

# Spring 2022

## Introduction to Artificial Intelligence

Homework 2: Route Finding 109550159 李驊恩

**Part I. Implementation** (The explanation of my implementation is in the **notation** of my code.)

### Part 1: Breadth-first Search

```
1 import csv
2 import sys
3 edgeFile = 'edges.csv'
4
5 sys.setrecursionlimit(20000)
6
7 # Store the information of every nodes
8 # EDGES[node number] = [(reachable node #1, distance #1),(reachable node #2, distance #2)]
9 EDGES = {}
10 with open(edgeFile, newline='') as csvfile:
11
12     # read every edges then construct a graph
13     rows = csv.DictReader(csvfile)
14     for row in rows:
15         s = int(row['start'])
16         e = int(row['end'])
17         d = float(row['distance'])
18         if s in EDGES:
19             EDGES[s] += [(e, d)]
20         else:
21             EDGES[s] = [(e, d)]
22
23 # Variables have to be put outside of the function because we applied the recursive version
24 PATH = []
25 DIST = 0
26 VISITED = set()
27 BEGIN = True
28
29 def dfs(start, end):
30     # Begin your code (Part 2)
31     global PATH
32     global DIST
33     global VISITED
34
35     PATH.append(start)
36     VISITED.add(start)
37
38     # The end of the recursion
39     if(start == end):
40         return PATH, DIST, len(VISITED)+1
41
42     # Visit every node which 'start' can reach and we never tried
43     for nxt_node, d in EDGES[start]:
44
45         if not nxt_node in EDGES:
46             continue
47
48         # If the node have been visited, then skip.
49         if nxt_node in VISITED:
50             continue
51
52         # Keep going
53         DIST += d
54         ret_path, ret_dist, ret_num_visited = dfs(nxt_node, end)
55         if ret_path != []:
56
57             # Find a path successfully, and return the path
58             return ret_path, ret_dist, ret_num_visited
59         else:
60
61             # No way to go, try the next one
62             DIST -= d
63
64     # No way to go
65     PATH.pop()
66     return [], None, None
67
68     raise NotImplementedError("To be implemented")
69
70 # End your code (Part 2)
```

## Part 2: Depth-first Search (recursive)

```
1 import csv
2 import sys
3 edgeFile = 'edges.csv'
4
5 sys.setrecursionlimit(20000)
6
7 # Store the information of every nodes
8 # EDGES[node number] = [(reachable node #1, distance #1),(reachable node #2, distance #2)]
9 EDGES = {}
10 with open(edgeFile, newline='') as csvfile:
11     # read every edges then construct a graph
12     rows = csv.DictReader(csvfile)
13     for row in rows:
14         s = int(row['start'])
15         e = int(row['end'])
16         d = float(row['distance'])
17         if s in EDGES:
18             EDGES[s] += [(e, d)]
19         else:
20             EDGES[s] = [(e, d)]
21
22 # Variables have to be put outside of the function because we applied the recursive version
23 PATH = []
24 DIST = 0
25 VISITED = set()
26 BEGIN = True
27
28 def dfs(start, end):
29     # Begin your code (Part 2)
30     global PATH
31     global DIST
32     global VISITED
33
34     PATH.append(start)
35     VISITED.add(start)
36
37     # The end of the recursion
38     if(start == end):
39         return PATH, DIST, len(VISITED)+1
40
41     # Visit every node which 'start' can reach and we never tried
42     for nxt_node, d in EDGES[start]:
43
44         if not nxt_node in EDGES:
45             continue
46         # If the node have been visited, then skip.
47         if nxt_node in VISITED:
48             continue
49
50         # Keep going
51         DIST += d
52         ret_path, ret_dist, ret_num_visited = dfs(nxt_node, end)
53         if ret_path != []:
54
55             # Find a path successfully, and return the path
56             return ret_path, ret_dist, ret_num_visited
57         else:
58
59             # No way to go, try the next one
60             DIST -= d
61
62     # No way to go
63     PATH.pop()
64     return [], None, None
65     raise NotImplementedError("To be implemented")
66
67 # End your code (Part 2)
```

