Homework 2: Route Finding

Part I. Implementation (6%):

- **BFS**

```python
def bfs(start, end):
    '''
    1. read all the data from .csv
    2. do bfs with a queue iteratively until end is found:
            pop first element -> add the path adjacent to the point into queue and fromNode
    3. trace back the path and compute the total distant
    '''
    # Begin your code (Part 1)
    data = []
    queue = []
    queue.append(start)
    fromNode = tuple()
    visited = [start]
    path = []
    done = False
    count = 0
    with open(edgeFile, newline='') as csvfile:                    # read from csv
        reader = csv.reader(csvfile, delimiter=',', quotechar='"')
        next(reader)
        for row in reader:
            data.append(tuple(row))

    while(done == False):                                          # loop & do bfs
        now = queue.pop(0)                                         # pop a point from queue
        visited.append(now)                                        # add it to visited
        for i in data:
            if (int(i[0]) == now and int(i[1]) not in visited and int(i[1]) not in queue):
                fromNode = fromNode + ((int(i[0]),int(i[1]),float(i[2])),)  # add possible path
                queue.append(int(i[1]))                            # add the dest. node into queue
                if int(i[1])==end:                                 # found end -> break
                    done = True

    now = end
    num_visited = len(visited)
    dist = 0
    while now!=start:                                              # trace back
        for way in fromNode:
            if way[1] == now:                                      # if it is the path
                path.insert(0,now)                                 # add to path
                dist += way[2]                                     # add the distance
                now = way[0]                                       # renew dest. point
                break
    path.insert(0,start)                                           # insert the start point
    return path,dist,num_visited
    # End your code (Part 1)
```

variable explaination:

($variableName means the variable in the code)

$data : 2D array of the content of edgeFile.csv

$queue : 1D array of the integer for node numbers in queue

$fromNode : tuple of possible path (int,int,float)

- **DFS**

```python
def dfs(start, end):
    '''
    1. read all the data from .csv
    2. do bfs with a stack iteratively until end is found:
            pop last element -> add the path adjacent to the point into stack and fromNode
    3. trace back the path and compute the total distant
    '''
    # Begin your code (Part 2)
    data = []
    stack = []
    stack.append(start)
    fromNode = tuple()
    visited = [start]
    path = []
    done = False
    with open(edgeFile, newline='') as csvfile:                          # read .csv
        reader = csv.reader(csvfile, delimiter=',', quotechar='"')
        next(reader)
        for row in reader:
            data.append(tuple(row))

    while(done == False):
        now = stack.pop()                                                # pop the last element
        visited.append(now)
        for i in data:
            if (int(i[0]) == now and int(i[1]) not in visited and int(i[1]) not in stack):
                fromNode = fromNode + ((int(i[0]),int(i[1]),float(i[2])),)      # add to possible path
                stack.append(int(i[1]))                                  # add to stack
                if int(i[1])==end:                                       # break if end is found
                    done = True

    now = end                                                            # trace back
    num_visited = len(visited)
    dist = 0
    while now!=start:
        for way in fromNode:
            if way[1] == now:
                path.insert(0,now)
                dist += way[2]
                now = way[0]
                break
    path.insert(0,start)
    return path,dist,num_visited
    # End your code (Part 2)
```

variable explaination:

($variableName means the variable in the code)

$data : 2D array of the content of edgeFile.csv

$stack : 1D array of the integer for node numbers in stack

$fromNode : tuple of possible path (int,int,float)

