HW2: Route Finding Report

110550093 蔡師睿

Part I. Implementation

Part 1: Breadth-first Search

```python
def bfs(start, to):
    # Begin your code (Part 1)
    """
    Read and process the data from the csv file.
    1. List the whole data from 'edges.csv'. Pop the first one, since it is the column name.
    2. In this case, I only need the first three of each row. Therefore, the first three
       data in each row are put in a dictionary called 'edge' to represent the adjacent list
       of this graph. Specifically, in my implementation, every node is a key with value being
       an empty list in dictionary. If there is a path in the graph, then add the tuple storing
       end ID and destination to the list where the start ID is its key.
    """
    edge = {}
    with open(edgeFile) as file:
        csvfile = list(csv.reader(file))
        csvfile.pop(0)
        for i in csvfile:
            num1, num2, distance = int(i[0]), int(i[1]), float(i[2])
            if num1 not in edge:
                edge[num1] = [(num2, distance)]
            else:
                edge[num1].append((num2, distance))
            if num2 not in edge:
                edge[num2] = []
    """
    Implement BFS.
    1. Initialize an empty dictionary called 'parent', a list with start ID called 'queue', a
       set to mark the visited node called 'visited', and 'num_visited' and 'isfind' to 0.
    2. Pop the element from the front of the list 'queue', add it to the set 'visited', and add
       1 to the 'num_visited'.
    3. Explore the neighbors of the popped node. For each unvisited neighbors, add them to the
       'visited' and to the 'queue', and record the popped node and the distance in the dictionary
       'parent'.
    4. Repeat steps 2-3 until the popped node is the end we expected, and then set 'isfind' to 1.
    """
    parent = {}
    queue = [start]
    visited = set()
    visited.add(start)
    num_visited, isfind = 0, 0
    while queue:
        node = queue.pop(0)
        if node == to:
            isfind = 1
            break
        num_visited += 1
        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.
    1. Initialize an empty list 'path' and 'dist' to 0.
    2. If the variable 'isfind' is 1, which means BFS finds a path from 'start' to 'end' successfully,
       use the dictionary 'parent' to find the parent of the node and calculate the diatance. The
       value from the 'parent' is a tuple storing distance and parent ID, so add them to the 'dist'
       and 'path', respectively.
    3. Repeat step 2 until we find the start ID.
    4. Return 'path', 'dist', and 'num_visited'.
    """
    path = []
    dist = 0
    if isfind:
        path.append(to)
        while path[-1] != start:
            dist += parent[path[-1]][1]
            path.append(parent[path[-1]][0])
        path.reverse()
    return path, dist, num_visited
    # End your code (Part 1)
```

Part 2: Depth-first Search

```
def dfs(start, end):
    # Begin your code (Part 2)
    """
    Read and process the data from the csv file.
    1. List the whole data from 'edges.csv'. Pop the first one, since it is the column name.
    2. In this case, I only need the first three of each row. Therefore, the first three
       data in each row are put in a dictionary called 'edge' to represent the adjacent list
       of this graph. Specifically, in my implementation, every node is a key with value being
       an empty list in dictionary. If there is a path in the graph, then add the tuple storing
       end ID and destination to the list where the start ID is its key.
    """
    edge = {}
    with open(edgeFile) as file:
        csvfile = list(csv.reader(file))
        csvfile.pop(0)
        for i in csvfile:
            num1, num2, distance = int(i[0]), int(i[1]), float(i[2])
            if num1 not in edge:
                edge[num1] = [(num2, distance)]
            else:
                edge[num1].append((num2, distance))
            if num2 not in edge:
                edge[num2] = []
```

```
    """
    Implement DFS using stack.
    1. Initialize an empty dictionary called 'parent', a list with start ID called 'stack', a
       set to mark the visited node ID called 'visited', and 'num_visited' and 'isfind' to 0.
    2. Pop the element from the back of the list 'stack', add it to the set 'visited', and add
       1 to the 'num_visited'.
    3. Explore the neighbors of the popped node. For each unvisited neighbors, add them to the
       'visited' and to the 'stack', and record the popped node and the distance in the dictionary
       'parent'.
    4. Repeat steps 2-3 until the popped node is the end we expected, and then set 'isfind' to 1.
    """
    parent = {}
    stack = [start]
    visited = set()
    visited.add(start)
    num_visited, isfind = 0, 0
    while stack:
        node = stack.pop()
        if node == end:
            isfind = 1
            break
        num_visited += 1
        for i, j in edge[node]:
            if i not in visited:
                stack.append(i)
                visited.add(i)
                parent[i] = (node, j)
```

```
    """
    Find the path and calculate the distance.
    1. Initialize an empty list 'path' and 'dist' to 0.
    2. If the variable 'isfind' is 1, which means DFS finds a path from 'start' to 'end' successfully,
       use the dictionary 'parent' to find the parent of the node and calculate the diatance. The
       value from the 'parent' is a tuple storing distance and parent ID, so add them to the 'dist'
       and 'path', respectively.
    3. Repeat step 2 until we find the start ID.
    4. Return 'path', 'dist', and 'num_visited'.
    """
    path = []
    dist = 0
    if isfind:
        path.append(end)
        while path[-1] != start:
            dist += parent[path[-1]][1]
            path.append(parent[path[-1]][0])
        path.reverse()
    return path, dist, num_visited
    # End your code (Part 2)
```

