Homework 2: Route Finding

Part I. Implementation (6%):

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

<u>Part 1:</u>

```python
def bfs(start, end):
    # Begin your code (Part 1)

    #load edgeFile to adjacency list
    adj_list = {}
    with open(edgeFile) as f:
        csvreader = csv.reader(f)
        next(csvreader) #first row
        for row in csvreader:
            tmp = []
            if int(row[0]) in adj_list: #node exist in adjacency list
                tmp.extend(adj_list[int(row[0])]) #row[0]:start_node
            tmp.append([int(row[1]), float(row[2]), float(row[3])])
            #row[1]:end_node #row[2]:distance #row[3]:speed limit
            adj_list[int(row[0])] = tmp #adjacency list

    #bfs
    queue = Queue()
    visited = []
    parent = {}
    queue.put((0, start)) #(distance from start to node, node)
    visited.append(start)
    parent[start] = None
    path_found = False
    while not queue.empty():
        curr = queue.get() #curr[0]:distance curr[1]:current node
        if curr[1] == end: #current node is end
            path_found = True
            break
        if curr[1] in adj_list: #current node exist
            for next_node in adj_list[curr[1]]: #next_node[0]:node #next_node[1]:distance
                if next_node[0] not in visited:
                    queue.put((curr[0]+next_node[1], next_node[0]))
                    #put distance from start to next_node, next_node to queue
                    parent[next_node[0]] = curr[1] #parent of next_node
                    visited.append(next_node[0])

    #result
    path = [] #start to end
    dist = curr[0] #distance from start to end
    num_visited = len(visited)
    if path_found: #store path
        path.append(end) #end - start
        while parent[end] is not None:
            path.append(parent[end])
            end = parent[end]
        path.reverse() #reverse path to get start to end
    return path, dist, num_visited
    # load edgeFile to get adjacency list (start_node: end_node, distance, speed limit)
    # use bfs to find path, start to end distance, number visited
    # get path from parent list
    #raise NotImplementedError("To be implemented")
    # End your code (Part 1)
```

Load egdeFile in every part is same.

Part 2:

```python
def dfs(start, end):
    # Begin your code (Part 2)
    #load edgeFile to adj_list
    adj_list = {}
    with open(edgeFile) as f:
        csvreader = csv.reader(f)
        next(csvreader) #first row
        for row in csvreader:
            tmp = []
            if int(row[0]) in adj_list:#node exist in adjacency list
                tmp.extend(adj_list[int(row[0])]) #row[0]:start_node
            tmp.append([int(row[1]), float(row[2]), float(row[3])])
            #row[1]:end #row[2]:distance #row[3]:speed limit
            adj_list[int(row[0])] = tmp #adjacency list
    #dfs_stack
    stack = []
    visited = []
    parent = {}
    stack.append((0, start)) #(distance from start to node, node)
    visited.append(start)
    parent[start] = None
    path_found = False
    while stack:
        curr = stack.pop() #curr[0]:distance curr[1]:current node
        if curr[1] == end: #current node is end
            path_found = True
            break

        if curr[1] in adj_list: #current node exist
            for next_node in adj_list[curr[1]]: #next_node[0]:node #next_node[1]:distance
                if next_node[0] not in visited:
                    stack.append((curr[0]+next_node[1], next_node[0]))
                    parent[next_node[0]] = curr[1] #parent of next_node
                    visited.append(next_node[0])
    #result
    path = [] #start to end
    dist = curr[0] #distance from start to end
    num_visited = len(visited)
    if path_found: #store path
        path.append(end) #end - start
        while parent[end] is not None:
            path.append(parent[end])
            end = parent[end]
        path.reverse() #reverse path to get start to end
    return path, dist, num_visited
    # load edgeFile to get adjacency list (start_node: end_node, distance, speed limit)
    # use dfs to find path, start to end distance, number visited
    # get path from parent list
    #raise NotImplementedError("To be implemented")
    # End your code (Part 2)
```