- **UCS**

```python
def ucs(start, end):
    '''
    1. read from the .csv
    2. push the point adjacent to the starting point with lengths
    3. loop: pop the smallest element, add to visited and path, and push element into priority queue
    4. trace back to get the path.
    '''
    # Begin your code (Part 3)
    data = []
    queue = []
    visited = [start]
    fromNode = tuple()
    path = []
    with open(edgeFile, newline='') as csvfile:                    # read from csv
        reader = csv.reader(csvfile, delimiter=',', quotechar='"')
        next(reader)
        for row in reader:
            data.append(tuple(row))

    for i in data:                                                  # push all the node for start point
        if(int(i[0]) == start):
            queue.append((i[0],i[1],i[2],i[2]))

    minis = 0

    while(1):                                                       # loop
        minindex = 0                                                # choose the one with min length
        minis = float(queue[0][2])
        for i in range(len(queue)):
            if(float(queue[i][2])<minis):
                minis = float(queue[i][2])
                minindex = i

        now = queue.pop(minindex)                                   # pop the min one
        visited.append(int(now[1]))                                 # add the dest. of the way to visited
        fromNode = fromNode + (now,)                                # add it to path

        if int(now[1])==end:                                        # break if end is found
            break
        count = 0
        for i in range(len(queue)):                                 # pop unnecessary element in priority queue
            if(queue[i-count][1]==now[1]):                          # (the point is visited)
                queue.pop(i-count)
                count+=1

        for i in data:                                              # push element into priority queue
            if (i[0] == now[1] and int(i[1]) not in visited):       # and also add the total length from starting point
                queue.append((i[0],i[1],str(float(i[2])+float(now[2])),i[2]))

    now = end
    num_visited = len(visited)
    dist = 0

    while now!=start:                                               # trace back from end to start
        #print(str(now)+"\n")
        for way in fromNode:                                        # if the dest. of a way is the end
            if int(way[1]) == now:
                path.insert(0,now)                                  # add it to path
                dist += float(way[3])                               # add the destination
                now = int(way[0])                                   # update the end
                break
    path.insert(0,start)
    return path,dist,num_visited
    # End your code (Part 3)
```

variable explaination:

$queue : array of edge in a tuple(starting node, destination node, total length from starting point, the length of edge)

When adding node into priority queue, append it with the total length (adding total length of previous node and the length of the edge), so when popping, you can use it directly.

- **A* search**

```python
def astar(start, end):
    '''
    1. read data from the 2 csv
    2. push nodes for starting point
    3. loop:
            pop the min g(n) + h(n) -> break if dest. is end -> add to visited and fromNode -> push adjacent nodes into queue
    4. trace back
    '''
    # Begin your code (Part 4)
    data = []
    heuristicData = []
    queue = []
    visited = [start]
    fromNode = tuple()
    path = []
    with open(edgeFile, newline='') as csvfile:                      # read from csv
        reader = csv.reader(csvfile, delimiter=',', quotechar='"')
        next(reader)
        for row in reader:
            data.append(tuple(row))
    with open(heuristicFile, newline='') as csvfile:                 # read from csv
        reader = csv.reader(csvfile, delimiter=',', quotechar='"')
        next(reader)
        for row in reader:
            heuristicData.append(tuple(row))

    for i in data:                                                   # push all the node for start point
        if(int(i[0]) == start):
            indexNew = next((j for j, x in enumerate(heuristicData) if x[0] == i[1]), None)
            if(end == 1079387396):
                    queue.append((i[0],i[1],str(float(i[2])+float(heuristicData[indexNew][1])),i[2],i[2]))
            elif(end == 1737223506):
                    queue.append((i[0],i[1],str(float(i[2])+float(heuristicData[indexNew][2])),i[2],i[2]))
            elif(end == 8513026827):
                    queue.append((i[0],i[1],str(float(i[2])+float(heuristicData[indexNew][3])),i[2],i[2]))

    minis = 0

    while(1):                                                        # loop
        minindex = 0                                                 # choose the one with min length
        minis = float(queue[0][2])
        for i in range(len(queue)):
            if(float(queue[i][2])<minis):
                minis = float(queue[i][2])
                minindex = i

        now = queue.pop(minindex)                                    # pop the min one
        visited.append(int(now[1]))                                  # add the dest. of the way to visited
        fromNode = fromNode + (now,)                                 # add it to path

        if int(now[1])==end:                                         # break if end is found
            break
        count = 0
        for i in range(len(queue)):                                  # pop unnecessary element in priority queue
            if(queue[i-count][1]==now[1]):                           # (the point is visited)
                queue.pop(i-count)
                count+=1

        for i in data:                                               # push element into priority queue
            if (i[0] == now[1] and int(i[1]) not in visited):
                indexNew = next((j for j, x in enumerate(heuristicData) if x[0] == i[1]), None)
                if(end == 1079387396):
                    queue.append((i[0],i[1],str(float(now[4])+float(i[2])+float(heuristicData[indexNew][1])),i[2],str(float(i[2])+float(now[4]))))
                elif(end == 1737223506):
                    queue.append((i[0],i[1],str(float(now[4])+float(i[2])+float(heuristicData[indexNew][2])),i[2],str(float(i[2])+float(now[4]))))
                elif(end == 8513026827):
                    queue.append((i[0],i[1],str(float(now[4])+float(i[2])+float(heuristicData[indexNew][3])),i[2],str(float(i[2])+float(now[4]))))

    now = end
    num_visited = len(visited)
    dist = 0

    while now!=start:                                                # trace back from end to start
        #print(str(now)+"\n")
        for way in fromNode:                                         # if the dest. of a way is the end
            if int(way[1]) == now:
                path.insert(0,now)                                   # add it to path
                dist += float(way[3])                                # add the destination
                now = int(way[0])                                    # update the end
                break
    path.insert(0,start)                                             # insert start in the path
    return path,dist,num_visited
    # End your code (Part 4)
```