Part 3: Uniform Cost Search

```python
def ucs(start, end):
    # Begin your code (Part 3)
    """
    Read and process the data from the csv file.
    1. List the whole data from 'edges.csv'. Pop the first one, since it is the column name.
    2. In this case, I only need the first three of each row. Therefore, the first three
       data in each row are put in a dictionary called 'edge' to represent the adjacent list
       of this graph. Specifically, in my implementation, every node is a key with value being
       an empty list in dictionary. If there is a path in the graph, then add the tuple storing
       end ID and destination to the list where the start ID is its key.
    """
    edge = {}
    with open(edgeFile) as file:
        csvfile = list(csv.reader(file))
        csvfile.pop(0)
        for i in csvfile:
            num1, num2, distance = int(i[0]), int(i[1]), float(i[2])
            if num1 not in edge:
                edge[num1] = [(num2, distance)]
            else:
                edge[num1].append((num2, distance))
            if num2 not in edge:
                edge[num2] = []
    """
    Implement UCS.
    1. Initialize an empty dictionary called 'parent', a heapified list like priority queue called
       'heap', an empty set called 'visited', and 'num_visited', 'isfind' and 'dist' to 0.
    2. Pop out the first tuple in the 'heap'. The tuple contains three elements, which represents
       the distance from start node, the current node ID, and its parent.
    3. If the current node is unvisited, add the node to the 'visited' and add 1 to the 'num_visited'.
       According to its parent in the tuple, use 'parent' to record.
    4. Then, explore its neighbor nodes, and add the tuple storing the distance from start to the
       neighbor, ID of the neighbor, and current node ID to the 'heap'.
    5. Repeat steps 2-5 until the end node is found, and then set 'isfind' to 1 and 'dist' to the first
       element of the tuple.
    """
    parent = {}
    heap = [(0, start, None)]
    heapq.heapify(heap)
    visited = set()
    num_visited, isfind, dist = 0, 0, 0
    while heap:
        (cost, node, p) = heapq.heappop(heap)
        if node == end:
            isfind, dist = 1, cost
            parent[node] = p
            break
        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))
    """
    Find the path.
    1. Initialize an empty list 'path'.
    2. If the variable 'isfind' is 1, which means UCS finds a path from 'start' to 'end' successfully,
       use the dictionary 'parent' to find the parent of the node. The value from the 'parent' is parent
       ID, so add it to the 'path'.
    3. Repeat step 2 until we find the start ID.
    4. Return 'path', 'dist', and 'num_visited'.
    """
    path = []
    if isfind:
        path.append(end)
        while path[-1] != start:
            path.append(parent[path[-1]])
        path.reverse()
    return path, dist, num_visited
    # End your code (Part 3)

```

Part 4: A* search

```python
def astar(start, end):
    # Begin your code (Part 4)
    """
    Read and process the data from the csv files.
    1. List the whole data from 'edges.csv'. Pop the first one, since it is the column name.
    2. In this case, I only need the first three of each row. Therefore, the first three
       data in each row are put in a dictionary called 'edge' to represent the adjacent list
       of this graph. Specifically, in my implementation, every node is a key with value being
       an empty list in dictionary. If there is a path in the graph, then add the tuple storing
       end ID and destination to the list where the start ID is its key.
    3. List the whole data from 'heuristic.csv', and find which elements in the rows are wanted.
    4. Put all the wanted values in each row in the dictionary called 'heuristic'.
    """
    edge = {}
    heuristic = {}
    with open(edgeFile) as file1, open(heuristicFile) as file2:
        csv1, csv2 = list(csv.reader(file1)), list(csv.reader(file2))
        csv1.pop(0)
        for i in csv1:
            num1, num2, distance = int(i[0]), int(i[1]), float(i[2])
            if num1 not in edge:
                edge[num1] = [(num2, distance)]
            else:
                edge[num1].append((num2, distance))
            if num2 not in edge:
                edge[num2] = []
        idx = next((i for i in range(1, 4) if csv2[0][i] == str(end)))
        csv2.pop(0)
        for i in csv2:
            heuristic[int(i[0])] = float(i[idx])
    """
    Implement A* search.
    1. Initialize an empty dictionary called 'parent', a heapified list like priority queue called
       'heap', an empty set called 'visited', and 'num_visited', 'isfind' and 'dist' to 0.
    2. Pop out the first tuple in the 'heap'. The tuple contains three elements, which represents
       the sum of distance from start node and the cost from heuristic function, the current node ID,
       and its parent.
    3. If the current node is unvisited, add the node to the 'visited' and add 1 to the 'num_visited'.
       According to its parent in the tuple, use 'parent' to record.
    4. Then, explore its neighbor nodes, and add the tuple storing the sum of distance from start to the
       neighbor and the cost from heuristic function, ID of the neighbor, and current node ID to the 'heap'.
    5. Repeat steps 2-5 until the end node is found, and then set 'isfind' to 1 and 'dist' to the first
       element of the tuple.
    """
    parent = {}
    heap = [(heuristic[start], start, None)]
    heapq.heapify(heap)
    visited = set()
    num_visited, isfind, dist = 0, 0, 0
    while heap:
        (cost, node, p) = heapq.heappop(heap)
        cost -= heuristic[node]
        if node not in visited:
            num_visited += 1
            visited.add(node)
            parent[node] = p
            if node == end:
                isfind, dist = 1, cost
                break
            for i, j in edge[node]:
                heapq.heappush(heap, (cost + j + heuristic[i], i, node))
```

```
    """
    Find the path.
    1. Initialize an empty list 'path'.
    2. If the variable 'isfind' is 1, which means A* search finds a path from 'start' to 'end'
       successfully, use the dictionary 'parent' to find the parent of the node. The value from
       the 'parent' is parent ID, so add it to the 'path'.
    3. Repeat step 2 until we find the start ID.
    4. Return 'path', 'dist', and 'num_visited'.
    """
    path = []
    if isfind:
        path.append(end)
        while path[-1] != start:
            path.append(parent[path[-1]])
        path.reverse()
    return path, dist, num_visited
    # End your code (Part 4)
```

