Homework 2: Route Finding

109550134 梁詠晴

Part I. Implementation (6%):

BFS :

```
def bfs(start, end):
    # Begin your code (Part 1)
    #raise NotImplementedError("To be implemented")
    visited=0
    with open(edgeFile, newline='') as file:# open edge data
        d = csv.reader(file)
        all_rows = list(d) #store data into an array
        all_rows.pop(0) #get rid of the first line
        for r in all_rows:
            r[0] = int(r[0]) # each row : [start,end,distance,speed limit,found @ which round, parent start, parent end]
            r[1] = int(r[1])
            r[2] = float(r[2])
            r.append(int(0))
            r.append(int(0))
            r.append(int(0))

    bfs_q = []
    cur_addr = start
    for r in all_rows: # put the start edge into the list
        if((r[0] == cur_addr)and r[4] == 0):
            visited = visited+1
            r[4] = 1
            bfs_q.append(r)

    dest = []
    find = False
    while(len(bfs_q)!=0 and find == False): # while not find or queue not empty
        cur = bfs_q[0] # get the first element in the list
        location = cur[1]
```

```
        for r in all_rows:
            if(r[0] == location and r[4] == 0): # if start of r = current edge's end，and hasn't been discovered
                visited = visited+1 #visit node +1
                r[4] = cur[4]+1 # round = parent's round+1
                r[5] = cur[0] # update parent's address
                r[6] = cur[1]
                bfs_q.append(r) # append r to the last of list
                if(r[1]==end): # when find end
                    num_visited = visited
                    dest = r #destination = r
                    find = True
                    break
        bfs_q.pop(0) # pop first

    d = [dest] # store path edges
    curr = dest
    while (curr[0]!=start):
        for r in all_rows:
            if(r[0]==curr[5] and r[1]== curr[6]): # start from destination, find parents iteratively until reach start
                d.append(r)
                curr = r
                break
    d.reverse()
    path = [start]
    dist = 0
    for r in d:
        path.append(r[1]) # store nodes of edge in sequence
        dist = dist + r[2] # sum distance

    return path,dist,num_visited
```

DFS(stack) :

```
def dfs(start, end):
    # Begin your code (Part 2)
    #raise NotImplementedError("To be implemented")
    visited=0
    with open(edgeFile, newline='') as file:# open edge data
        d = csv.reader(file)
        all_rows = list(d) #store data into an array
        all_rows.pop(0) #get rid of the first line
        for r in all_rows: # each row : [start,end,distance,speed limit,found @ which round, parent start, parent end]
            r[0] = int(r[0])
            r[1] = int(r[1])
            r[2] = float(r[2])
            r.append(int(0))
            r.append(int(0))
            r.append(int(0))

    bfs_q = []
    cur_addr = start
    for r in all_rows: # put the start edge into the list
        if((r[0] == cur_addr)and r[4] == 0):
            visited = visited+1
            r[4] = 1
            bfs_q.insert(0, r)

    dest = []
    find = False
```

```
    while(len(bfs_q)!=0 and find == False):  # while not find or queue not empty

        cur = bfs_q[0] # get the first element in the list
        bfs_q.pop(0) # pop first
        location = cur[1]

        #else:
        for r in all_rows:
            if(r[0] == location and r[4] == 0):# if start of r = current edge's end , and hasn't been discovered
                visited = visited+1 #visit node +1
                r[4] = cur[4]+1 # round = parent's round+1
                r[5] = cur[0] # update parent's address
                r[6] = cur[1]
                bfs_q.insert(0, r) # insert r to the head of the list
                if(r[1]==end): # when find end
                    num_visited = visited
                    dest = r #destination = r
                    find = True
                    break

    d = [dest] # store path edges
    curr = dest
    while (curr[0]!=start):
        for r in all_rows:
            if(r[0]==curr[5] and r[1]== curr[6]): # start from destination, find parents iteratively until reach start
                d.append(r)
                curr = r
                break
    d.reverse()
    path = [start]
    dist = 0
    for r in d:
        path.append(r[1]) # store nodes of edge in sequence
        dist = dist + r[2] # sum distance

    return path,dist,num_visited
```