queue : (string startNode, string destNode, string g(destNode) + h(destNode), string lengthBetween2Node, string lengthFromStartToNode)

f(n) = g(n) + h(n) is in the queue and used when choosing the node for popping

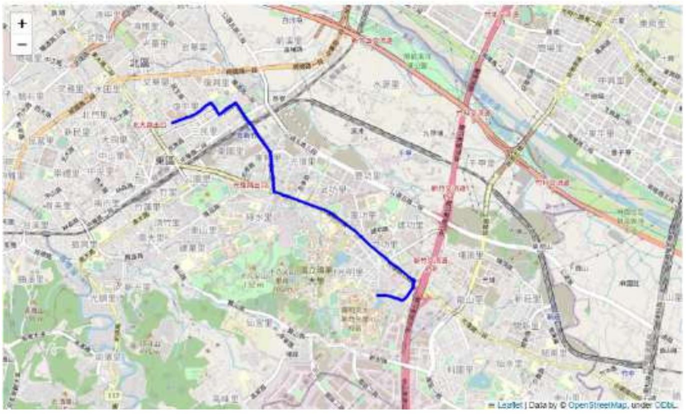
use the content in heuristic.csv directly (3 cases)

Part II. Results & Analysis (12%):

Test 1: 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: 4131
```

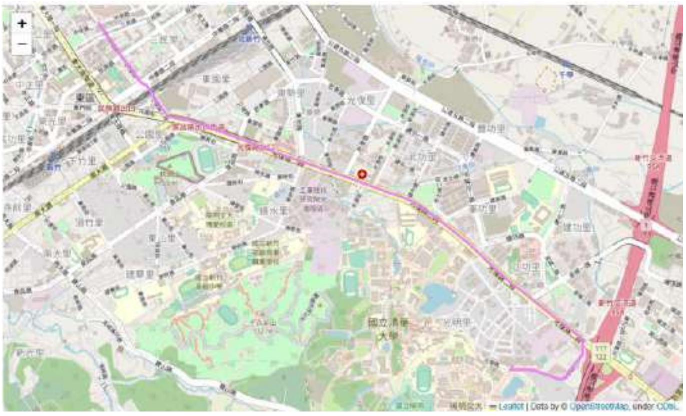


DFS (stack):

```
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: 4712
```

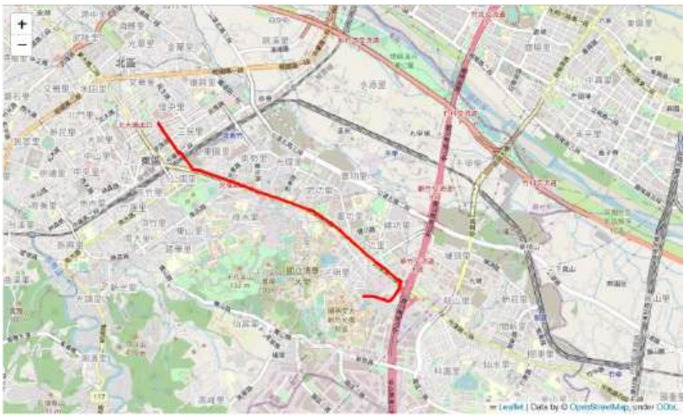
UCS :

```
The number of nodes in the path found by UCS: 89
Total distance of path found by UCS: 4367.8809999999985 m
The number of visited nodes in UCS: 5086
```



A* :