Part 6: A* search(time)

```
def astar_time(start, end):
    # Begin your code (Part 6)
    """
    Read and process the data from the csv files same as the part in 'astar.py'.
    Difference :
    1. Record the speed limit and maximal speed limit, change them from kh/hr to m/s. Then 'distance'
       divided by its speed limit is the edge cost.
    2. After finishing 'edges.csv', we obtain the maximal speed limit, so the new heuristic function
       is the estimated distance from heuristic divided by max speed limit, which is sure to be an
       admissible heuristic function since the new heuristic value is always smaller than the actual
       cost.
    """
    edge = {}
    heauristic = {}
    max_speed = 0
    with open(edgeFile) as file1, open(heuristicFile) as file2:
        csv1, csv2 = list(csv.reader(file1)), list(csv.reader(file2))
        csv1.pop(0)
        for i in csv1:
            num1, num2, distance, speed = int(i[0]), int(i[1]), float(i[2]), float(i[3])
            speed = speed * 1000 / 3600
            max_speed = max(max_speed, speed)
            if num1 not in edge:
                edge[num1] = [(num2, float(distance / speed))]
            else:
                edge[num1].append((num2, float(distance / speed)))
            if num2 not in edge:
                edge[num2] = []
        idx = next((i for i in range(1, 4) if csv2[0][i] == str(end)), None)
        csv2.pop(0)
        for i in csv2:
            heauristic[int(i[0])] = float(i[idx]) / max_speed
```

```
    """
    Implement A* search same as the part in 'astar.py'.
    """
    parent = {}
    heap = [(heauristic[start], start, None)]
    heapq.heapify(heap)
    visited = set()
    num_visited, isfind, time = 0, 0, 0
    while heap:
        (cost, node, p) = heapq.heappop(heap)
        cost -= heauristic[node]
        if node not in visited:
            num_visited += 1
            visited.add(node)
            parent[node] = p
            if node == end:
                isfind, time = 1, cost
                break
            for i, j in edge[node]:
                heapq.heappush(heap, (cost + j + heauristic[i], i, node))
    """
    Find the path same as the part in 'astar.py'.
    """
    path = []
    if isfind:
        path.append(end)
        while path[-1] != start:
            path.append(parent[path[-1]])
        path.reverse()
    return path, time, num_visited
    # End your code (Part 6)
```