Part 3:

```python
def ucs(start, end):
    # Begin your code (Part 3)
    #load edgeFile to adj_list
    adj_list = {}
    with open(edgeFile) as f:
        csvreader = csv.reader(f)
        next(csvreader) #first row
        for row in csvreader:
            tmp = []
            if int(row[0]) in adj_list: #node exist in adjacency list
                tmp.extend(adj_list[int(row[0])]) #row[0]:start_node
            tmp.append([int(row[1]), float(row[2]), float(row[3])])
            #row[1]:end_node #row[2]:distance #row[3]:speed limit
            adj_list[int(row[0])] = tmp #adjacency list

    #ucs
    queue = PriorityQueue() #use priority queue to priority smallest distance
    visited = []
    parent = {}
    distance = {}
    queue.put((0, start)) #(distance from start to node, node)
    distance[start] = 0
    visited.append(start)
    parent[start] = None
    path_found = False
    while not queue.empty():
        curr = queue.get() #curr[0]:distance curr[1]:current node

        if curr[1] == end: #current node is end
            path_found = True
            break
        if curr[1] in adj_list:
            for next_node in adj_list[curr[1]]:
                #none_visited or has smaller than distance
                if next_node[0] not in visited or curr[0]+next_node[1] < distance[next_node[0]]:
                    queue.put((curr[0]+next_node[1], next_node[0]))
                    #calculator start to next_node distance
                    distance[next_node[0]] = distance[curr[1]] + next_node[1]
                    parent[next_node[0]] = curr[1] #parent of next_node
                    if next_node[0] not in visited:
                        visited.append(next_node[0])

    #result
    path = [] #start to end
    dist = curr[0] #distance from start to end
    num_visited = len(visited)
    if path_found: #store path
        path.append(end) #end - start
        while parent[end] is not None:
            path.append(parent[end])
            end = parent[end]
        path.reverse() #reverse path to get start to end
    return path, dist, num_visited
    # load edgeFile to get adjacency list (start_node: end_node, distance, speed limit)
    # use usc to find path, start to end smallest distance, number visited
    # queue will be priority smallest distance from start to node
    # get path from parent list
    #raise NotImplementedError("To be implemented")
    # End your code (Part 3)
```

Part 4:

```python
def astar(start, end):
    # Begin your code (Part 4)
    #load edgeFile to adj_list
    adj_list = {}
    with open(edgeFile) as f:
        csvreader = csv.reader(f)
        next(csvreader) #first row
        for row in csvreader:
            tmp = []
            if int(row[0]) in adj_list: #node exist in adjacency list
                tmp.extend(adj_list[int(row[0])]) #row[0]:start_node
            tmp.append([int(row[1]), float(row[2]), float(row[3])])
            #row[1]:end_node #row[2]:distance #row[3]:speed limit
            adj_list[int(row[0])] = tmp #adjacency list

    #load heuristicFile
    with open(heuristicFile) as f:
        csvreader = csv.reader(f)
        ids = next(csvreader) #end node id
        for index in range(1, len(ids)): #get end column index
            if (int(ids[index]) == end): #column is end node
                break
        heu = {}
        for row in csvreader:
            heu[int(row[0])] = float(row[index]) #heuristic list

    #Astar
    queue = PriorityQueue() #use priority queue to priority smallest distance
    visited = []
    parent = {}
```

```python

    #Astar
    queue = PriorityQueue() #use priority queue to priority smallest distance
    visited = []
    parent = {}
    distance = {} #distance from start to node
    f = {}
    queue.put((0, start)) #(start to node + node to end distance, node)
    distance[start] = 0
    f[start] = 0
    parent[start] = None
    visited.append(start)
    path_found = None
    while not queue.empty():
        curr = queue.get()
        if curr[1] == end:
            path_found = True
            break
        if curr[1] in adj_list:
            for next_node in adj_list[curr[1]]:
                #none_visited or has smaller than f
                if next_node[0] not in visited or heu[next_node[0]]+next_node[1] + distance[curr[1]] < f[next_node[0]]:
                    queue.put((heu[next_node[0]]+next_node[1] + distance[curr[1]], next_node[0]))
                    f[next_node[0]] = heu[next_node[0]]+next_node[1] + distance[curr[1]]
                    # calculator total start to node and node to end distance
                    distance[next_node[0]] = distance[curr[1]] + next_node[1] #distance start to next_node
                    parent[next_node[0]] = curr[1] #next_node of parent
                    if next_node[0] not in visited:
                        visited.append(next_node[0])

    #result
    path = []
    dist = distance[end]
    num_visited = len(visited)

    if path_found: #store path
        path.append(end) #end - start
        while parent[end] is not None:
            path.append(parent[end])
            end = parent[end]
        path.reverse() #reverse path to get start to end
    return path, dist, num_visited
    # load edgeFile to get adjacency list (start_node: end_node, distance, speed limit)
    # load heuristicFile to get heuritic list of node to end distance (node: distance)
    # use a* to find path, start to end smallest distance, number visited
    # queue will be priority smallest f
    # f = distance + heuristic
    # get path from parent list
    #raise NotImplementedError("To be implemented")
    # End your code (Part 4)
```

