Homework 2: Route Finding

學號: 111550129　姓名: 林彥亨

Part 1. Implementation

bfs.py

```python
    # Load the graph from the edges.csv file
    edge = {}
    with open(edgeFile, 'r') as file:
        reader = csv.DictReader(file)
        for row in reader:
            start_node = int(row['start']) # read the start
            end_node = int(row['end']) # read the end
            distance = float(row['distance']) # read the distance

            if start_node not in edge:
                edge[start_node] = [(end_node, distance)]
            else:
                edge[start_node].append((end_node, distance))
            if end_node not in edge:
                edge[end_node] = []
```

```python
    # Perform BFS to find the shortest path
    # initialize a dictionary 'parent' to save the parent node and a list 'queue'.
    # initialize a set to save the visited node.
    # set num_visited and done = 0.
    parent = {}
    visited = set()
    queue = [start]
    # queue = deque(start)
    visited.add(start)
    # queue = deque([(start, [start], 0)])  # current node, path, distance of path
    num_visited = 0
    done = 0


    while queue:
        # pop the first node in queue
        node = queue.pop(0)
        # node = queue.popleft()

        # check if reach the end
        if node == end:
            done = 1
            break
        # if not add 1 to the num_visited
        num_visited += 1
        # record the parent node into visited
        for i, j in edge[node]:
            if i not in visited:
                queue.append(i)
                visited.add(i)
                parent[i] = (node, j)
```

```
    # find the path and calculate the distance
    # set a list 'path' to save parent node.
    # set dist = 0
    path = []
    dist = 0
    if done:
        path.append(end)
        # calculate the distance until the 'start'
        while path[-1] != start:
            # add the distance from parent
            dist += parent[path[-1]][1]
            path.append(parent[path[-1]][0])
        path.reverse()
    return path, round(dist, 3), num_visited
```

dfs.py

```
    # Load the graph from the edges.csv file
    edge = {}
    with open(edgeFile, 'r') as file:
        reader = csv.DictReader(file)
        for row in reader:
            start_node = int(row['start']) # read the start
            end_node = int(row['end']) # read the end
            distance = float(row['distance']) # read the distance

            if start_node not in edge:
                edge[start_node] = [(end_node, distance)]
            else:
                edge[start_node].append((end_node, distance))
            if end_node not in edge:
                edge[end_node] = []
```

```
    # Implement DFS using a stack instead of a queue
    parent = {}
    stack = [start]  # Use a stack to store the nodes to visit
    visited = set()
    visited.add(start)
    num_visited = 0
    done = 0
    while stack:
        node = stack.pop()  # Pop from the stack to do DFS
        num_visited += 1
        if node == end:
            done = 1
            break
        for i, j in edge.get(node, []):  # Get the children of the current node, if any
            if i not in visited:
                stack.append(i)  # Push the node to the stack for DFS
                visited.add(i)
                parent[i] = (node, j)  # Store the parent and distance for path reconstruction
```

```
    # Find the path and calculate the distance
    path = []
    dist = 0
    if done:  # If the destination has been found
        current_node = end
        while current_node != start:
            path.append(current_node)
            parent_node, parent_distance = parent[current_node]
            dist += parent_distance
            current_node = parent_node
        path.append(start)  # Append the start node at the end
        path.reverse()  # Reverse the path to start from the beginning

    return path, round(dist, 3), num_visited
```