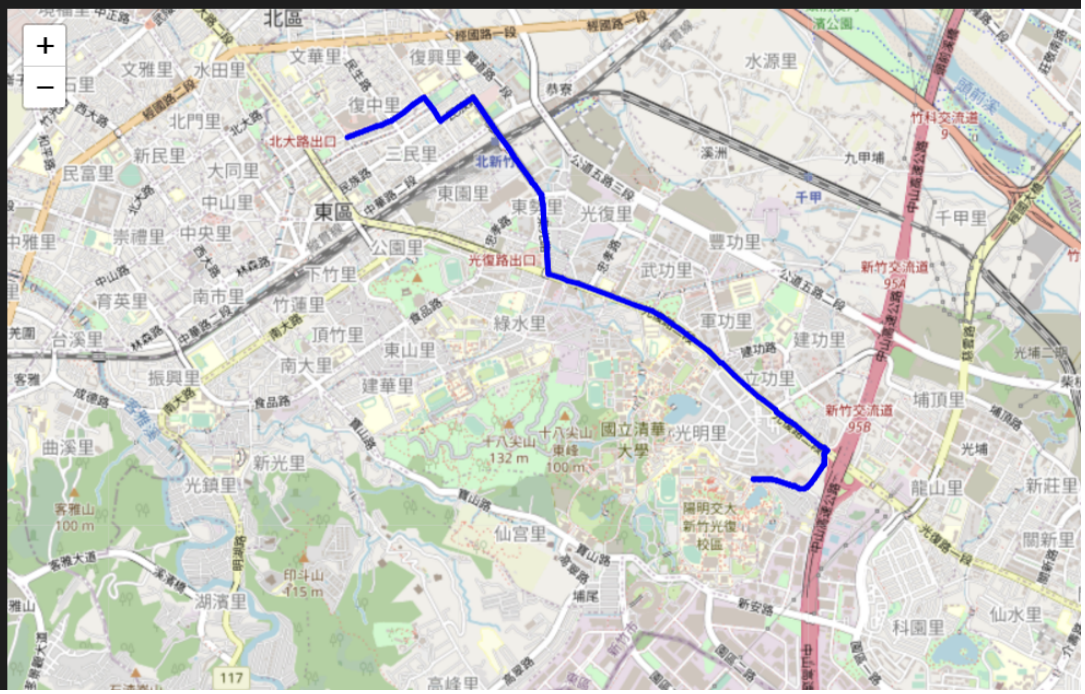
Part II. Results & Analysis

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: 4273
```

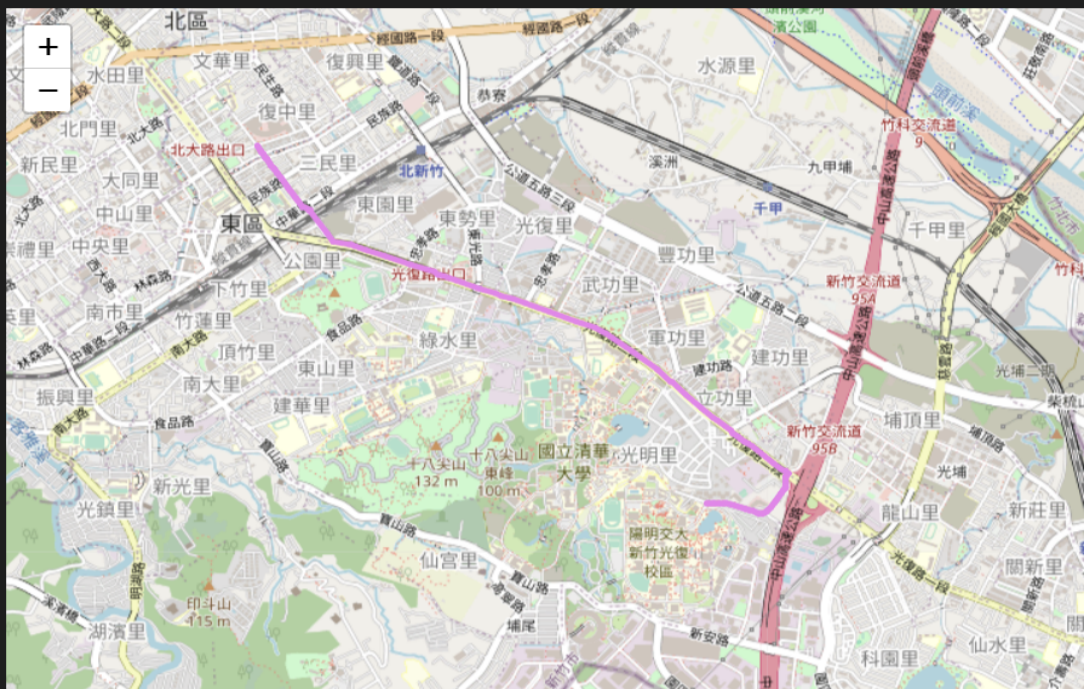


DFS(stack)

```
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: 4711
```

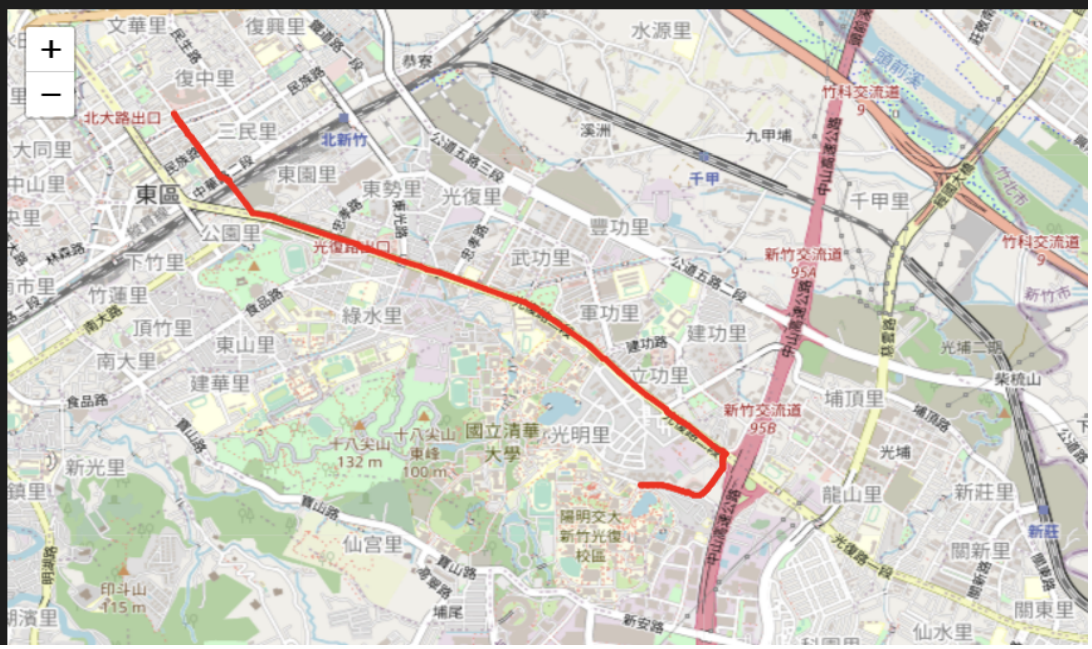
UCS