```
The number of nodes in the path found by A* search: 89
Total distance of path found by A* search: 4367.8809999999985 m
The number of visited nodes in A* search: 1319
```

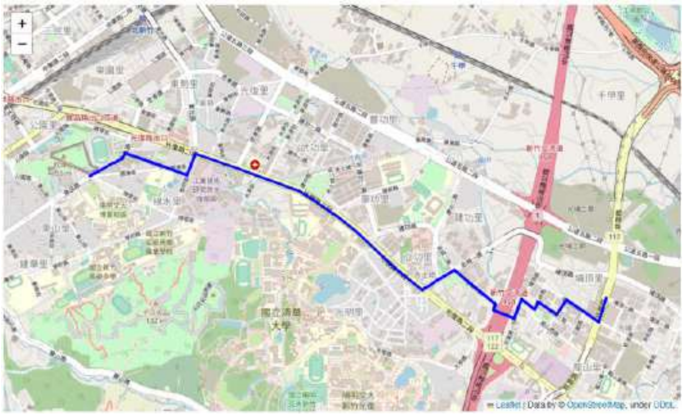


Test 2: from Hsinchu Zoo (ID: 426882161)

to COSTCO Hsinchu Store (ID: 1737223506)

BFS :

```
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: 4469
```

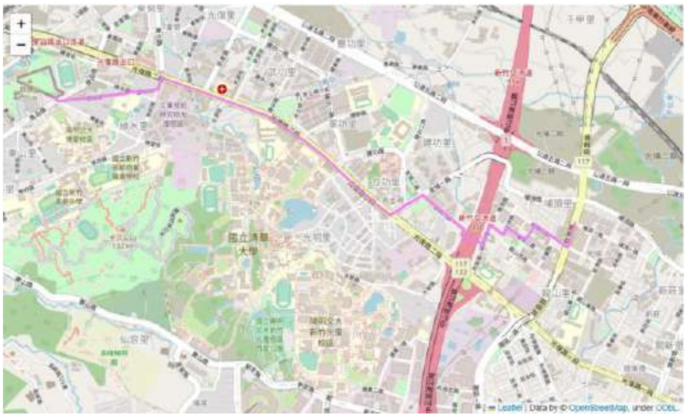


DFS (stack) :

```
The number of nodes in the path found by DFS: 930
Total distance of path found by DFS: 38752.307999999895 m
The number of visited nodes in DFS: 9366
```

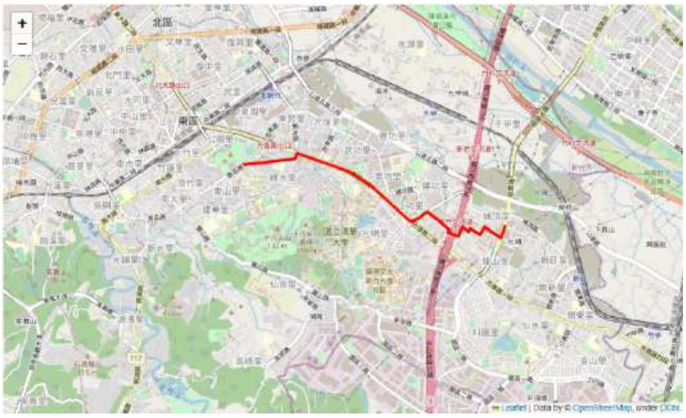
UCS :

```
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: 7213
```



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: 1172
```



Test 3: from National Experimental High School At Hsinchu Science Park (ID: 1718165260) to Nanliao Fighing Port (ID: 8513026827)

BFS :

```
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: 11217
```

DFS (stack) :

```
The number of nodes in the path found by DFS: 900
Total distance of path found by DFS: 39219.993000000024 m
The number of visited nodes in DFS: 2248
```

UCS :

```
The number of nodes in the path found by UCS: 288
Total distance of path found by UCS: 14212.413 m
The number of visited nodes in UCS: 11926
```

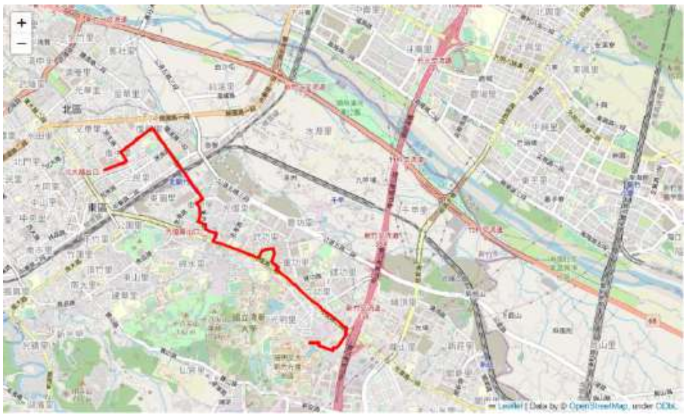
A* :

```
The number of nodes in the path found by A* search: 288
Total distance of path found by A* search: 14212.413 m
The number of visited nodes in A* search: 7073
```