### Part 3: Uniform Cost Search

```
1  import csv
2  import heapq
3  edgeFile = 'edges.csv'
4
5
6  def ucs(start, end):
7      # Begin your code (Part 3)
8      edges = {}
9      with open(edgeFile, newline='') as csvfile:
10         rows = csv.DictReader(csvfile)
11         for row in rows:
12             s = int(row['start'])
13             e = int(row['end'])
14             d = float(row['distance'])
15             if s in edges:
16                 edges[s] += [(e, d)]
17             else:
18                 edges[s] = [(e, d)]
19
20         # dist[node number] = (distance from the starting point, the number of the former node)
21         dist = {}
22         Q = []
23
24         # Put it in the starting point first, every information in the queue is shown as (node number, distance, prev_node)
25         heapq.heappush(Q, (0, start, None))
26         while len(Q) > 0:
27             cur_dist, cur_node, prev_node = heapq.heappop(Q)
28
29             # exclude those have been visited, because we won't find a shorter path
30             if cur_node in dist:
31                 continue
32
33             # The distance of the starting point and the current node can be decided here
34             dist[cur_node] = (cur_dist, prev_node)
35             # Find the path
36             if cur_node == end:
37                 break
38
39             # Expand from the current node, and try the nodes that are connected to the current node
40             if not cur_node in edges:
41                 continue
42
43             # Exclude those nodes that have no way to go
44             for nxt_node, d in edges[cur_node]:
45
46                 # Exclude those have been visited, because we won't find a shorter path
47                 if nxt_node in dist:
48                     continue
49                 heapq.heappush(Q, (cur_dist + d, nxt_node, cur_node))
50
51         # Backtrace from the ending node
52         ret_path = [end]
53         while dist[ret_path[-1]][1] != None:
54             ret_path.append(dist[ret_path[-1]][1])
55         ret_dist = dist[ret_path[-1]][0]
56         ret_num_visited = len(dist)
57         return ret_path[::-1], ret_dist, ret_num_visited
58
59         raise NotImplementedError("To be implemented")
60
61     # End your code (Part 3)
```

## Part 4: A\* Search

```
1 import csv
2 import heapq
3 edgeFile = 'edges.csv'
4 heuristicFile = 'heuristic.csv'
5
6
7 def astar(start, end):
8     # Begin your code (Part 4)
9
10    # Store the information of every nodes.
11    # edges[node number] = [(reachable node #1, distance #1), (reachable node #2, distance #2)]
12    edges = {}
13    with open(edgeFile, newline='') as csvfile:
14
15        # read every edges then construct a graph
16        rows = csv.DictReader(csvfile)
17        for row in rows:
18            s = int(row['start'])
19            e = int(row['end'])
20            d = float(row['distance'])
21            if s in edges:
22                edges[s] += [(e, d)]
23            else:
24                edges[s] = [(e, d)]
25            if e in edges:
26                edges[e] += [(s, d)]
27            else:
28                edges[e] = [(s, d)]
29    # Construct a graph that represent the linear distance of each node to the destination.
30    # heuristic[node number] = the linear distance to the destination.
31    heuristic = {}
32    with open(heuristicFile, newline='') as csvfile:
33        # Read in the distance and construct a graph.
34        rows = csv.DictReader(csvfile)
35        for row in rows:
36            heuristic[int(row['node'])] = float(row[str(end)])
37
38    # dist[node number] =
39    # (The path's distance with the starting point + The linear distance with the destination)
40    dist = {}
41    Q = []
42    heapq.heappush(Q, (0+heuristic[start], start, None))
43    while len(Q) > 0:
44        cur_dist, cur_node, prev_node = heapq.heappop(Q)
45        # Exclude those have been visited, because we can't find a shorter path.
46        if cur_node in dist:
47            continue
48        dist[cur_node] = (cur_dist, prev_node)
49        # Find the path
50        if cur_node == end:
51            break
52        # Expand from the current node, and try the nodes that are connected to the current node.
53        for nxt_node, d in edges[cur_node]:
54            # Exclude those have been visited, because we won't find a shorter path.
55            if nxt_node in dist:
56                continue
57            heapq.heappush(Q, (cur_dist+d-heuristic[cur_node]+heuristic[nxt_node], nxt_node, cur_node))
58
59    # Backtrace from the ending node
60    ret_path = [end]
61    while dist[ret_path[-1]][1] != None:
62        ret_path.append(dist[ret_path[-1]][1])
63    ret_dist = dist[ret_path[-1]][0]
64    ret_num_visited = len(dist)
65    return ret_path[::-1], ret_dist, ret_num_visited
66    raise NotImplementedError("To be implemented")
67    # End your code (Part 4)
```