```
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: 5085
```



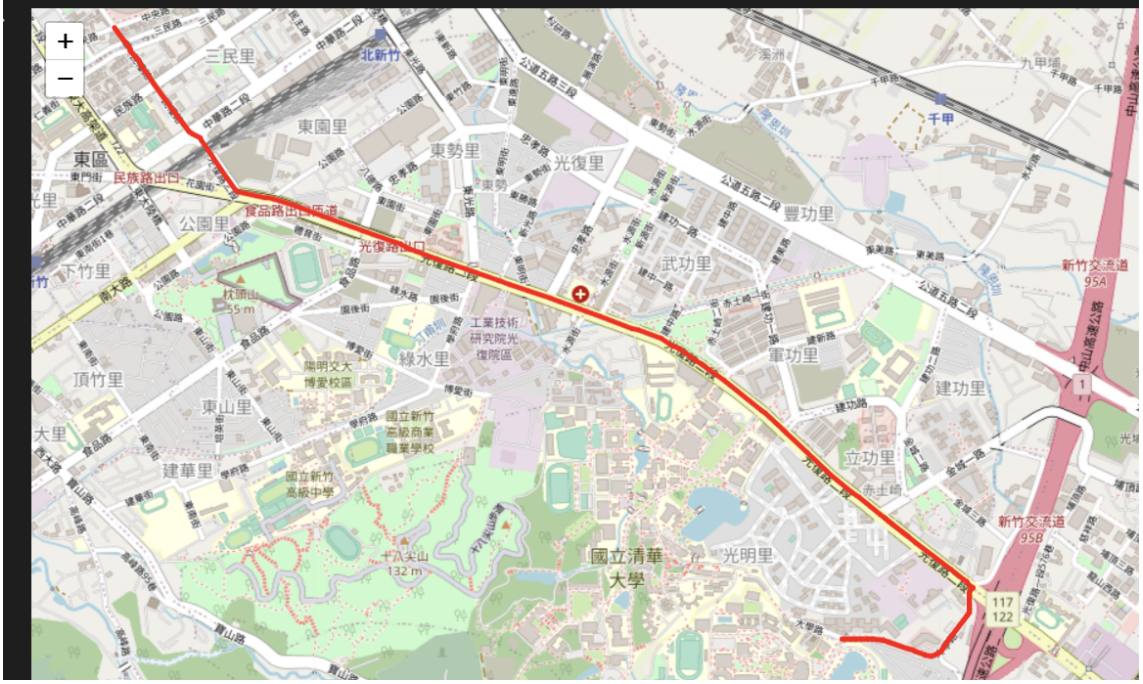
A* search

```
The number of nodes in the path found by A* search: 89
Total distance of path found by A* search: 4367.880999999999 m
The number of visited nodes in A* search: 261
```



A* search(time)

```
✓ 27.5s
The number of nodes in the path found by A* search: 89
Total second of path found by A* search: 320.8782316308318 s
The number of visited nodes in A* search: 5117
```

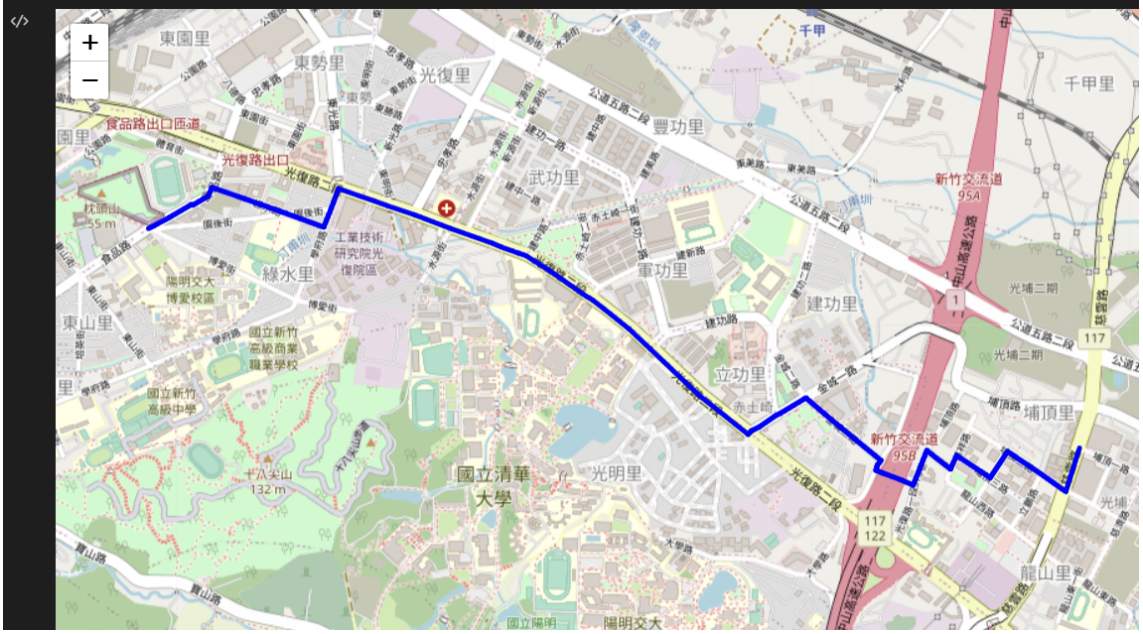


Test 2

From Hsinchu Zoo (ID:426882161) to COSTCO Hsinchu Store (ID:1737223506)

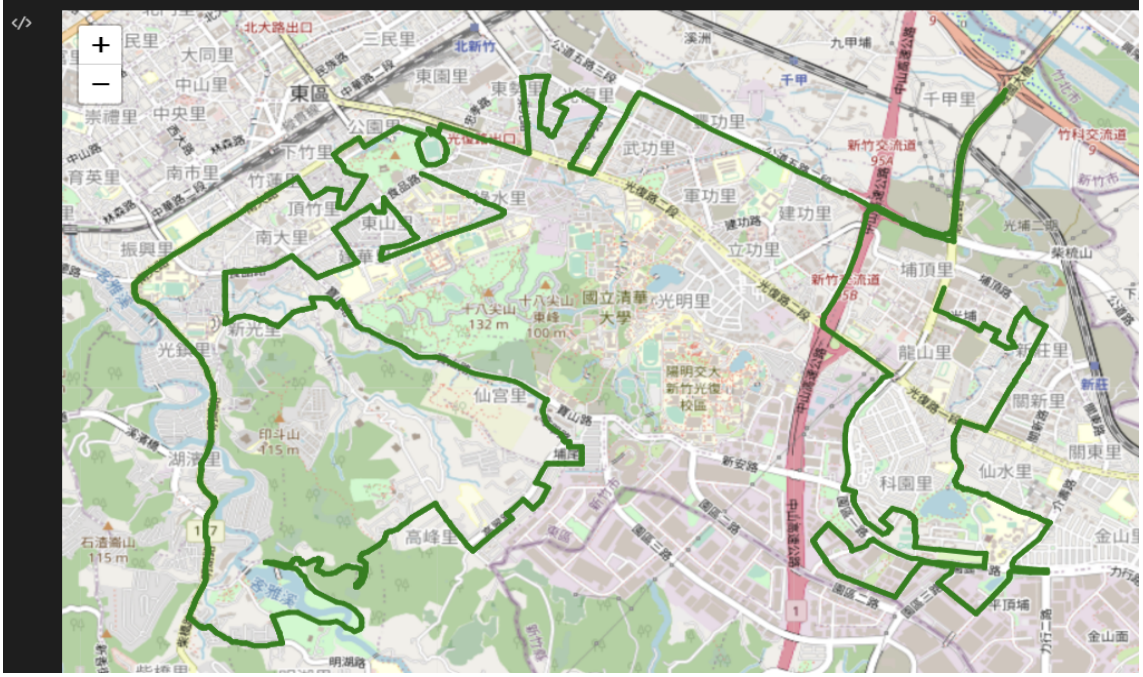
BFS