Bonus part

Test 1:

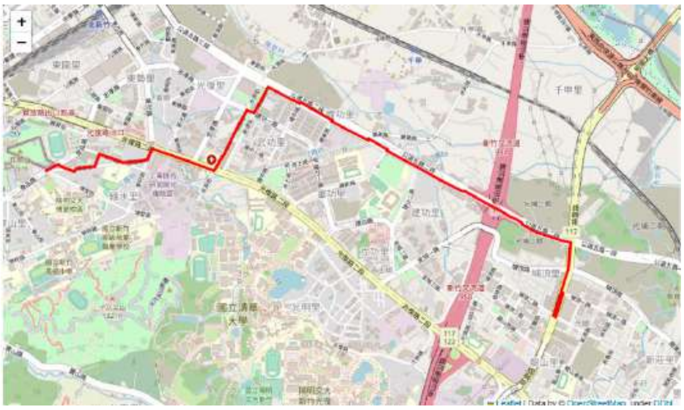
```
The number of nodes in the path found by A* search: 119
Total second of path found by A* search: 418.09889263002117 s
The number of visited nodes in A* search: 263
```



(multiplier = 2)

Test 2

```
The number of nodes in the path found by A* search: 70
Total second of path found by A* search: 348.73111329006076 s
The number of visited nodes in A* search: 130
```



(multiplier = 2)

Test 3:

```
The number of nodes in the path found by A* search: 186
Total second of path found by A* search: 792.2234071776189 s
The number of visited nodes in A* search: 227
```

(multiplier = 1)

code:

```python
def astar_time(start, end, multiplier):
    # Begin your code (Part 6)
    '''
    1. read from edges.csv
    2. use bfs to know how many edges a node is to the end
    3. use the length computed by the bfs in 2 as heuristic function to conduct A* search

    '''
    if end == 8513026827:
        heuristicFile = 'bfsHeuristic3.csv'
    elif end == 1737223506:
        heuristicFile = 'bfsHeuristic2.csv'
    else:
        heuristicFile = 'bfsHeuristic1.csv'
    data = []
    heuristicData = []
    queue = []
    visited = [start]
    fromNode = tuple()
    path = []
    with open(edgeFile, newline='') as csvfile:                          # read from csv
        reader = csv.reader(csvfile, delimiter=',', quotechar='"')
        next(reader)
        for row in reader:
            data.append(tuple(row))
    '''
    bfsqueue = []
    bfsvisited = [end]
    bfsqueue.append((end,0))
    while(len(bfsqueue)>0):                                              # loop & do bfs
        now = bfsqueue.pop(0)                                            # pop a point from queue
        heuristicData.append(now)
        bfsvisited.append(int(now[0]))                                       # add it to visited
        for i in data:
            if (int(i[1]) == now[0] and int(i[0]) not in bfsvisited and int(i[0]) not in [x[0] for x in bfsqueue]):
                bfsqueue.append((int(i[0]),now[1]+1))                        # add the dest. node into queue

    with open(heuristicFile, 'a', newline='') as csvfile:
        writer = csv.writer(csvfile)
        for row in heuristicData:
            writer.writerow(row)
    '''
    with open(heuristicFile, newline='') as csvfile:                         # read from csv
        reader = csv.reader(csvfile, delimiter=',', quotechar='"')
        next(reader)
        for row in reader:
            heuristicData.append(tuple(row))

    # astar search
    for i in data:                                                       # push all the node for start point
        if(int(i[0]) == start):
            indexNew = next((j for j, x in enumerate(heuristicData) if int(x[0]) == start), None)
            queue.append((i[0],i[1],str(float(i[2])/float(i[3])+multiplier*float(heuristicData[indexNew][1])),str(float(i[2])/float(i[3]))))

    while(1):                                                            # loop
        minindex = 0                                                     # choose the one with min length
        minis = float(queue[0][2])
        for i in range(len(queue)):
            if(float(queue[i][2])<minis):
                minis = float(queue[i][2])
                minindex = i

        now = queue.pop(minindex)                                        # pop the min one
        visited.append(int(now[1]))                                      # add the dest. of the way to visited
        fromNode = fromNode + (now,)                                     # add it to possible path

        if int(now[1])==end:                                             # break if end is found
            break

        count = 0
        for i in range(len(queue)):                                      # pop unnecessary element in priority queue
            if(queue[i-count][1]==now[1]):                               # (the point is visited)
                queue.pop(i-count)
                count+=1

        for i in data:                                                     # push element into priority queue
            if (i[0] == now[1] and int(i[1]) not in visited):
                indexNew = -1
                for j in range(len(heuristicData)):
                    if(heuristicData[j][0] == now[1]):
                        indexNew = j
                queue.append((i[0],i[1],str(float(now[3])+float(i[2])/float(i[3])+multiplier*float(heuristicData[indexNew][1])),str(float(i[2])/float(i[3]))))

    now = end
    num_visited = len(visited)
    time = 0

    while now!=start:                                                      # trace back from end to start
        #print(str(now)+"\n")
        for way in fromNode:                                               # if the dest. of a way is the end
            if int(way[1]) == now:
                path.insert(0,now)                                         # add it to path
                time += float(way[3])                                      # add the destination
                now = int(way[0])                                          # update the end
                break
    path.insert(0,start)                                                   # insert start in the path
    time = time*3600/1000

    return path,time,num_visited
    # End your code (Part 6)
```