## Part 6: A\* search (fastest path)

```
1 import csv
2 import heapq
3 edgeFile = 'edges.csv'
4 heuristicFile = 'heuristic.csv'
5
6
7 def astar_time(start, end):
8     # Begin your code (Part 6)
9     # Store the information of every nodes.
10    # edges[node number] = [(reachable node #1, distance #1), (reachable node #2, distance #2)]
11    edges = {}
12    max_speed = 0
13    with open(edgeFile, newline='') as csvfile:
14        # read every edges then construct a graph
15        rows = csv.DictReader(csvfile)
16        for row in rows:
17            s = int(row['start'])
18            e = int(row['end'])
19            d = float(row['distance'])
20            t = d/(float(row['speed limit'])/3.6)
21            if s in edges:
22                edges[s] += [(e, d, t)]
23            else:
24                edges[s] = [(e, d, t)]
25                if float(row['speed limit']) > max_speed:
26                    max_speed = float(row['speed limit'])
27    # Construct a graph that represent the linear distance of each node to the destination.
28    # heuristic[node number] = the linear distance to the destination.
29    heuristic = {}
30    with open(heuristicFile, newline='') as csvfile:
31        # Read in the distance and construct a graph.
32        rows = csv.DictReader(csvfile)
33        for row in rows:
34            heuristic[int(row['node'])] = float(row['str(end)'])/(max_speed/3.6)
35    # times[node number] =
36    # (The path's travel time with the starting point + The linear distance with the destination /Maximum speed s/m
37    # , the previous node)
38    times = {}
39    Q = []
40    heapq.heappush(Q, (0+heuristic[start], start, None))
41    while len(Q) > 0:
42        cur_time, cur_node, prev_node = heapq.heappop(Q)
43        # Exclude those have been visited, because we can't find a shorter path.
44        if cur_node in times:
45            continue
46        times[cur_node] = (cur_time, prev_node)
47        # Find the path
48        if cur_node == end:
49            break
50        # Exclude those node that have no way to go
51        if not cur_node in edges:
52            continue
53        # Expand from the current node, and try the nodes that are connected to the current node.
54        for nxt_node, d, t in edges[cur_node]:
55            # Exclude those have been visited, because we won't find a shorter path.
56            if nxt_node in times:
57                continue
58            heapq.heappush(Q, (cur_time+t+heuristic[nxt_node], nxt_node, cur_node))
59
60    # Backtrace from the ending node
61    ret_path = [end]
62    while times[ret_path[-1]][1] != None:
63        ret_path.append(times[ret_path[-1]][1])
64    ret_time = times[ret_path[0]][0]
65    ret_num_visited = len(times)
66    return ret_path[::-1], ret_time, ret_num_visited
67    # Begin your code (Part 6)
```