```
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: 4606
```



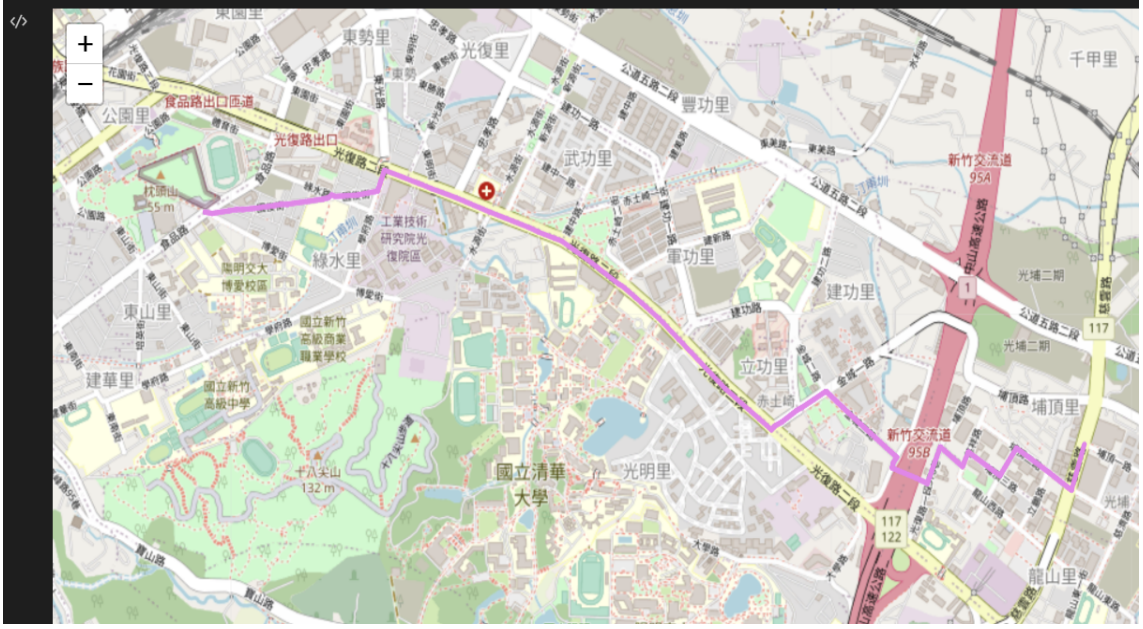
DFS(stack)

