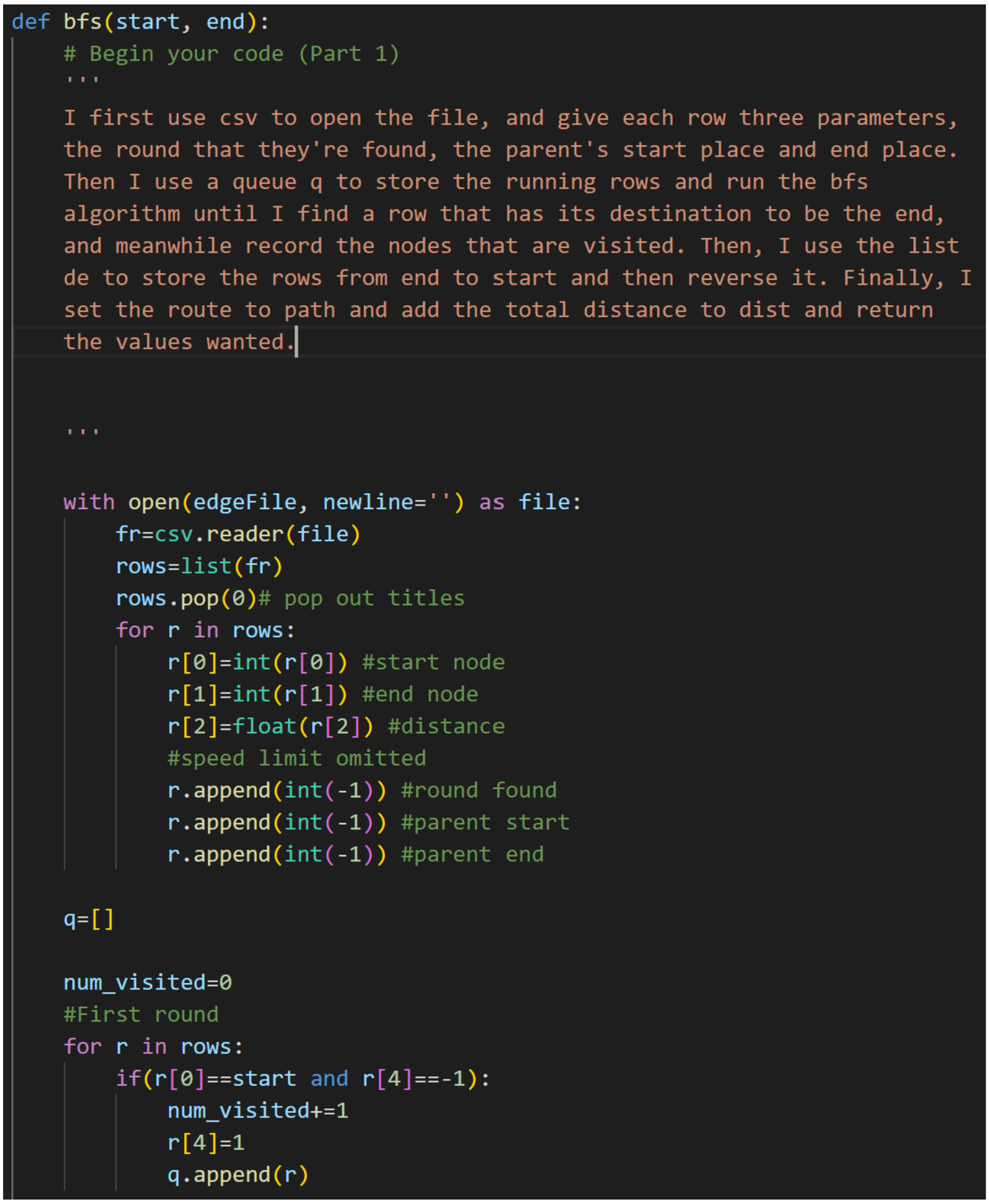
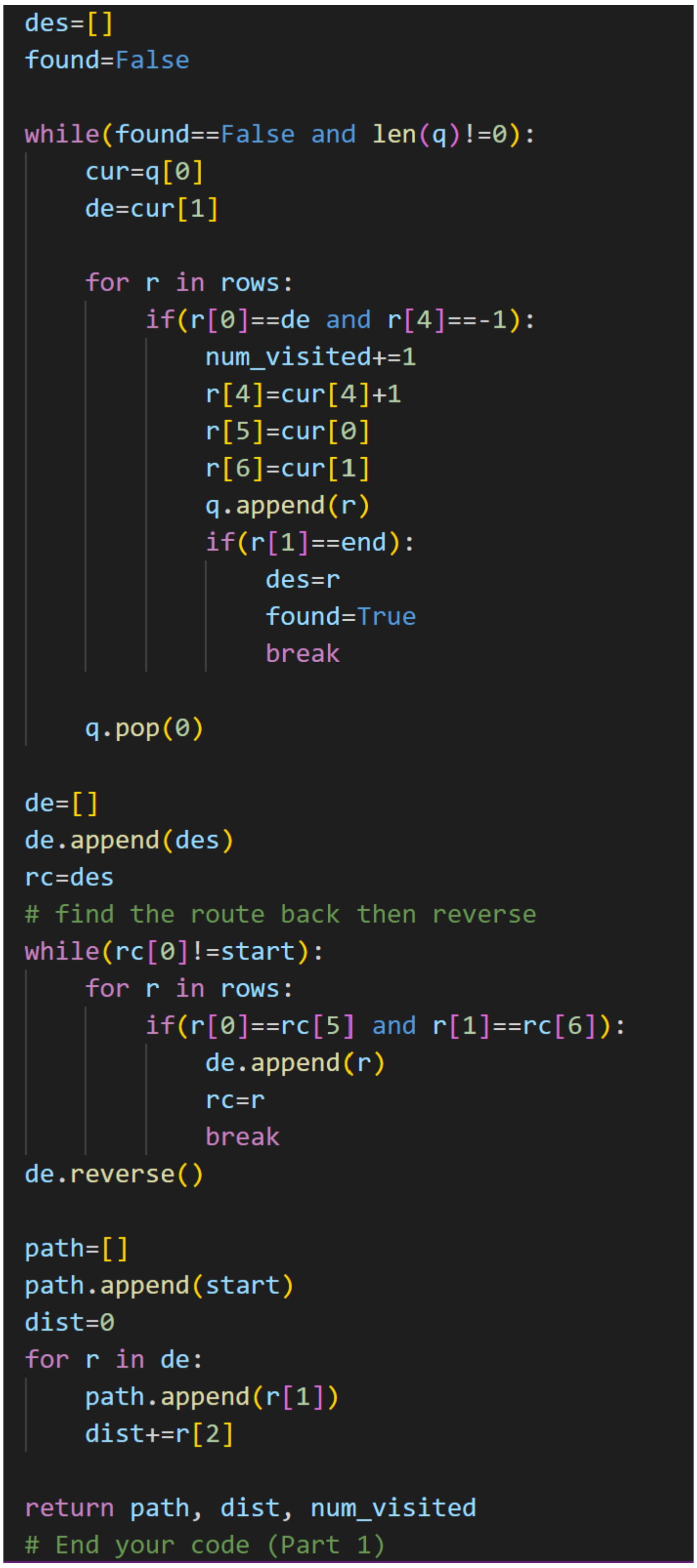
Homework 2: Route Finding