ucs.py

```
    # Load the graph from the edges.csv file
    edge = {}
    with open(edgeFile, 'r') as file:
        reader = csv.DictReader(file)
        for row in reader:
            start_node = int(row['start']) # read the start
            end_node = int(row['end']) # read the end
            distance = float(row['distance']) # read the distance

            if start_node not in edge:
                edge[start_node] = [(end_node, distance)]
            else:
                edge[start_node].append((end_node, distance))
            if end_node not in edge:
                edge[end_node] = []
```

```
    # set a 'parent' dictionary and a heapfied list 'heap'
    # set num_visited, done, dist = 0
    parent = {}
    heap = [(0, start, None)]
    heapq.heapify(heap)
    visited = set()
    num_visited = 0
    done = 0
    dist = 0


    while heap:
        # pop the first
        (cost, node, p) = heapq.heappop(heap)
        # check if reach the 'end'
        if node == end:
            done, dist = 1, cost
            parent[node] = p
            break
        # If the current node is unvisited, add the node to the 'visited'.
        # Then add 1 to the 'num_visited'.
        if node not in visited:
            num_visited += 1
            visited.add(node)
            parent[node] = p
            for i, j in edge[node]:
                heapq.heappush(heap, (cost + j, i, node))
```

```
    path = [] # initialize a list of path
    if done == 1 :
        path.append(end)
        while path[-1] != start:  # to read the first and check if it is start
            path.append(parent[path[-1]]) # add the parent id to the path
        path.reverse() # finally reverse the path
    return path, round(dist, 3), num_visited
```

astar.py

```
    # Load the graph from the edges.csv file
    edge = {}
    with open(edgeFile, 'r') as e_file:
        reader = csv.DictReader(e_file)
        for row in reader:
            start_node = int(row['start']) # read the start
            end_node = int(row['end']) # read the end
            distance = float(row['distance']) # read the distance

            if start_node not in edge:
                edge[start_node] = [(end_node, distance)]
            else:
                edge[start_node].append((end_node, distance))
            if end_node not in edge:
                edge[end_node] = []

    # Load the graph from the heuristic.csv file
    heur = {}
    with open(heuristicFile) as h_file:
        reader = list(csv.reader(h_file))
        idx = next((i for i in range(1, 4) if reader[0][i] == str(end)))
        # pop the 'title' row
        reader.pop(0)
        # record the data into the 'heur' dictionary
        for row in reader:
            heur[int(row[0])] = float(row[idx])
```

```python
    # implement the a*
    # initial a 'parent' dictionary and a priority queue 'queue'
    parent = {}
    heap = [(heur[start], start, None)]
    heapq.heapify(heap)
    # set a set 'visited' to save the visited node
    #set num_visited, done, dist = 0
    visited = set()
    num_visited = 0
    done = 0
    dist = 0


    while heap:
        # pop the first tuple in the priority queue 'queue'
        (cost, node, current) = heapq.heappop(heap)
        cost -= heur[node]

        if node not in visited:
            # if not add 1 to the num_visited
            num_visited += 1
            visited.add(node)
            parent[node] = current
            if node == end:
                done, dist = 1, cost
                break
            # explore the neighbor node
            for i, j in edge[node]:
                heapq.heappush(heap, (cost + j + heur[i], i, node))
```

```python
    # Initial a list 'path'
    path = []
    if done:
        path.append(end)
        while path[-1] != start:
            path.append(parent[path[-1]])
        path.reverse()
    return path, round(dist, 3), num_visited
```