```
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: 9365
```



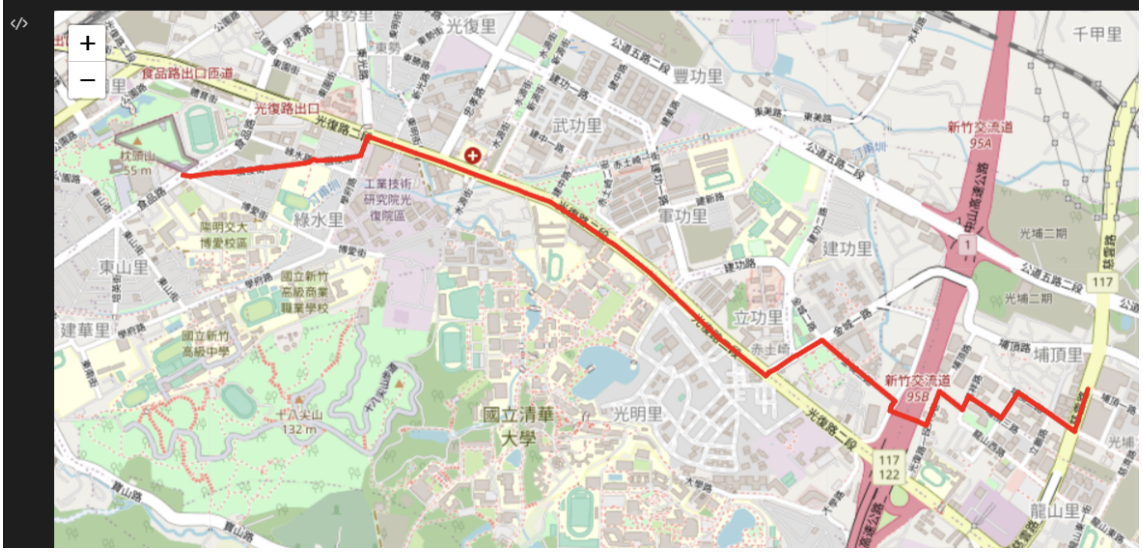
UCS

```
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: 7212
```

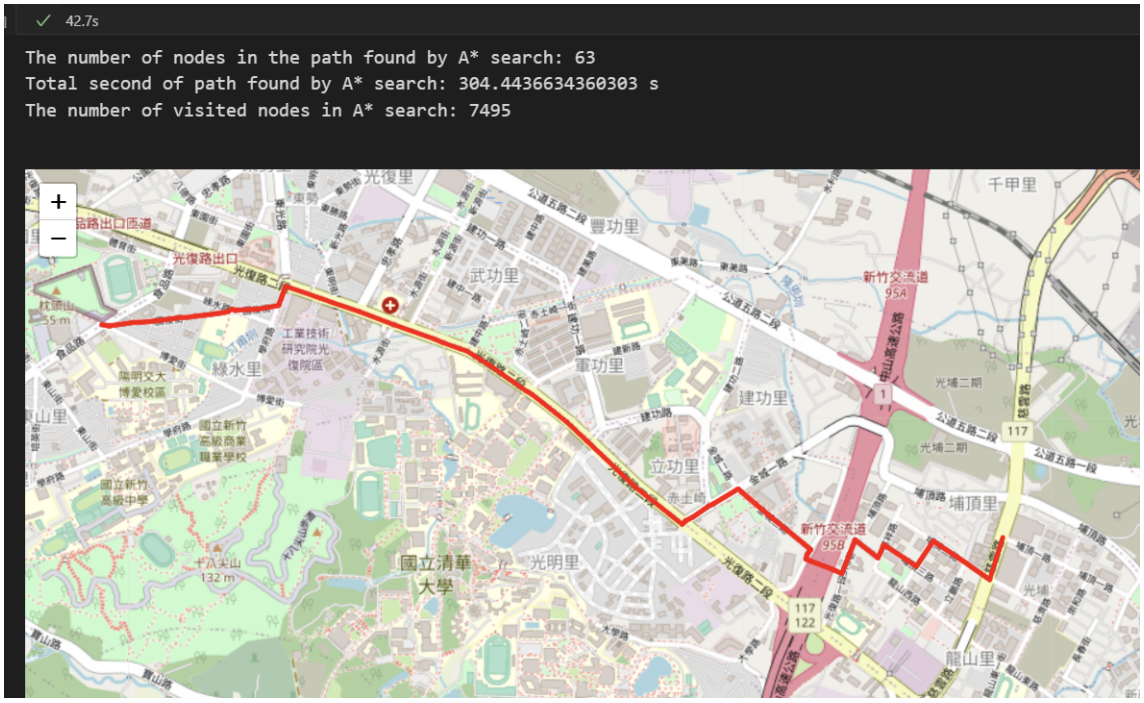


A* search