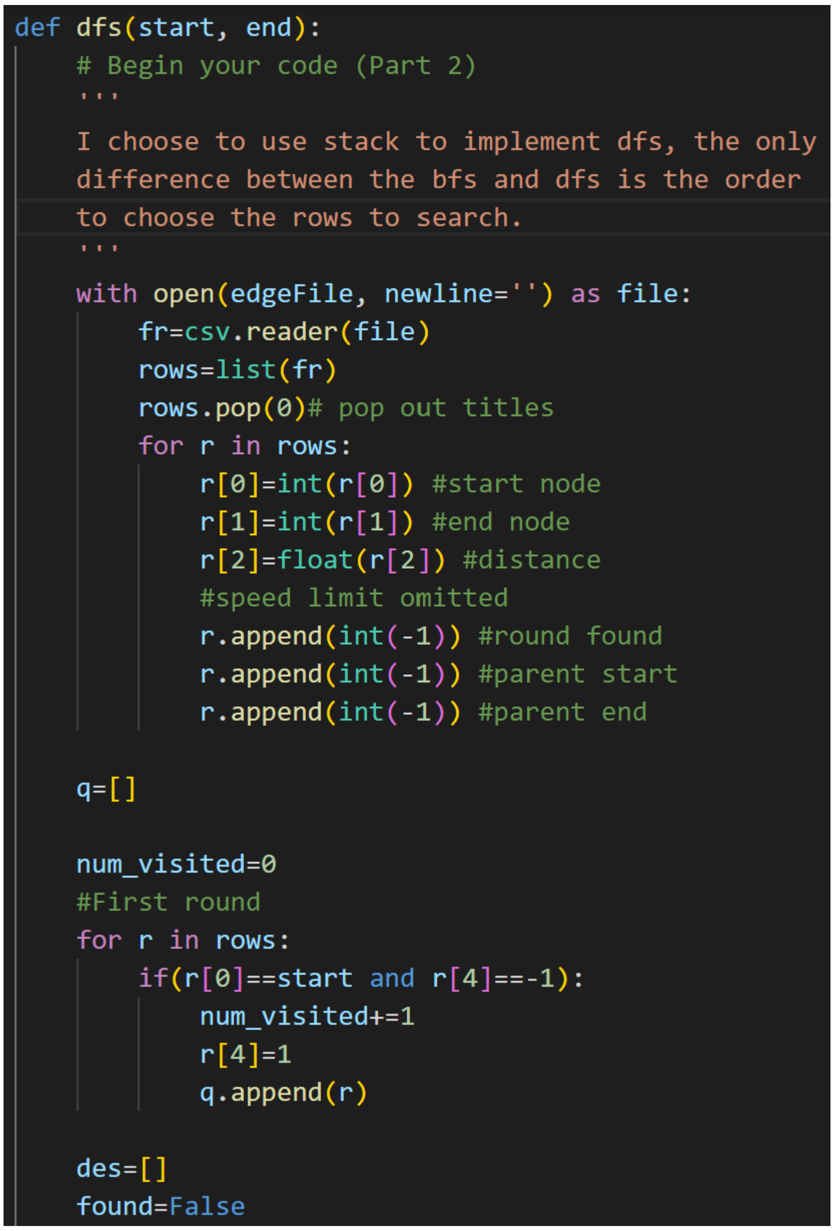
110550085 房天越

Part I. Implementation (6%):

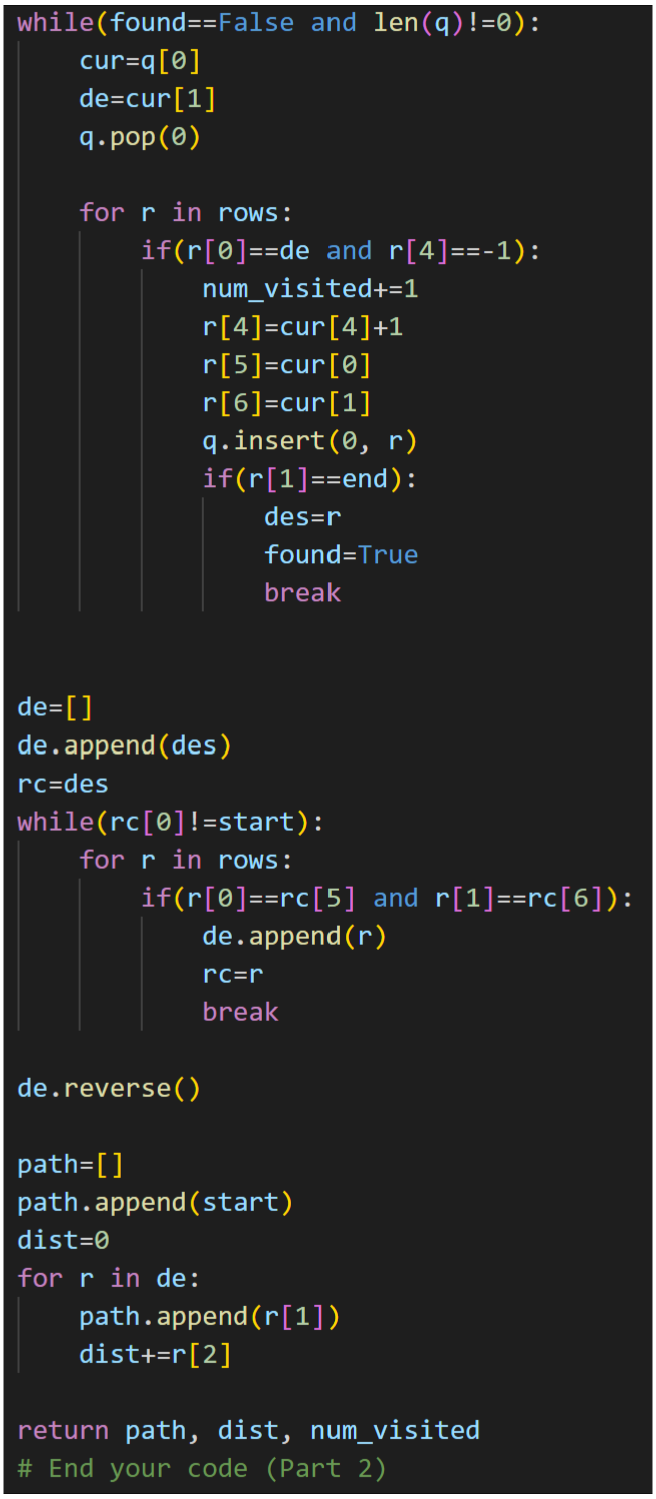


```python
def bfs(start, end):
    # Begin your code (Part 1)
    '''
    I first use csv to open the file, and give each row three parameters,
    the round that they're found, the parent's start place and end place.
    Then I use a queue q to store the running rows and run the bfs
    algorithm until I find a row that has its destination to be the end,
    and meanwhile record the nodes that are visited. Then, I use the list
    de to store the rows from end to start and then reverse it. Finally, I
    set the route to path and add the total distance to dist and return
    the values wanted.


    '''

    with open(edgeFile, newline='') as file:
        fr=csv.reader(file)
        rows=list(fr)
        rows.pop(0)# pop out titles
        for r in rows:
            r[0]=int(r[0]) #start node
            r[1]=int(r[1]) #end node
            r[2]=float(r[2]) #distance
            #speed limit omitted
            r.append(int(-1)) #round found
            r.append(int(-1)) #parent start
            r.append(int(-1)) #parent end

    q=[]

    num_visited=0
    #First round
    for r in rows:
        if(r[0]==start and r[4]==-1):
            num_visited+=1
            r[4]=1
            q.append(r)
```