Load heuristicFile in Part 4, Part 5 is same.

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: 4978.8820000000005 m

The number of visited nodes in BFS: 4403

DFS (STACK):

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

Total distance of path found by DFS: 75504.31499999983 m

The number of visited nodes in DFS: 5236

UCS:

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

Total distance of path found by UCS: 4367.881 m

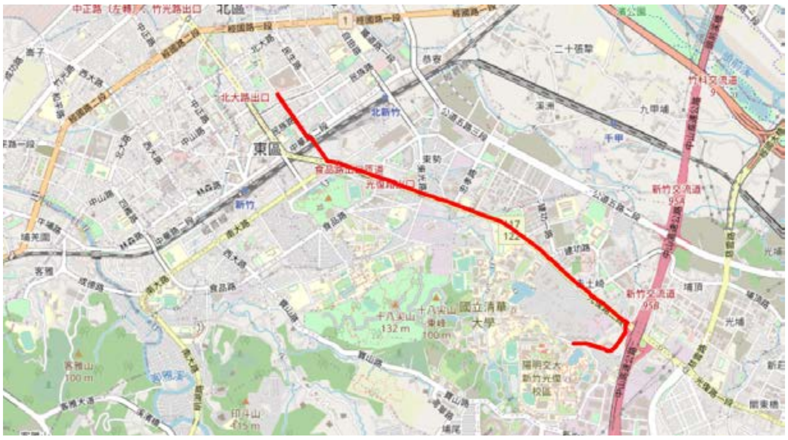
The number of visited nodes in UCS: 5158

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: 312



Test2: 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: 4215.521 m

The number of visited nodes in BFS: 4752

DFS (STACK):

The number of nodes in the path found by DFS: 930

Total distance of path found by DFS: 38752.30799999996 m

The number of visited nodes in DFS: 9616

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: 7318

Astar:

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: 1228

Test3: 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: 15442.395000000002 m

The number of visited nodes in BFS: 11266

DFS (STACK):

The number of nodes in the path found by DFS: 900

Total distance of path found by DFS: 39219.99299999996 m

The number of visited nodes in DFS: 2494

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: 11934

Astar:

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: 7247

● Analysis

BFS has smallest number of nodes in the path but total distance of path and number of visited nodes are larger.

DFS has largest number of nodes in the path, total distance of path and number of visited nodes.

UCS and A* search have same smallest number of nodes in the path and total distance of path. But A* number of visited nodes smaller than UCS.

Number of nodes in the path: BFS > A* = UCS > DFS

Total distance of path: A* = UCS > BFS > DFS

Number of visited nodes: A* > UCS > BFS > DFS

So A* search is best choice to find the shortest path.

Part III. Question Answering (12%):

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

I had error result in test 2 and test 3, I found out I didn't check from the current node to visited nodes that had shorter distance. In addition to unvisited nodes, I have added a distance test condition.

I don't know how to use JUPYTER NOTEBOOK, I use google search to find out about it.

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 lightsm, traffic jam, pedestrians, blocked or unworkable roads…

Traffic lights, traffic jam, passersby will affect speed and increase waiting time at each stop, or blocked or unworkable roads will affect the route.

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

Address

Coordinates

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.

Heuristic function will be time from shipper to seller and time from seller to buyer. Will set to 2 start nodes and 2 end nodes. First is shipper to seller location, second is seller location to buyer location. This will ensure the delivery and delivery process of the shipper.