```
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
```



A* search(time)



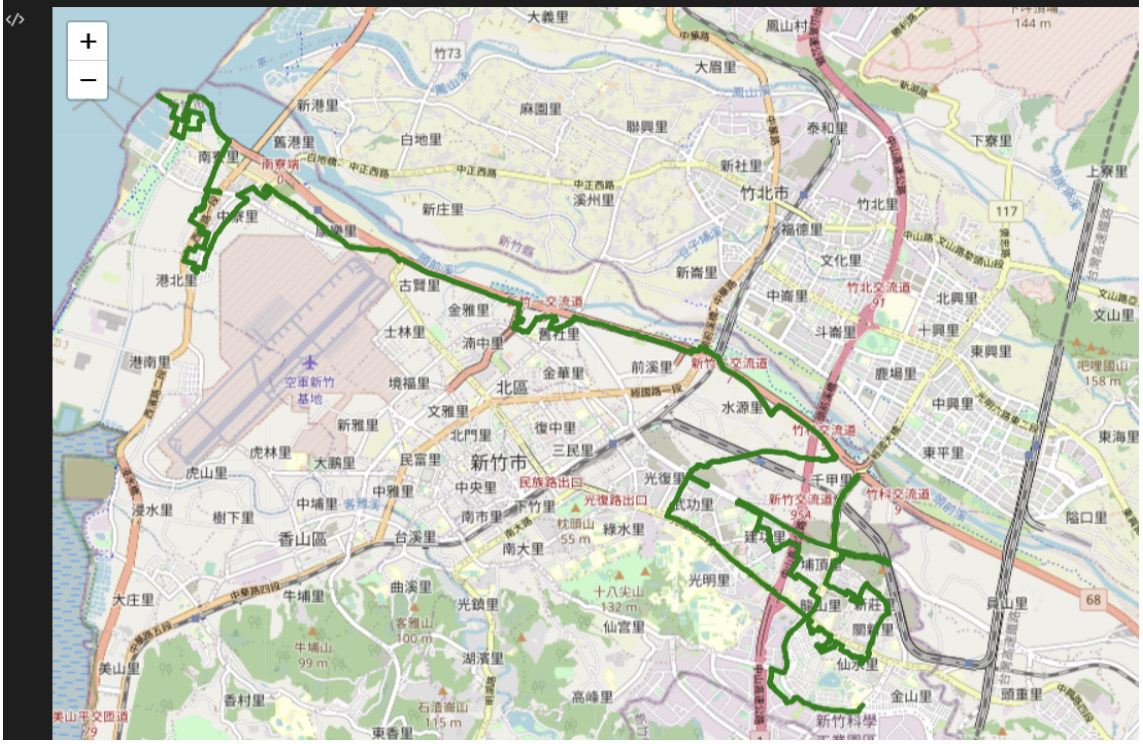
Test 3

From National Experimental High School At Hsinchu Science Park (ID:1718165260) to Nanliao Fishing Port (ID:8513026827)

BFS

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: 2247
```



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: 11925
```

A* search

```
The number of nodes in the path found by A* search: 288
Total distance of path found by A* search: 14212.413000000008 m
The number of visited nodes in A* search: 7073
```

A* search(time)

```
2m 36.5s
The number of nodes in the path found by A* search: 209
Total second of path found by A* search: 779.5279228368478 s
The number of visited nodes in A* search: 11552
```

Based on the three route finding tests, we can conclude that DFS is the worst way for route finding, because its results lead us to make a lengthy detour. In BFS, though the total distance found by BFS aren't the smallest, the results are nearly correct; hence, BFS is a more better method compared to DFS. The best solution for route finding is UCS and A* search. They provide the most reasonable and shortest route. However, as the map getting larger or the two points being farther, UCS needs more calculate time and node visited. As a result, A* search is the best and most efficiency algorithm for route finding. No wonder it is widely used in real world.

In bonus part finding the fastest path, my heuristic function is distance divided by the maximal speed limit. Theoretically, it is an admissible function since distance divided by the maximal speed limit is certain of being smaller than actual time cost and according the results above, it is proved that my heuristic function is definitely admissible. However, because of using the maximal speed limit as divisor, the $h(n)$ may be much smaller than the true cost from n to the goal in some n, which causes the A* search to visit more points. Thus, there is still a better heuristic function to make the A* search for time more efficient.

Part III. Question Answering

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

1. A little bit difference in DFS implementation will cause different result in route finding, For example, my DFS results are quite different from the given results.
2. In the beginning, I had no idea to deal with the data, since the data format isn't what I expected. It took time to think a good way for processing the data, but once the process was done, it became easier to start with.
3. Tracking path in Python took me for a while to think and design. Finally, I consider using dictionary is a more suitable way for record their parents.

2. Besides speed limit and distance, could you please come up with another attribute that is essential for route finding in the real world?

1. Traffic congestion : Sometimes when the traffic is heavy, it is wise and time-saving if we make a detour. Thus, traffic congestion is essential for route finding in the real world. To receive the traffic condition promptly or in advance, observation and prediction according to the past data are two ways to obtain the traffic data.

2. Traffic accidents : Traffic accidents will cause traffic jam, and that's why the traffic accidents should be considered in route finding as well. Since we cannot predict the accident in advance, we have to update the traffic condition and an alternative route as soon as the accident happened. Thus, the amount of calculation is larger and it's necessary to ensure the route finding algorithm being efficient.

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 : Use satellites or camera drones to take pictures and we can obtain the information we want.
Localization : Use GPS in cellphones to localize where we are in the map.

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.

In this case, we consider the following factors :
A. Preparation time for restaurant to prepare the order.
B. The time the delivery man take to ride to the restaurant. We can consider the estimated distance is the straight-line distance between restaurant and the delivery man, and the estimated time is the estimated distance divided by the delivery man's average speed.
C. The time the delivery man take to ride to the user's location. We can consider the estimated distance is the straight-line distance between the delivery man and destination, and the estimated time is the estimated distance divided by the delivery man's average speed.
D. Traffic condition. We can only estimate how long will it delay the delivery man.

The possible heuristic equation for ETA could be :
1. ETA when the delivery man haven't took the order yet : $max(A, B) + C + D$
2. ETA after the delivery man took the order : $C + D$