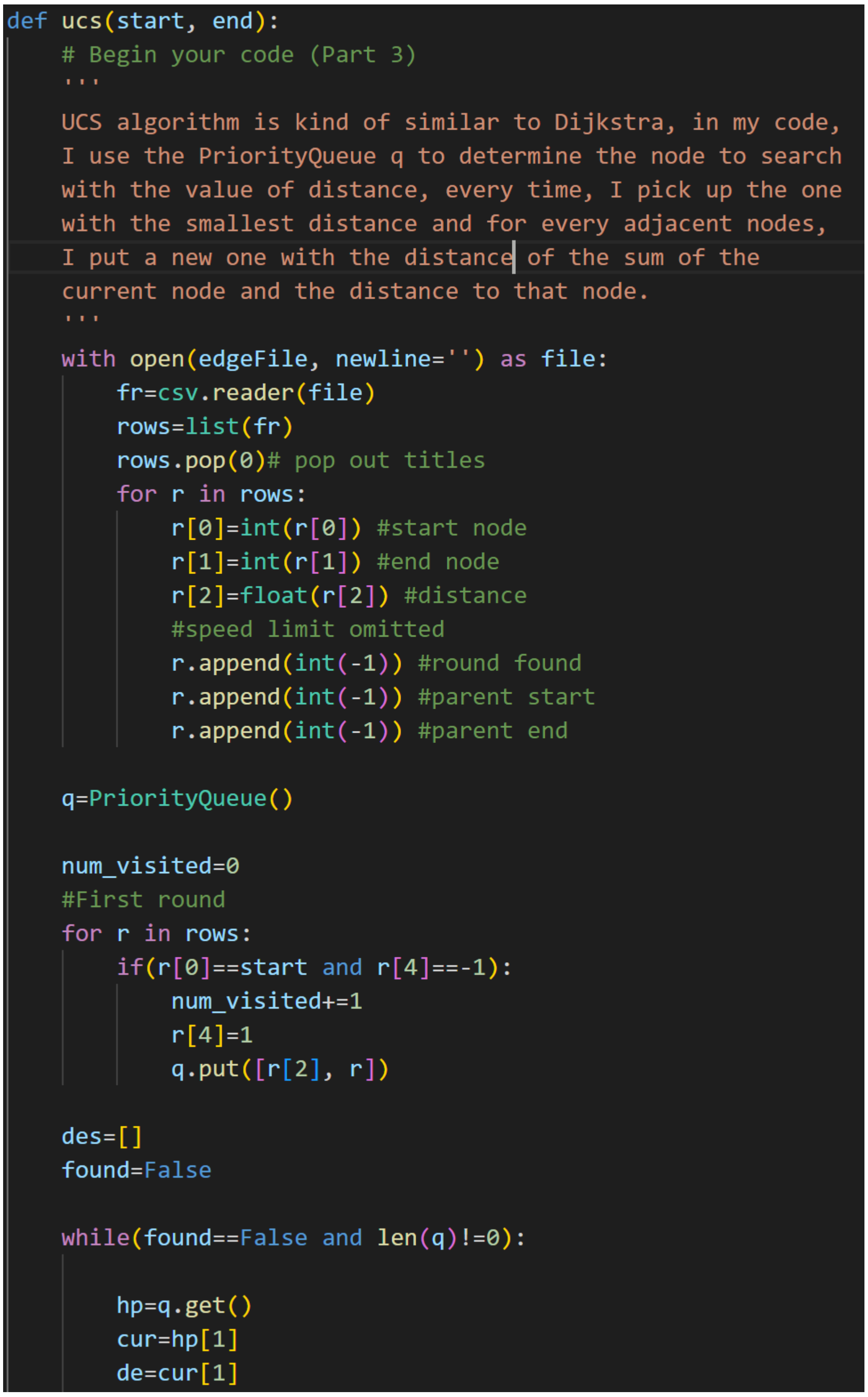

```
des=[]
found=False

while(found==False and len(q)!=0):
    cur=q[0]
    de=cur[1]

    for r in rows:
        if(r[0]==de and r[4]==-1):
            num_visited+=1
            r[4]=cur[4]+1
            r[5]=cur[0]
            r[6]=cur[1]
            q.append(r)
            if(r[1]==end):
                des=r
                found=True
                break

    q.pop(0)

de=[]
de.append(des)
rc=des
# find the route back then reverse
while(rc[0]!=start):
    for r in rows:
        if(r[0]==rc[5] and r[1]==rc[6]):
            de.append(r)
            rc=r
            break
de.reverse()

path=[]
path.append(start)
dist=0
for r in de:
    path.append(r[1])
    dist+=r[2]

return path, dist, num_visited
# End your code (Part 1)
```



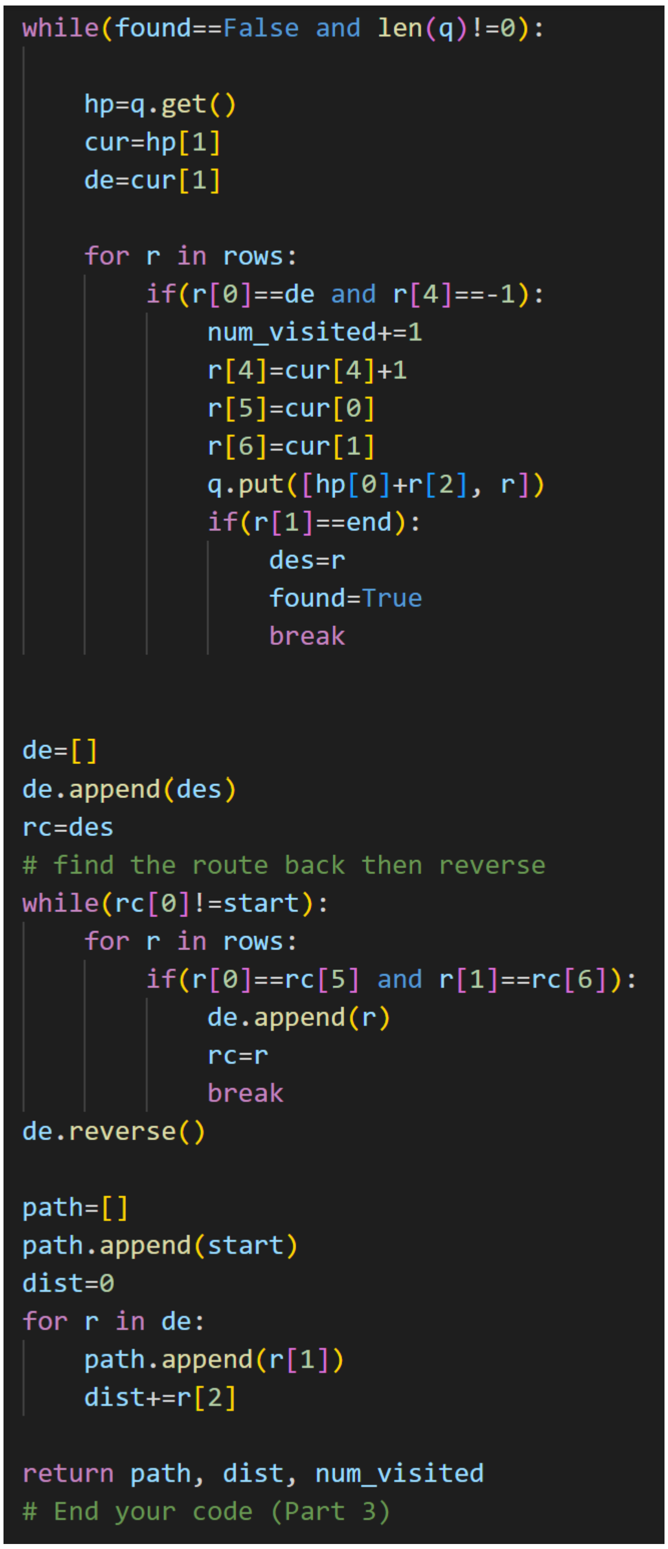
```
def dfs(start, end):
    # Begin your code (Part 2)
    '''
    I choose to use stack to implement dfs, the only
    difference between the bfs and dfs is the order
    to choose the rows to search.
    '''
    with open(edgeFile, newline='') as file:
        fr=csv.reader(file)
        rows=list(fr)
        rows.pop(0)# pop out titles
        for r in rows:
            r[0]=int(r[0]) #start node
            r[1]=int(r[1]) #end node
            r[2]=float(r[2]) #distance
            #speed limit omitted
            r.append(int(-1)) #round found
            r.append(int(-1)) #parent start
            r.append(int(-1)) #parent end

    q=[]

    num_visited=0
    #First round
    for r in rows:
        if(r[0]==start and r[4]==-1):
            num_visited+=1
            r[4]=1
            q.append(r)

    des=[]
    found=False
```