Part 2. Results & Analysis

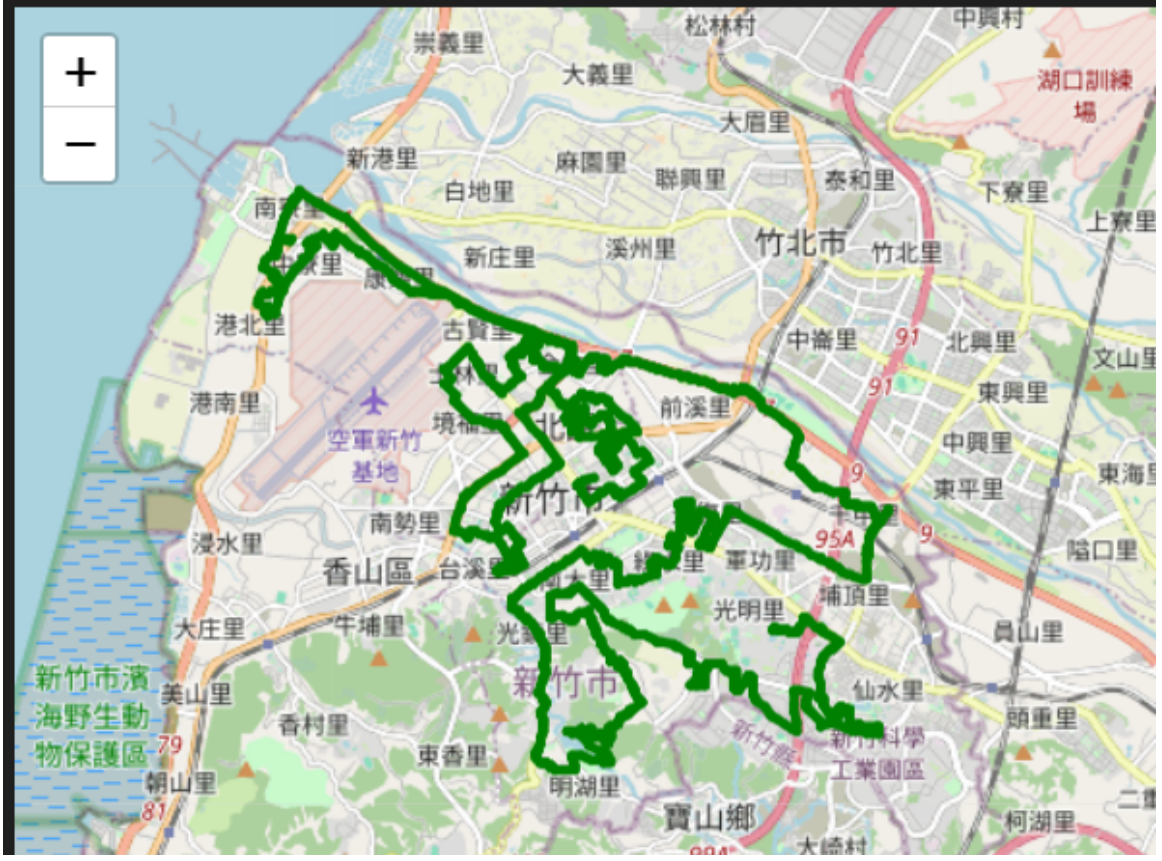
Test 1 : from National Yang Ming Chiao Tung University (ID: 2270143902) to Big City Shopping Mall (ID: 1079387396)

bfs.py:

```
The number of nodes in the path found by BFS: 88
Total distance of path found by BFS: 4978.882 m
The number of visited nodes in BFS: 4273
```

dfs.py:

```
The number of nodes in the path found by DFS: 1718
Total distance of path found by DFS: 75504.315 m
The number of visited nodes in DFS: 4712
```



ucs.py:

```
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: 5085
```

astar.py:

```
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
```

Test 2 : from Hsinchu Zoo (ID: 426882161) to COSTCO Hsinchu Store (ID: 1737223506)

bfs.py:

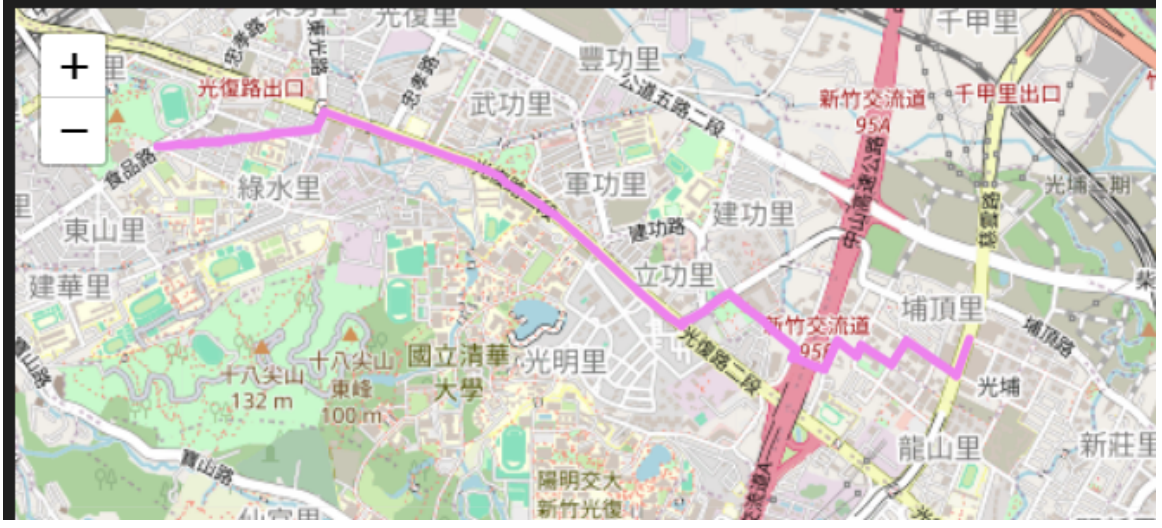
```
The number of nodes in the path found by BFS: 60
Total distance of path found by BFS: 4215.521 m
The number of visited nodes in BFS: 4606
```

dfs.py:

```
The number of nodes in the path found by DFS: 930
Total distance of path found by DFS: 38752.308 m
The number of visited nodes in DFS: 9366
```

ucs.py:

```
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: 7212
```



astar.py:

```
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.py:

```
The number of nodes in the path found by BFS: 183
Total distance of path found by BFS: 15442.395 m
The number of visited nodes in BFS: 11241
```

dfs.py:

```
The number of nodes in the path found by DFS: 900
Total distance of path found by DFS: 39219.993 m
The number of visited nodes in DFS: 2248
```

ucs.py:

```
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: 11925
```

astar.py:

```
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
```

Part 3. Question Answering

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

   A: First, the results of dfs.py in test cases, my results are quite larger than the expected answer that TA given, and the astar_time.py i still don't know how to tackle with this problem. Second, for myself, i never heard the astar and UCS algorithm. So, I spent a few time to understand the meaning and to do the python file. It is quite difficult for me. Third, to complete the python file it took me a lot of time complete the script.

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.

   A: I think the "traffic impact" could be one of the attribute. For example the area that may have higher rate of traffic accident, or the easy congestion areas. Hence, navigation systems can contribute to reduced the cost of

time to the traffic. Whatsmore, we could let the traffic flow more smooth and lead to lower fuel consumption when we traveled.

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

   A: For “mapping”, we can use the GPS data and satellite images to enhance the mapping of the navigation system in urban areas.

   For “loaclization”, we can use the GPS system in the smart phone or use multiple sensors to improve the accuracy of the localization.

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.

   A: We consider the following factors:

   - *T_drive* is the base driving time from the restaurant to the customer, calculated using the fastest known route under optimal conditions.
   - *F_traffic* is a traffic factor that represents the percentage increase in driving time due to current traffic conditions, where 0 is no traffic and 1 would double the base driving time.
   - *T_prep* is the estimated food preparation time, which could be a dynamic value reported by the restaurant at the time of the order.
   - *T_buffer* is a buffer time that accounts for variables like parking, waiting for the customer, weather conditions, and small unforeseen delays.

   Rationale:

   - **Base Driving Time** (*T_drive*) is calculated under ideal conditions to establish a baseline.
   - **Traffic Factor** (*F_traffic*) dynamically adjusts the driving time based on real-time traffic data.
   - **Preparation Time** (*T_prep*) reflects the restaurant's current workload and order complexity.
   - **Buffer Time** (*T_buffer*) adds a margin to account for last-mile variables and uncertainties.

   $ETA = T\_drive \times (1 + F\_traffic) + T\_prep + T\_buffer$

In this case, dynamic data that can be updated to reflect real-world conditions, thus providing more accurate ETAs.