UCS :

```
from queue import PriorityQueue

def ucs(start, end):
    # Begin your code (Part 3)
    #raise NotImplementedError("To be implemented")
    visited=0
    with open(edgeFile, newline='') as file:# open edge data
        d = csv.reader(file)
        all_rows = list(d) #store data into an array
        all_rows.pop(0) #get rid of the first line
        for r in all_rows:
            r[0] = int(r[0])# each row : [start,end,distance,speed limit,found @ which round, parent start, parent end]
            r[1] = int(r[1])
            r[2] = float(r[2])
            r.append(int(0))
            r.append(int(0))
            r.append(int(0))

    bfs_q = PriorityQueue()
    cur_addr = start
    for r in all_rows:  # put the start edge into the queue
        if((r[0] == cur_addr)and r[4] == 0):
            visited = visited+1
            r[4] = 1
            bfs_q.put([r[2],r])#priority : distance(cost)

    dest = []
    find = False
```

```
    while( (not bfs_q.empty()) and find == False):  # while not find or queue not empty

        c = bfs_q.get() # get the first element in the queue (with highest priority)
        cur = c[1]
        location = cur[1]

        #else:
        for r in all_rows:
            if(r[0] == location and r[4] == 0):# if start of r = current edge's end , and hasn't been discovered
                visited = visited+1 #visit node +1
                r[4] = cur[4]+1 # round = parent's round+1
                r[5] = cur[0] # update parent's address
                r[6] = cur[1]
                bfs_q.put([(c[0]+r[2]),r]) # priority : c[0] = cummulated cost,cur[7]=parent's h, r[2] = cost
                if(r[1]==end): # when find end
                    num_visited = visited
                    dest = r #destination = r
                    find = True
                    break

    d = [dest] # store path edges
    curr = dest
    while (curr[0]!=start):
        for r in all_rows:
            if(r[0]==curr[5] and r[1]== curr[6]): # start from destination, find parents iteratively until reach start
                d.append(r)
                curr = r
                break
    d.reverse()
    path = [start]
    dist = 0
    for r in d:
        path.append(r[1]) # store nodes of edge in sequence
        dist = dist + r[2] # sum distance

    return path,dist,num_visited
```

A* :

```
edgeFile = 'edges.csv'
heuristicFile = 'heuristic.csv'
from queue import PriorityQueue


def astar(start, end):
    # Begin your code (Part 4)
    #raise NotImplementedError("To be implemented")
    # End your code (Part 4)
    visited=0
    with open(edgeFile, newline='') as file: # open edge data
        d = csv.reader(file)
        all_rows = list(d) #store data into an array
        all_rows.pop(0) #get rid of the first line
        for r in all_rows:
            r[0] = int(r[0]) # each row : [start,end,distance,speed limit,found @ which round, parent start, parent end, h]
            r[1] = int(r[1])
            r[2] = float(r[2])
            r.append(int(0))
            r.append(int(0))
            r.append(int(0))
            r.append(int(0))

    node_distance = [] # store h of each node, h depends on end
    if(end == 1079387396) : key = 1
    elif (end == 1737223506) : key = 2
    elif(end == 8513026827) : key = 3

    with open(heuristicFile, newline='') as file: # read heuristic data
        d = csv.reader(file)
        n = list(d)
        n.pop(0)
        for row in n:
            node_distance.append([int(row[0]),float(row[key])]) # store the data into an array
```

```
    for r in all_rows:
        for n in node_distance: #update h data of edges in all_rows
            if(n[0]==r[1]):
                r[7] = n[1]
                break


    bfs_q = PriorityQueue()
    for r in all_rows:
        if((r[0] == start)and r[4] == 0): # put the start edge into the queue
            visited = visited+1
            r[4] = 1
            bfs_q.put([(r[2]+r[7]),r]) #priority : distance(cost) + h of the edge


    dest = []
    find = False
    while( (not bfs_q.empty()) and find == False): # while not find or queue not empty

        c = bfs_q.get() # get the first element in the queue (with highest priority)
        cur = c[1]
        location = cur[1] # get current location


        for r in all_rows:
            if(r[0] == location and r[4] == 0): # if start of r = current edge's end , and and hasn't been discovered
                visited = visited+1 #visit node +1
                r[4] = cur[4]+1 # round = parent's round+1
                r[5] = cur[0] # update parent's address
                r[6] = cur[1]
                bfs_q.put([(c[0]-cur[7]+r[2]+r[7]),r]) #priority :  c[0] = cummulated cost,cur[7]=parent's h, r[2] = cost, r[7] = h(x)
                if(r[1]==end): # when find end
                    num_visited = visited
                    dest = r #destination = r
                    find = True
                    break
```

```
    d = [dest] # store path edges
    curr = dest
    while (curr[0]!=start):
        for r in all_rows:
            if(r[0]==curr[5] and r[1]== curr[6]): # start from destination, find parents iteratively until reach start
                d.append(r)
                curr = r
                break
    d.reverse()
    path = [start]
    dist = 0
    for r in d:
        path.append(r[1]) # store nodes of edge in sequence
        dist = dist + r[2] # sum distance

    return path,dist,num_visited
```