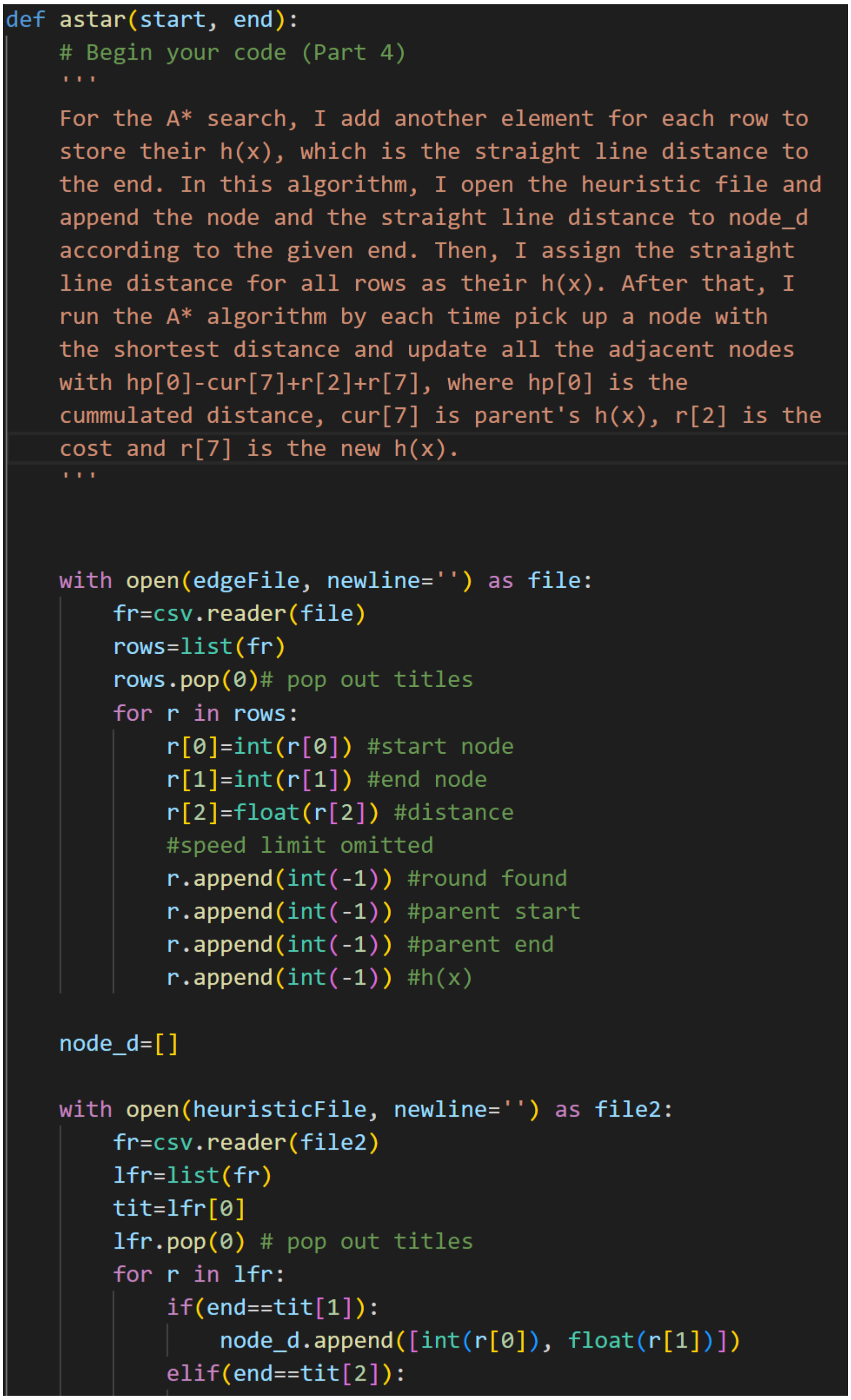
```python
while(found==False and len(q)!=0):
    cur=q[0]
    de=cur[1]
    q.pop(0)

    for r in rows:
        if(r[0]==de and r[4]==-1):
            num_visited+=1
            r[4]=cur[4]+1
            r[5]=cur[0]
            r[6]=cur[1]
            q.insert(0, r)
            if(r[1]==end):
                des=r
                found=True
                break


de=[]
de.append(des)
rc=des
while(rc[0]!=start):
    for r in rows:
        if(r[0]==rc[5] and r[1]==rc[6]):
            de.append(r)
            rc=r
            break

de.reverse()

path=[]
path.append(start)
dist=0
for r in de:
    path.append(r[1])
    dist+=r[2]

return path, dist, num_visited
# End your code (Part 2)
```



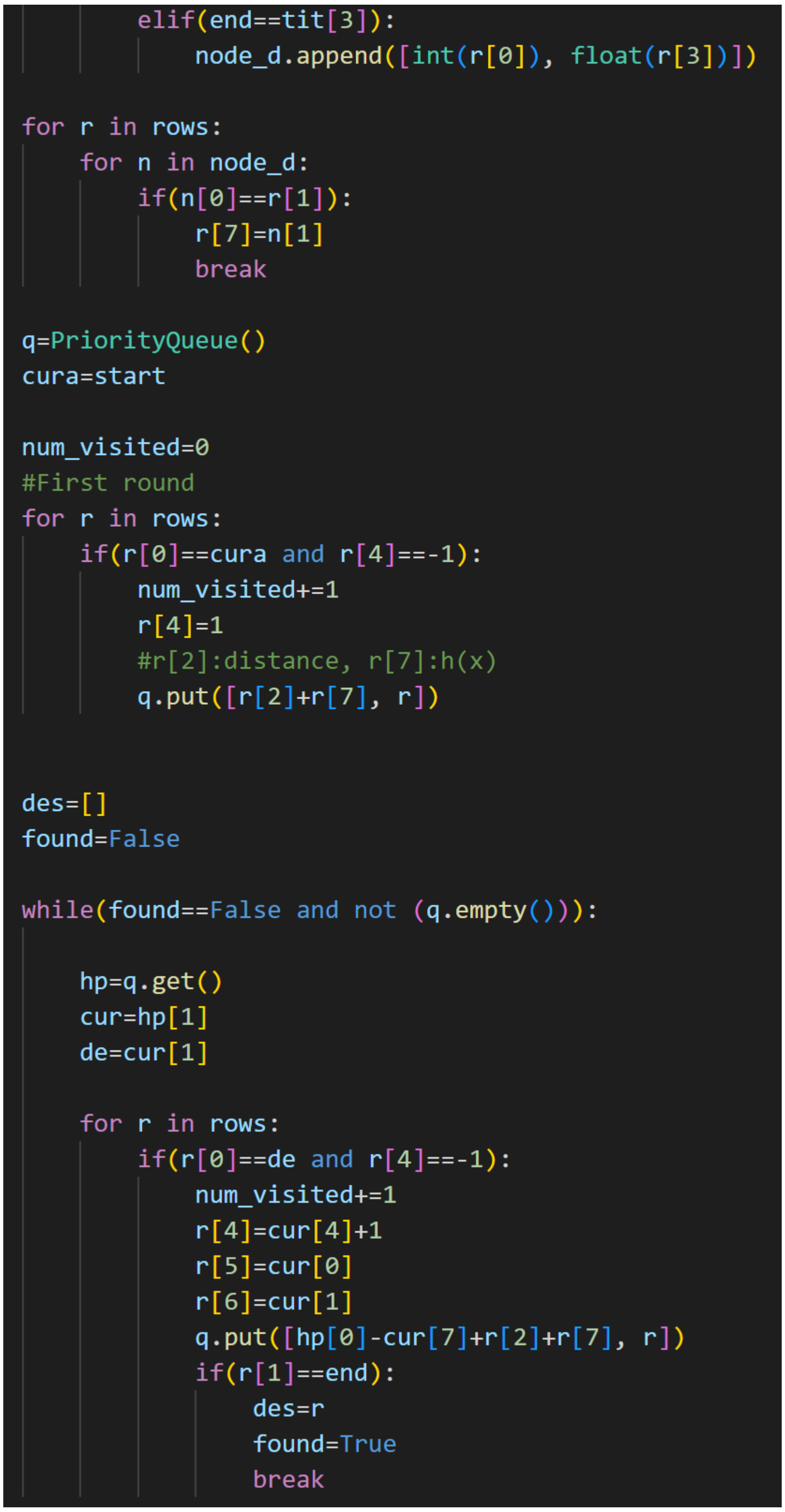
```python
def ucs(start, end):
    # Begin your code (Part 3)
    '''
    UCS algorithm is kind of similar to Dijkstra, in my code,
    I use the PriorityQueue q to determine the node to search
    with the value of distance, every time, I pick up the one
    with the smallest distance and for every adjacent nodes,
    I put a new one with the distance of the sum of the
    current node and the distance to that node.
    '''
    with open(edgeFile, newline='') as file:
        fr=csv.reader(file)
        rows=list(fr)
        rows.pop(0)# pop out titles
        for r in rows:
            r[0]=int(r[0]) #start node
            r[1]=int(r[1]) #end node
            r[2]=float(r[2]) #distance
            #speed limit omitted
            r.append(int(-1)) #round found
            r.append(int(-1)) #parent start
            r.append(int(-1)) #parent end

    q=PriorityQueue()

    num_visited=0
    #First round
    for r in rows:
        if(r[0]==start and r[4]==-1):
            num_visited+=1
            r[4]=1
            q.put([r[2], r])

    des=[]
    found=False

    while(found==False and len(q)!=0):

        hp=q.get()
        cur=hp[1]
        de=cur[1]
```