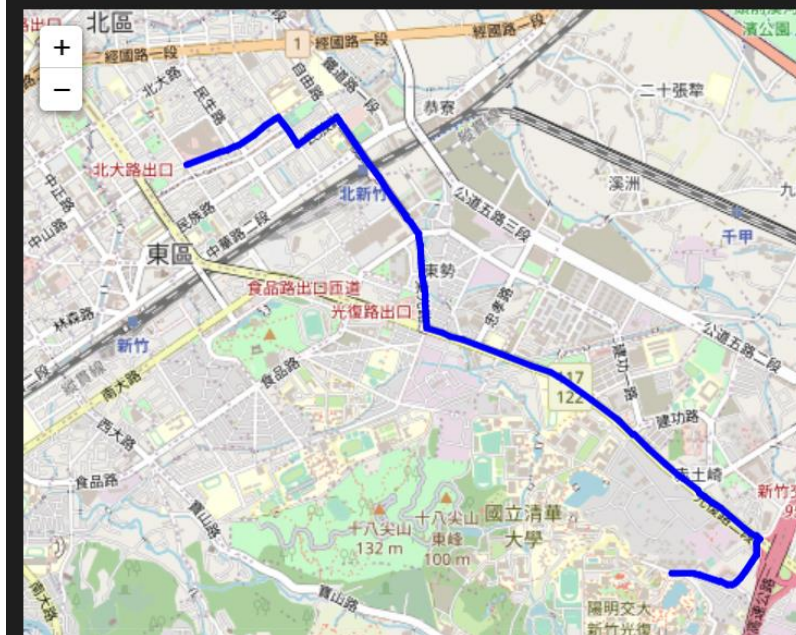
## Part II. Results & Analysis

### Part 5: Test my implementation

Test1: from National Yang Ming Chiao Tung University (ID: 2270143902) to Big City Shopping Mall (ID: 1079387396)

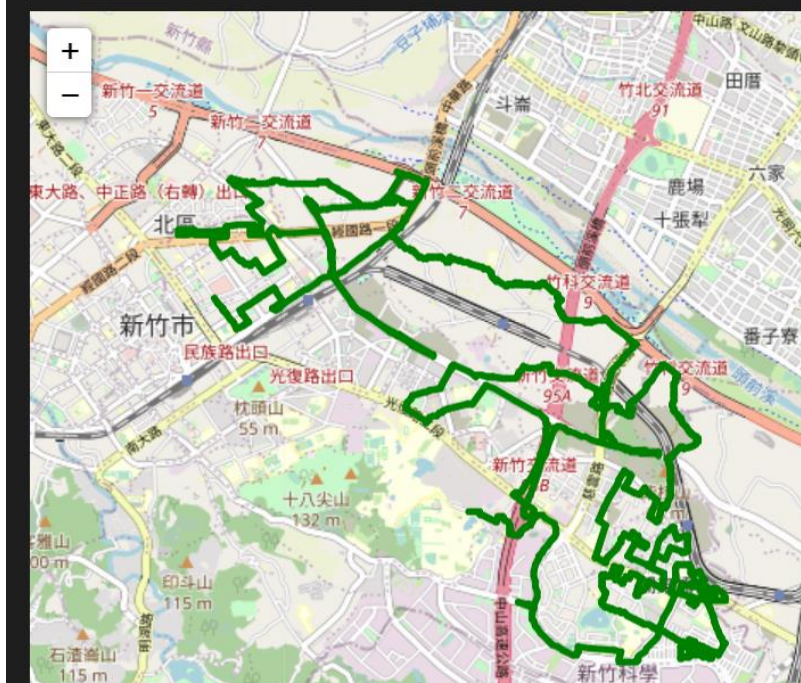
BFS :

```
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: 12054
```



DFS (recursion) :

```
The number of nodes in the path found by DFS: 1311  
Total distance of path found by DFS: 48954.321000000007 m  
The number of visited nodes in DFS: 3514
```