heuristic function design & analysis:

my design:

f(x) = g(x) + h(x) = distance / timeLimit + multiplier * bfs result

I used the number of bfs path as heuristic function to implement A* search. When the bfs result shows that it take more step to reach the ending node, the bfs result will be bigger. (the multiplier is used to adjust the size of bfs result)

I add a multiplier into the function to fitune the bfs result with a multiplier. I've tried several multipliers for the function, but it seems that there isn't a particular multiplier that can make the three test cases fast.

I think it might be because of the design of my heuristic function. When the distance of the destination is far away, the bfs result will be absolutely huge comparing to the g(x), and if we have a big multiplier for that, the difference of heurstic function value will have a huge affect on the edge choice, which is not I wanted, so I set the multiplier to be 1 for test3.In contrast, if we have small multiplier for short distances, the h(x) wouldn't have appropriate affect on the choices, so I choosed the multiplier 1 for test1 and test2.

code explanation:

The commented part is to get the heuristic function with the process of bfs and save it to a .csv file so that you wouldn't need to do the bfs all the time. The other things are almost the same as A* search.

Part III. Question Answering (12%):

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

   I had a trouble finding the value in the heuristic data same as the popped node. In my opinion, using a loop to find the corresponding node is time comsuming, and it will make my code more ugly. So I look it up on the Internet and found some advanced python syntax like "index = next((j for j, x in enumerate(heuristicData) if x[0] == i[1]), None)" and some simple ways to do a particular task without using a loop (e.g. when I want to get first element of a tuple from a list of tuple, I can use "[x[0] for x in list]"),and it made my code a little easier to read.

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.

   Traffic lights.

   When you are driving, the most time-consuming thing is the red light. Most traffic lights have a red light for about 45 seconds, and some will have more than 1 minutes, e.g. the traffic light on road next to 西門町 has a 100 sec. red light. In one minute, if you have the speed of 40km/hr, you can move a distance of 666 meters, which is longer than most of the edges we used, so traffic lights is also a attribute that will affact the route finding result.

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 can be done by satelites, and radar or some mobile devices. For satelites, you can take a picture directly and contruct the map using several pictures. For radar or some mobil devices, if you have a lot of time or human resource, you can collect a lot of data by wandering on the street and put it on your device. You can construct the map by leveraging all the data collected.

   Localization can be done by GPS or a compass. With GPS, you can know exactly where you are and the direction you are heading toward, and with a compass, you can know the direction you are going.

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.

   ETA = Meal preparation time + Delivery time + Time for pickup and drop-off
   Meal preparation time :

   you have to wait for the restaurant when they are preparing the order you are delivering, so the time will be included in the ETA.

Delivery time :

you should consider several factors in delivering time. For example, you have to know the path with shortest time, traffic, red lights, the weather, or even there is a car accident or not.

Time for pickup and drop-off :

when the restaurant finsish making the food, the restaurant have to check whether the food is yours, so the pick-up process takes some time. For the drop-off, since you might have many orders at the same time, if you arrived the destination of the first order, and the buyer did not show up on time, you will have to wait to get paid, so the drop-off process will also take some time and you will have to add the time in the next buyer's ETA.