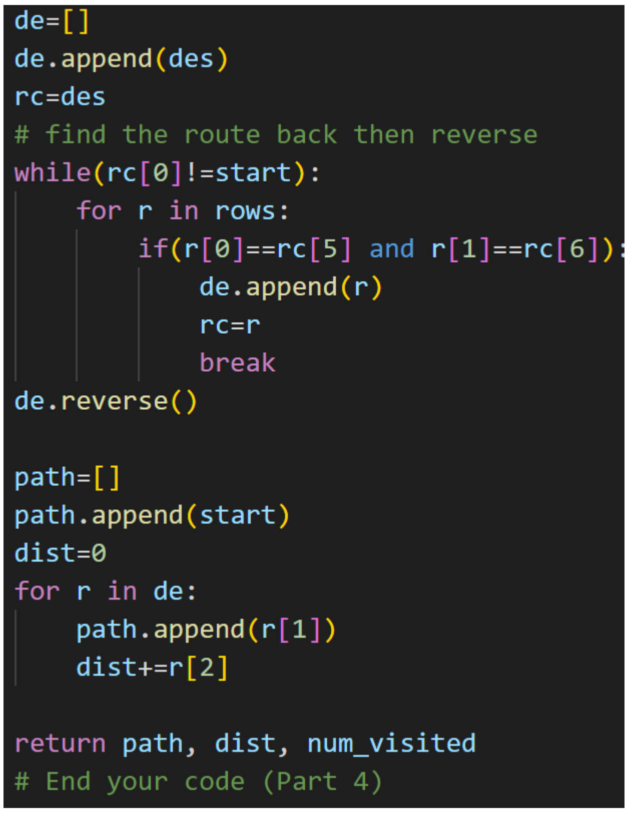
```python
while(found==False and len(q)!=0):

    hp=q.get()
    cur=hp[1]
    de=cur[1]

    for r in rows:
        if(r[0]==de and r[4]==-1):
            num_visited+=1
            r[4]=cur[4]+1
            r[5]=cur[0]
            r[6]=cur[1]
            q.put([hp[0]+r[2], r])
            if(r[1]==end):
                des=r
                found=True
                break


de=[]
de.append(des)
rc=des
# find the route back then reverse
while(rc[0]!=start):
    for r in rows:
        if(r[0]==rc[5] and r[1]==rc[6]):
            de.append(r)
            rc=r
            break
de.reverse()

path=[]
path.append(start)
dist=0
for r in de:
    path.append(r[1])
    dist+=r[2]

return path, dist, num_visited
# End your code (Part 3)
```



```python
def astar(start, end):
    # Begin your code (Part 4)
    '''
    For the A* search, I add another element for each row to
    store their h(x), which is the straight line distance to
    the end. In this algorithm, I open the heuristic file and
    append the node and the straight line distance to node_d
    according to the given end. Then, I assign the straight
    line distance for all rows as their h(x). After that, I
    run the A* algorithm by each time pick up a node with
    the shortest distance and update all the adjacent nodes
    with hp[0]-cur[7]+r[2]+r[7], where hp[0] is the
    cummulated distance, cur[7] is parent's h(x), r[2] is the
    cost and r[7] is the new h(x).
    '''


    with open(edgeFile, newline='') as file:
        fr=csv.reader(file)
        rows=list(fr)
        rows.pop(0)# pop out titles
        for r in rows:
            r[0]=int(r[0]) #start node
            r[1]=int(r[1]) #end node
            r[2]=float(r[2]) #distance
            #speed limit omitted
            r.append(int(-1)) #round found
            r.append(int(-1)) #parent start
            r.append(int(-1)) #parent end
            r.append(int(-1)) #h(x)

    node_d=[]

    with open(heuristicFile, newline='') as file2:
        fr=csv.reader(file2)
        lfr=list(fr)
        tit=lfr[0]
        lfr.pop(0) # pop out titles
        for r in lfr:
            if(end==tit[1]):
                node_d.append([int(r[0]), float(r[1])])
            elif(end==tit[2]):
```



```
        elif(end==tit[3]):
            node_d.append([int(r[0]), float(r[3])])

for r in rows:
    for n in node_d:
        if(n[0]==r[1]):
            r[7]=n[1]
            break

q=PriorityQueue()
cura=start

num_visited=0
#First round
for r in rows:
    if(r[0]==cura and r[4]==-1):
        num_visited+=1
        r[4]=1
        #r[2]:distance, r[7]:h(x)
        q.put([r[2]+r[7], r])


des=[]
found=False

while(found==False and not (q.empty())):

    hp=q.get()
    cur=hp[1]
    de=cur[1]

    for r in rows:
        if(r[0]==de and r[4]==-1):
            num_visited+=1
            r[4]=cur[4]+1
            r[5]=cur[0]
            r[6]=cur[1]
            q.put([hp[0]-cur[7]+r[2]+r[7], r])
            if(r[1]==end):
                des=r
                found=True
                break
```



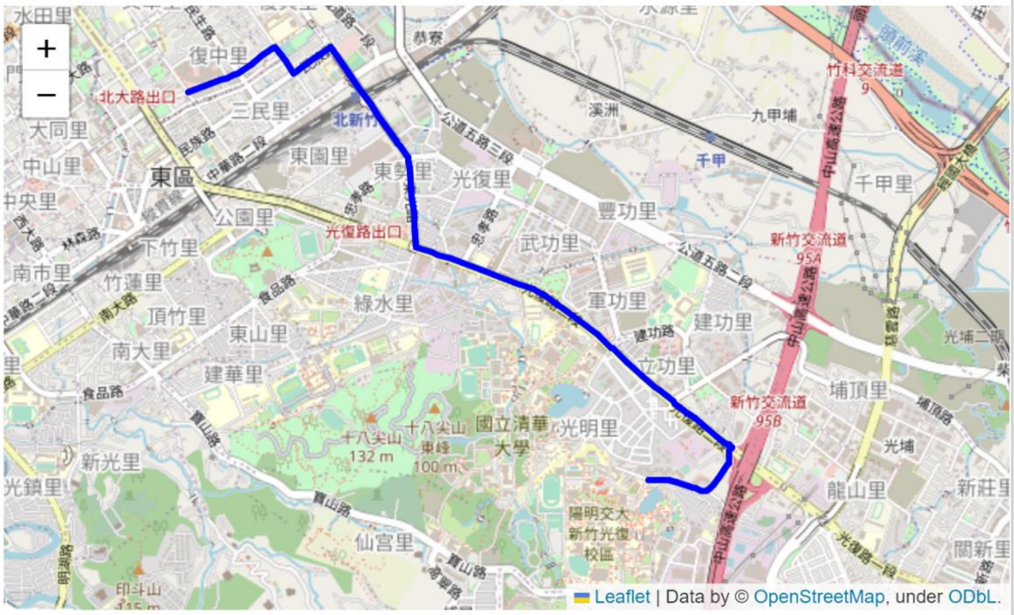
```python
de=[]
de.append(des)
rc=des
# find the route back then reverse
while(rc[0]!=start):
    for r in rows:
        if(r[0]==rc[5] and r[1]==rc[6]):
            de.append(r)
            rc=r
            break
de.reverse()

path=[]
path.append(start)
dist=0
for r in de:
    path.append(r[1])
    dist+=r[2]

return path, dist, num_visited
# End your code (Part 4)
```

Part II. Results & Analysis (12%):

Testcase 1:

BFS:

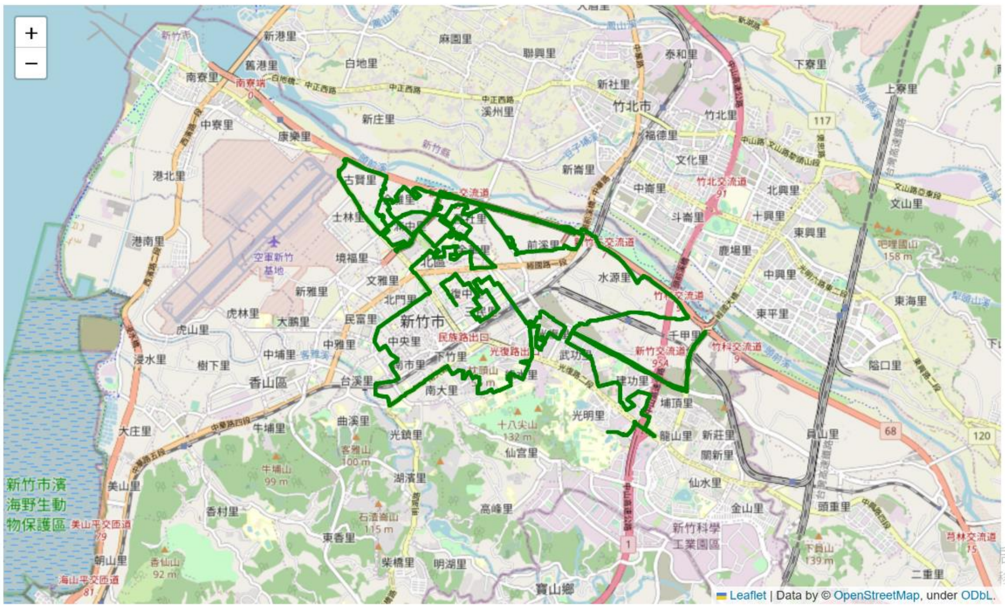
```
The number of nodes in the path found by BFS: 88
Total distance of path found by BFS: 4978.8820000000005 m
The number of visited nodes in BFS: 8115
```



DFS:

```
The number of nodes in the path found by DFS: 1232
Total distance of path found by DFS: 57208.987000000045 m
The number of visited nodes in DFS: 8420
```

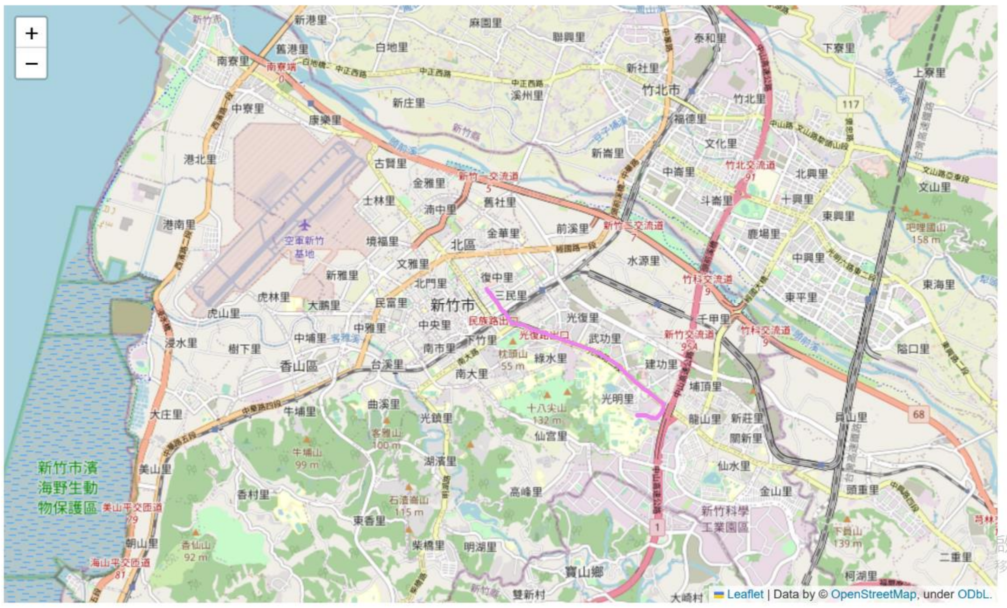
Out[5]:



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: 9568
```

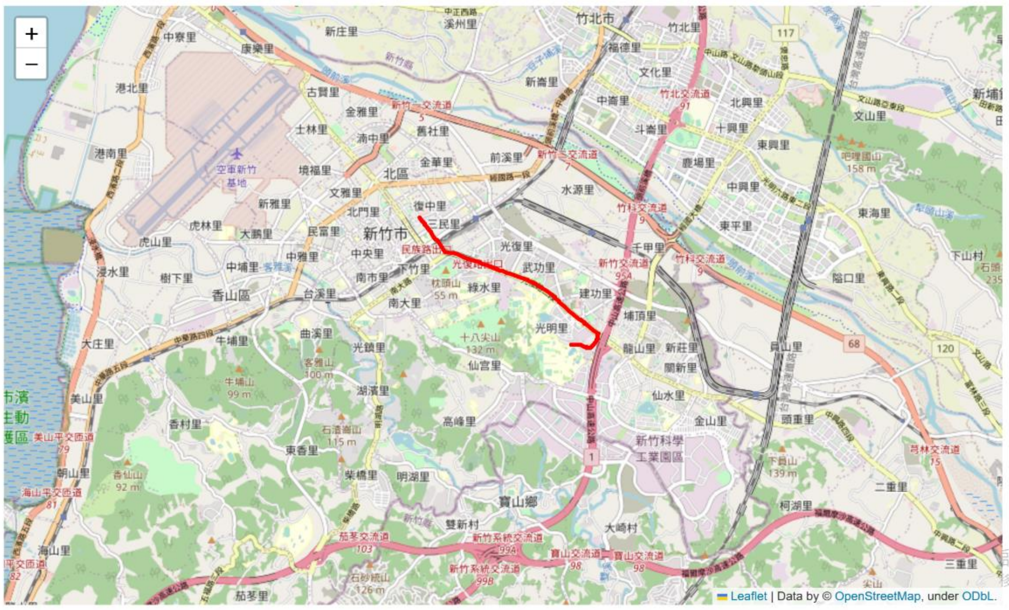
Out[6]:



A* Search:

```
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: 523
```

Out[7]:

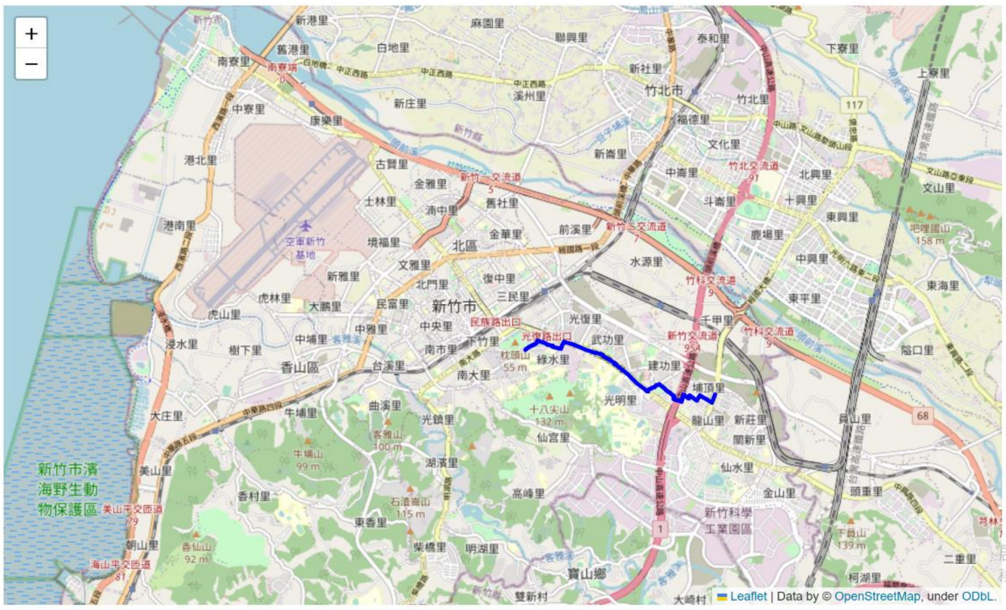


Testcase 2:

BFS:

```
The number of nodes in the path found by BFS: 60
Total distance of path found by BFS: 4215.521 m
The number of visited nodes in BFS: 9204
```

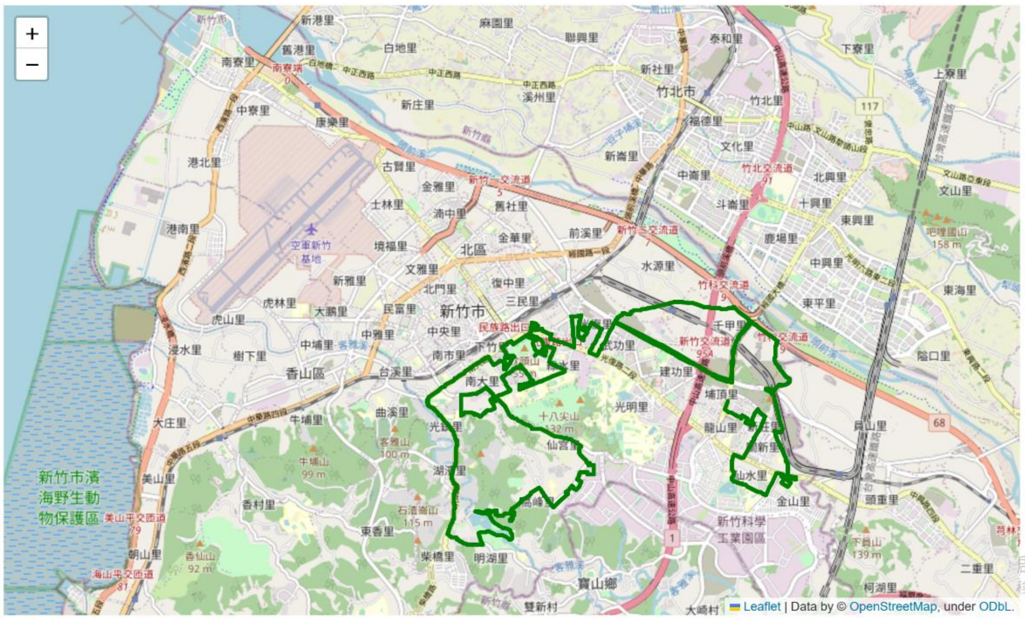
Out[4]:



DFS:

```
The number of nodes in the path found by DFS: 1010
Total distance of path found by DFS: 41597.10999999994 m
The number of visited nodes in DFS: 15925
```

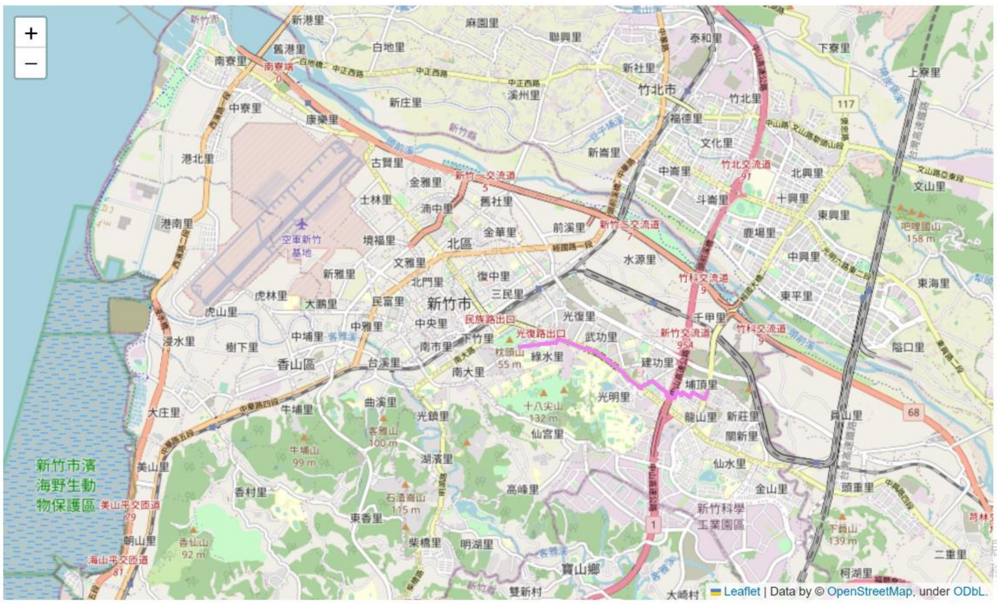
Out[5]:



UCS:

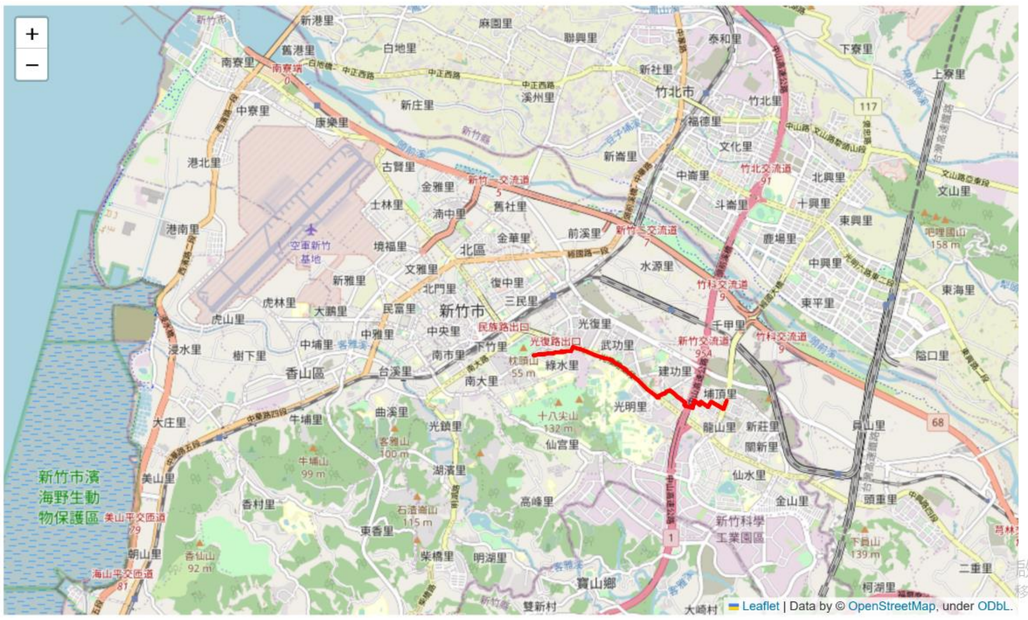
```
The number of nodes in the path found by UCS: 63
Total distance of path found by UCS: 4101.84 m
The number of visited nodes in UCS: 13560
```

Out[6]:



```
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: 13560
```

Out[7]:

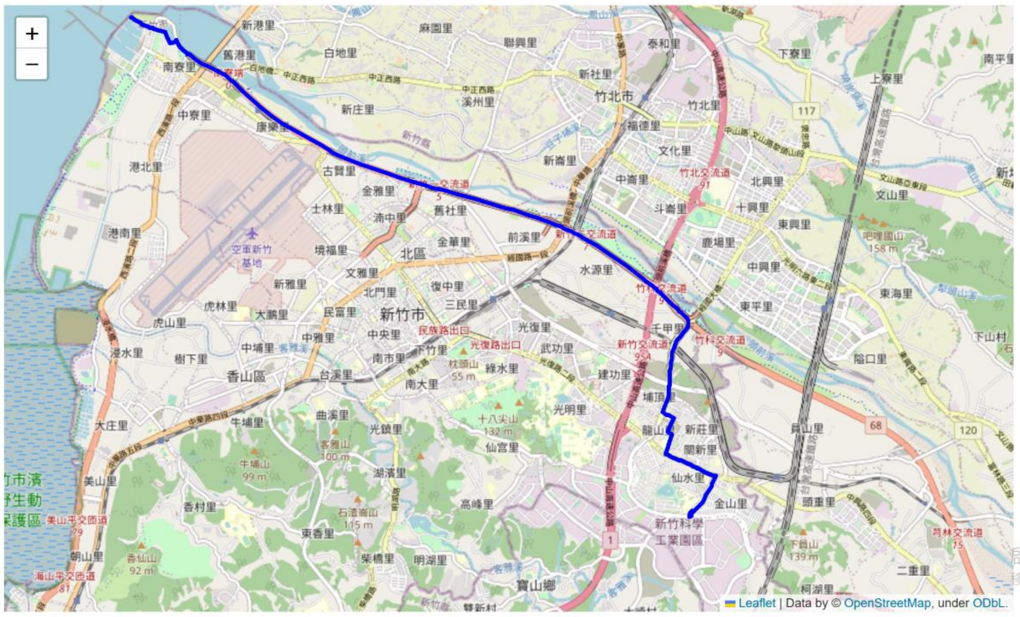


Testcase3:

BFS:

```
The number of nodes in the path found by BFS: 183
Total distance of path found by BFS: 15442.395000000002 m
The number of visited nodes in BFS: 21713
```

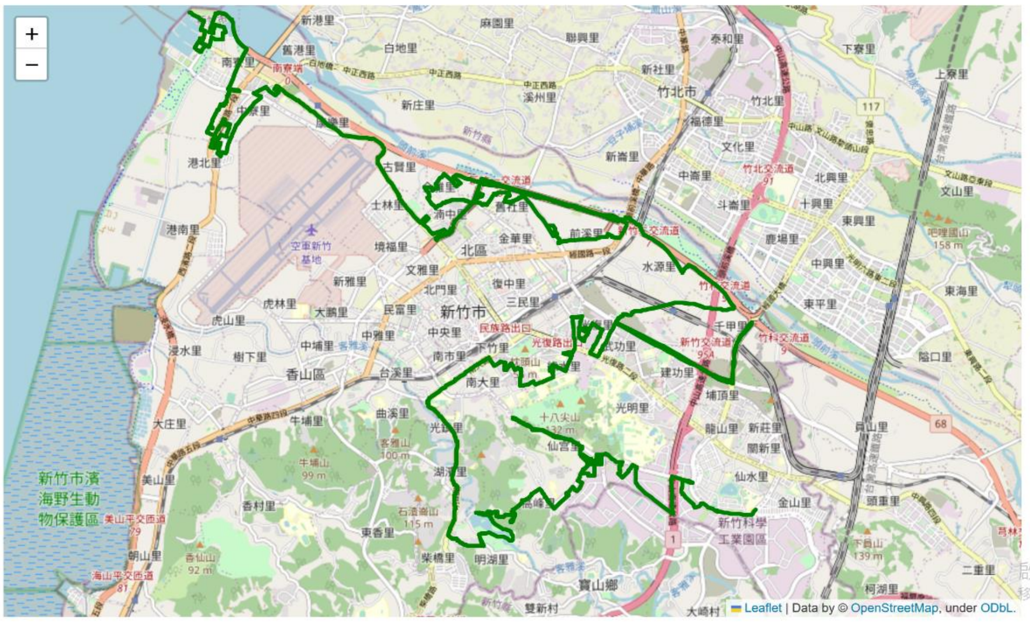
Out[4]:



DFS:

```
The number of nodes in the path found by DFS: 1521
Total distance of path found by DFS: 64821.60399999987 m
The number of visited nodes in DFS: 6331
```

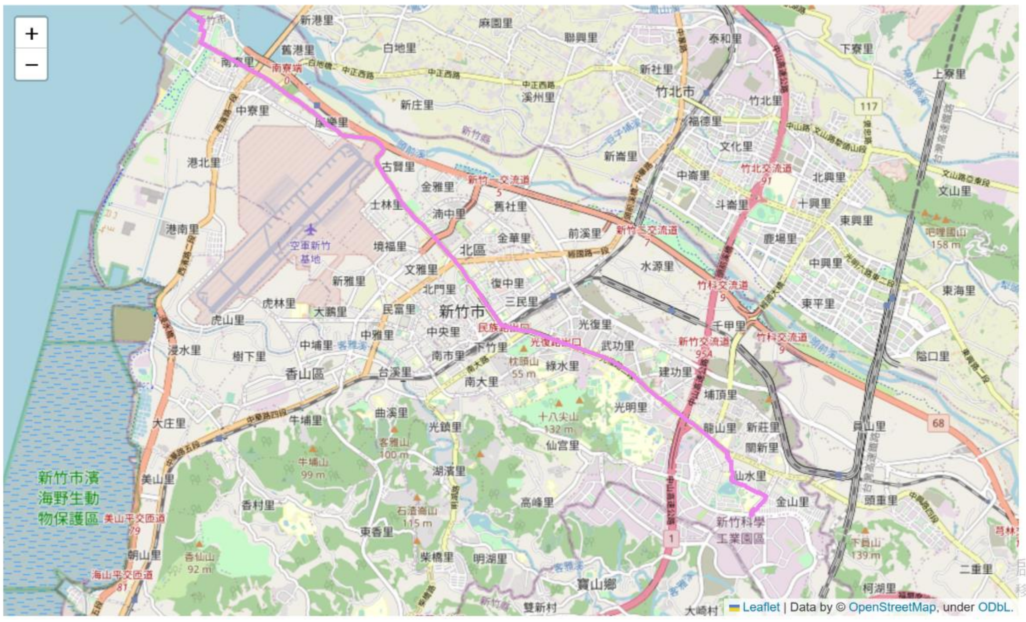
Out[5]:



UCS:

```
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: 23112
```

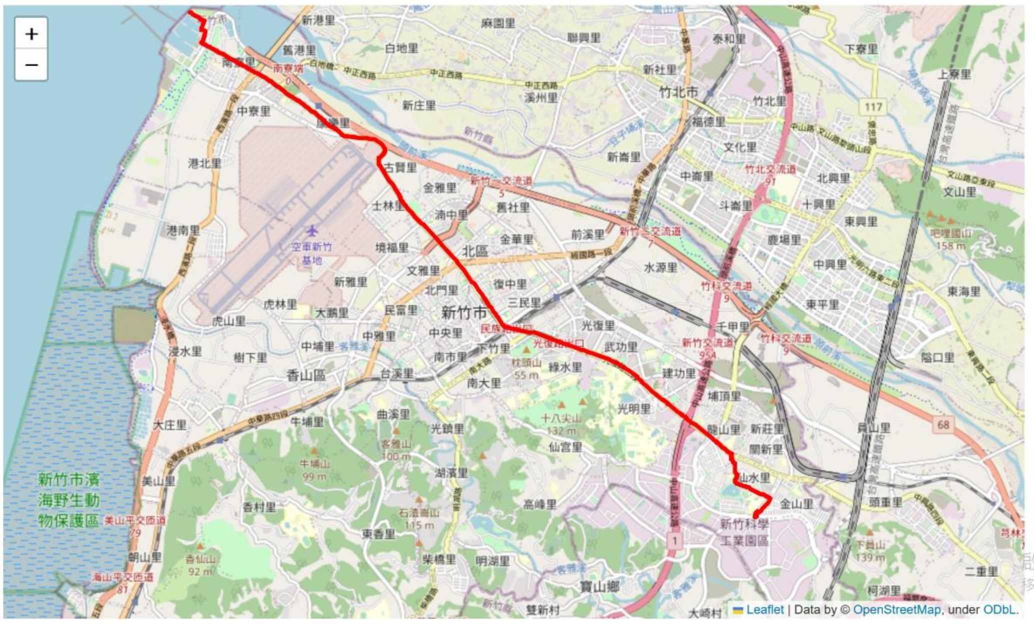
Out[6]:



A* Search:

```
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: 23112
```

Out[4]:



By observing the results, we could clearly see that DFS is not suitable for the shortest path search. And for the remaining three searches, BFS is simple, but doesn't yield the best result. And for UCS and A*, they both give us the best result. I don't know why but I get these two the same, while I am pretty sure that the way I implement them are different. A* should have given the best one.

Part III. Question Answering (12%):

1. Please describe a problem you encountered and how you solved it.

   Since I am still a noob to Python, I had a lot of problem looking for what kind of function or things that I should use for my algorithms, like for priority queues, I had quite a difficult time realizing what kind of parameters should I put in in the brackets of put() function and some other functions like that, so I searched for a long time and finally realized the meaning of them.

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 traffic situation also plays an essential role. For such a crowded island like Taiwan, there could always be a traffic jam, or may even be some traffic accidents. These slow down the speed and can result in worse choice for us. I know that Google would collect the number of users on the road and calculate whether it is crowded, and can use this as an important parameters when recommending users what routes they should go on.

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

   For mapping, we may use drones or even satellites to collect the images and get the route data (such as road distance, speed limit etc.) given by the website of government or by collecting ourselves.

   And for localization, we could use satellites to construct GPS system, and connect it to the phone to locate where we are.

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 ETA based on their mechanism. Please define a **dynamic heuristic equation** for ETA and explain the rationale of your design. Hint: You can consider meal prep time, delivery priority, multiple orders, etc.

The function I design is:

$$\Sigma_{(i\in n)}((\text{distance}_i)/(\text{average car speed}_i))$$

,where n is the route between one place to another place that are on the route of the delivery man.

There may be more than one places to go for the delivery man, such as the store, the spots that the man must go before they come to our place, etc. So, the heuristic time for the delivery man may be like this.