## Uniform Cost Search :

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: 5086

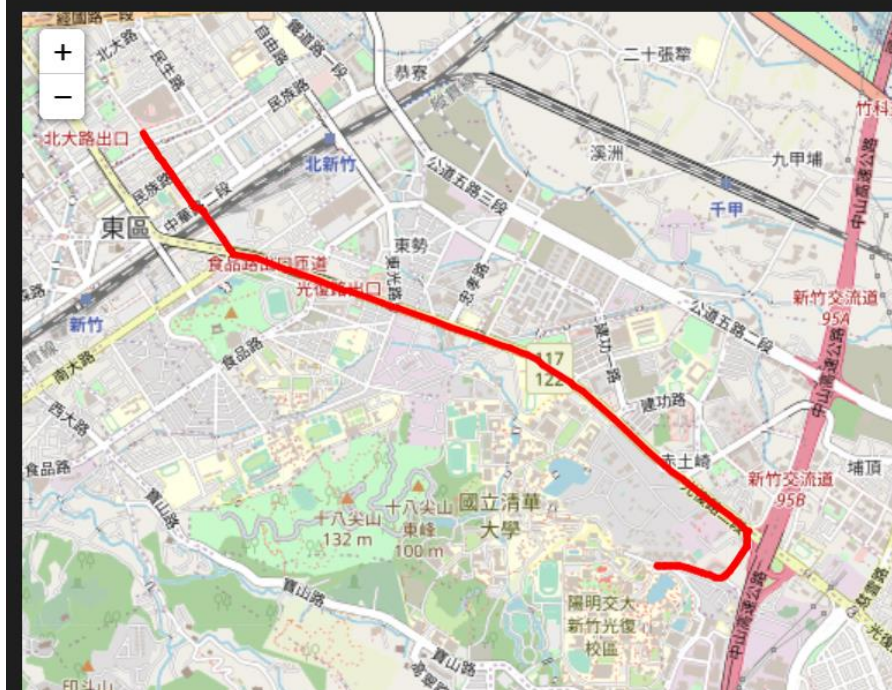


## A\* Search :

The number of nodes in the path found by A\* search: 85

Total distance of path found by A\* search: 4354.451000000003 m

The number of visited nodes in A\* search: 318

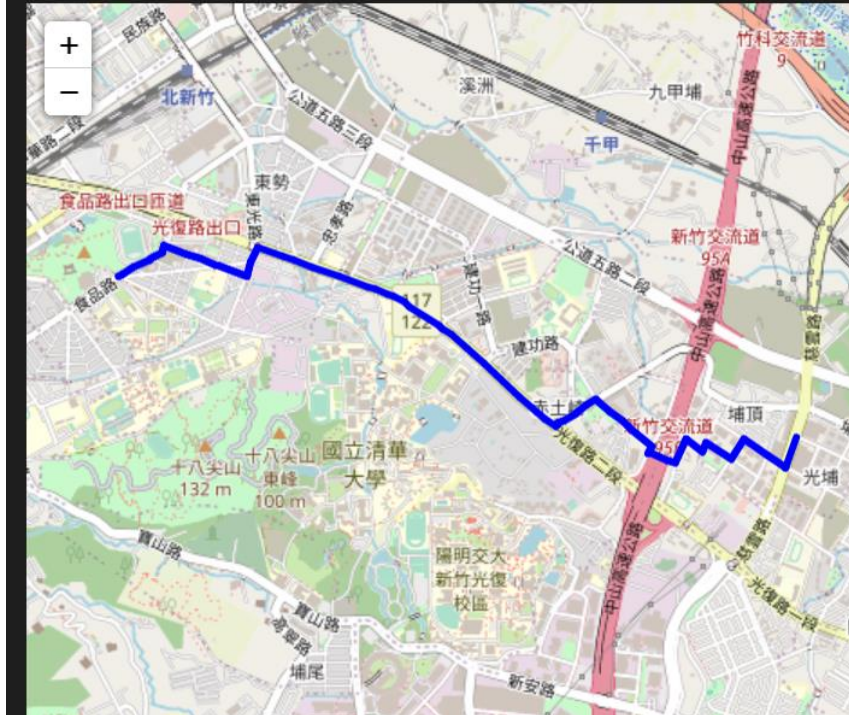




Test2: from Hsinchu Zoo (ID: 426882161) to COSTCO Hsinchu Store (ID: 1737223506)