Part II. Results & Analysis (12%):

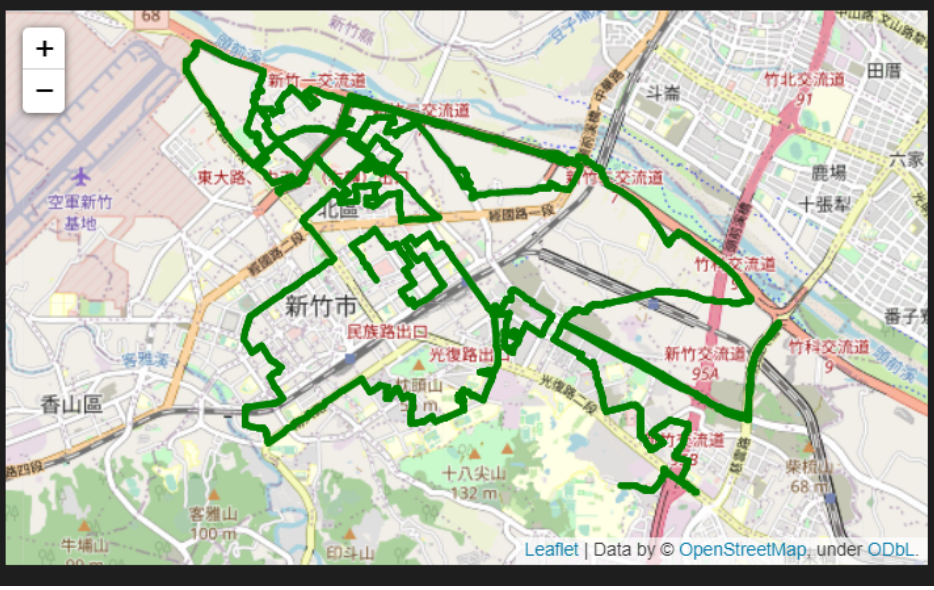
Test 1 (start = 2270143902, end = 1079387396) :

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: 8397
```

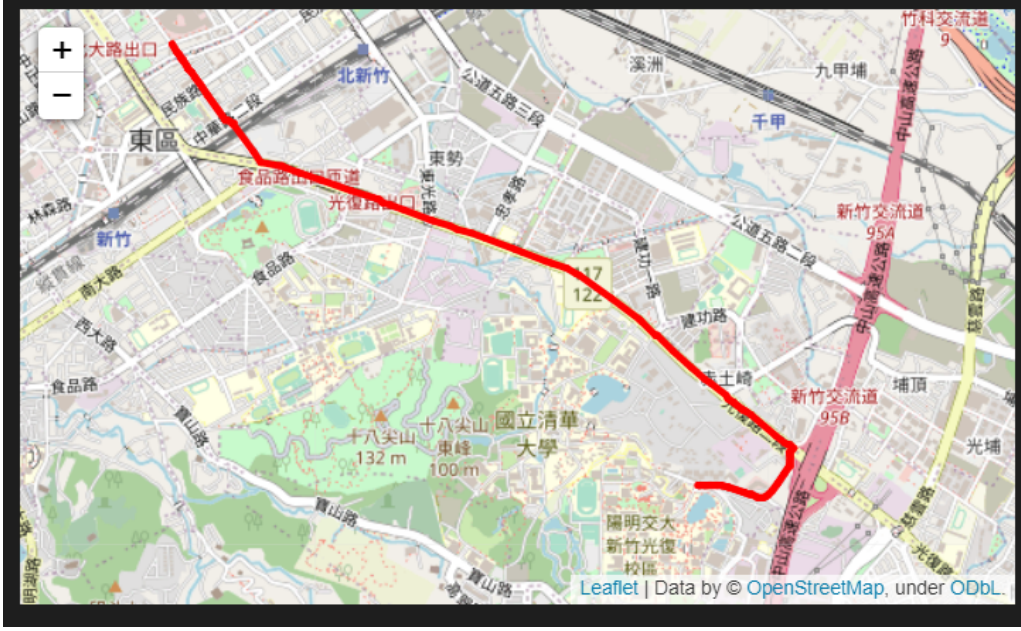


UCS :