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: 12054



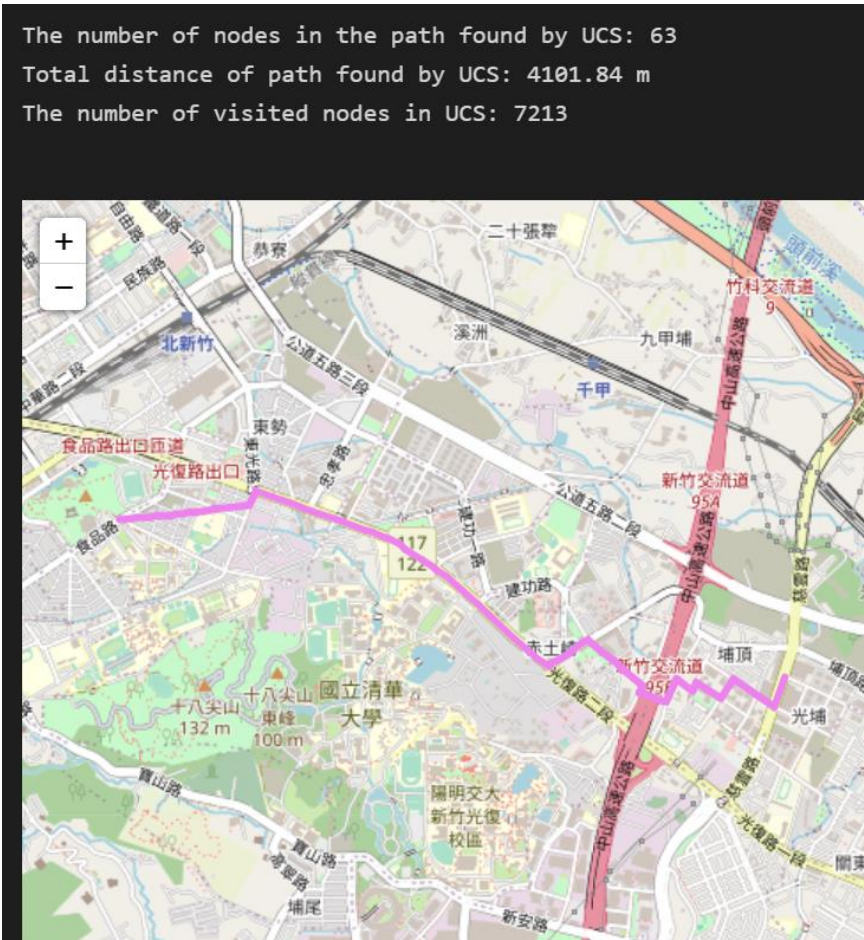
DFS (recursion) :

The number of nodes in the path found by DFS: 1016  
Total distance of path found by DFS: 43504.768999999935 m  
The number of visited nodes in DFS: 10610