```
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
```

A* :

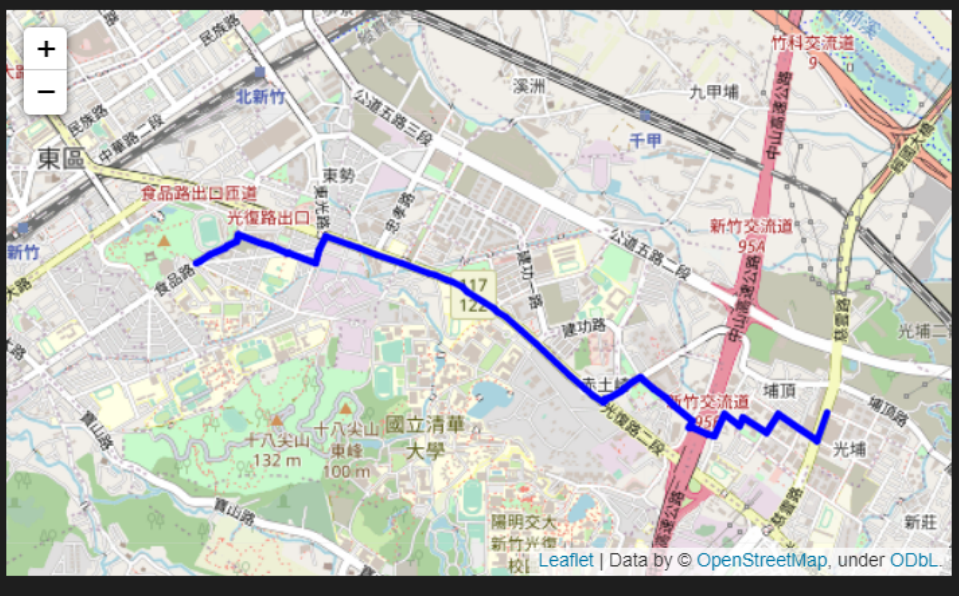
```
The number of nodes in the path found by A* search: 89
Total distance of path found by A* search: 4367.881 m
The number of visited nodes in A* search: 523
```



Test 2 (start = 426882161, end = 1737223506) :

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
```

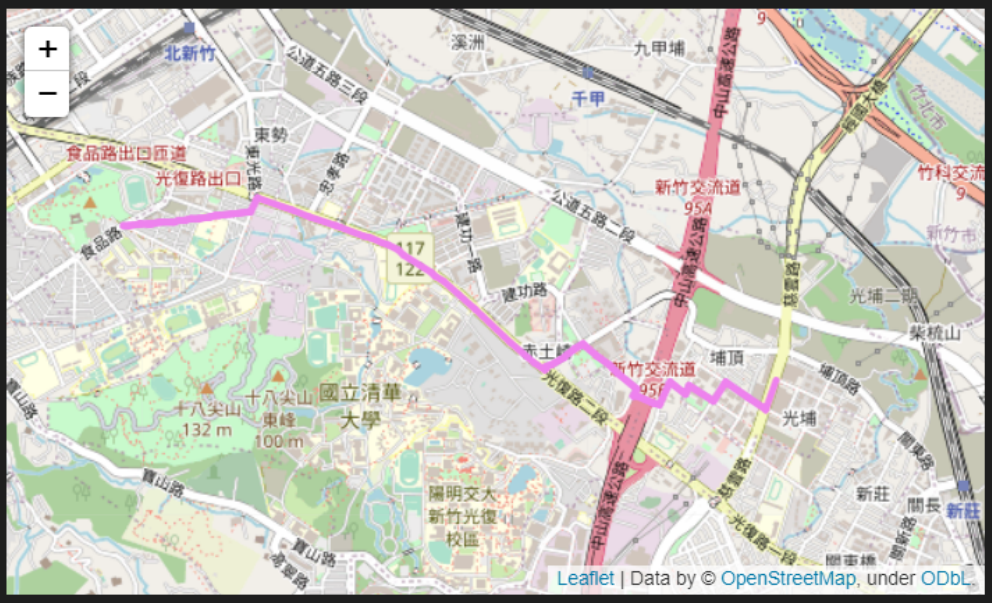


DFS :

```
The number of nodes in the path found by DFS: 998
Total distance of path found by DFS: 41094.65799999992 m
The number of visited nodes in DFS: 15889
```

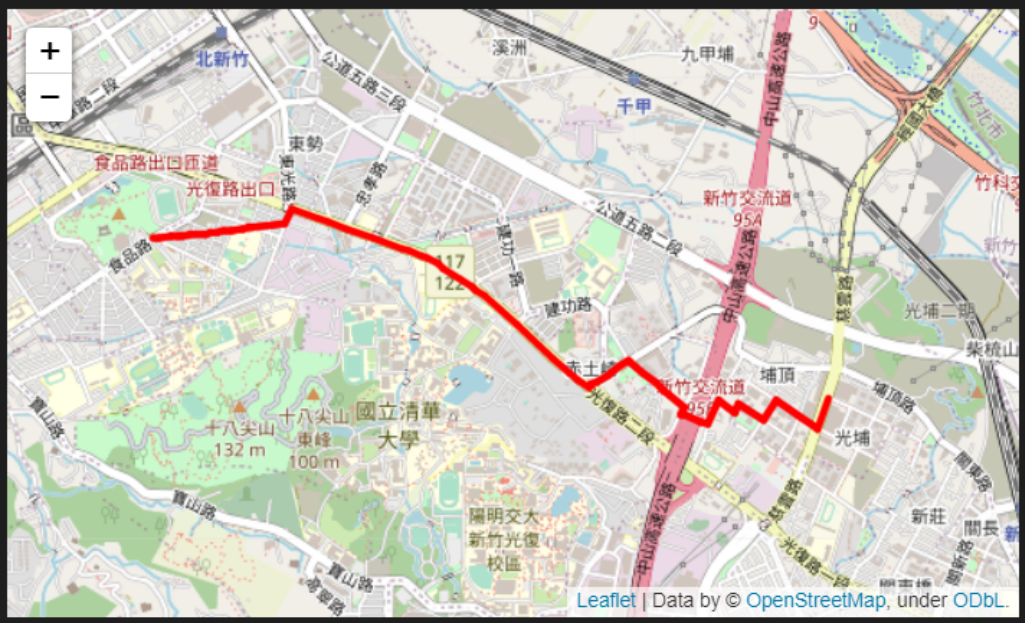
UCS :

```
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
```



A* :

```
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: 2429
```



Test 3 (start = 1718165260 ,end = 8513026827) :

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
```

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
```

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
```

A* :

```
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: 14428
```

Analysis :

According to the results of three cases, I found that the path found by DFS is the longest one. May because DFS continues deep on the same path without considering whether there's a faster way to go, this path made a detour a lot. BFS had a better result compared with DFS, because BFS finds paths with fewest nodes. Both UCS and A* finds the shortest path, because they considered the actual distance of the path and chose the shortest one, and A* visited less nodes than UCS, because A* considered goal proximity and thus raise the priority of possible choices when deciding the priority of edges.

Part III. Question Answering (12%):

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

   At first, when checking if an edge is connected to the current location, I checked both 'start' and 'end' addresses. However, the result is different from the answer. Then I figure paths have their own directions, and there may be one-way streets, thus it's inappropriate to check the edge's 'end'. I solved the problem by checking only the 'start' address of the edge.After modification, the result was correct.

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.

   Road conditions may also affect route finding. For example, when there's a car accident or a traffic jam, the speed will be slower than the speed limit, and it may take more time going on the original shortest path.

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 : may construct a map based on satellite imagery.

   Localization : may use GPS to connect to the mobile device besides a person.

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.

   H = (distance to the goal) / (average speed of all cars on the road)

   Calculate the possible time distance according to the actual speed a car may have on the path, and choose the path that currently takes least time to reach the goal.