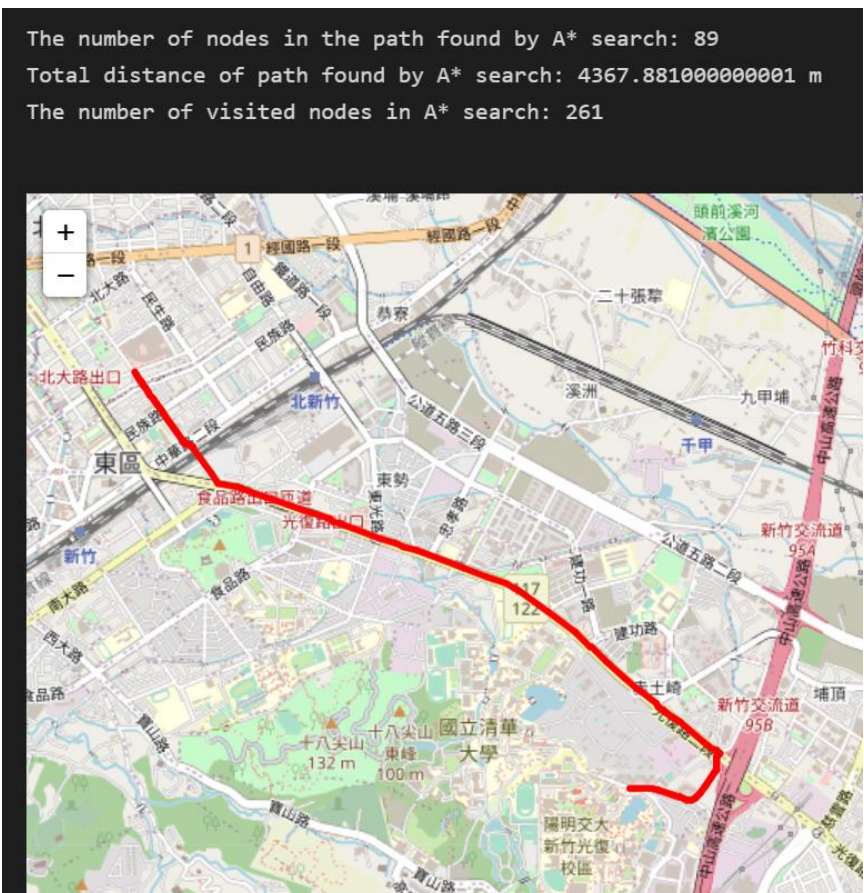




Uniform Cost Search :



A\* search :



Test3: from National Experimental High School At Hsinchu Science Park (ID: 1718165260) to Nanliao Fishing 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: 12054
```



DFS (recursion) :

```
The number of nodes in the path found by DFS: 2635  
Total distance of path found by DFS: 120440.442999999985 m  
The number of visited nodes in DFS: 7510
```





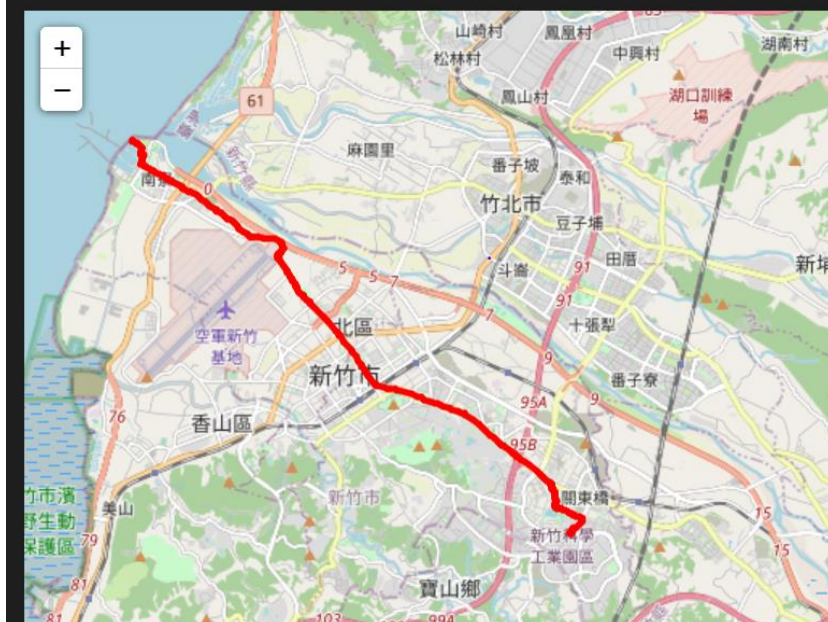
### Uniform Cost Search :

```
The number of nodes in the path found by UCS: 288
Total distance of path found by UCS: 14212.41299999997 m
The number of visited nodes in UCS: 11926
```

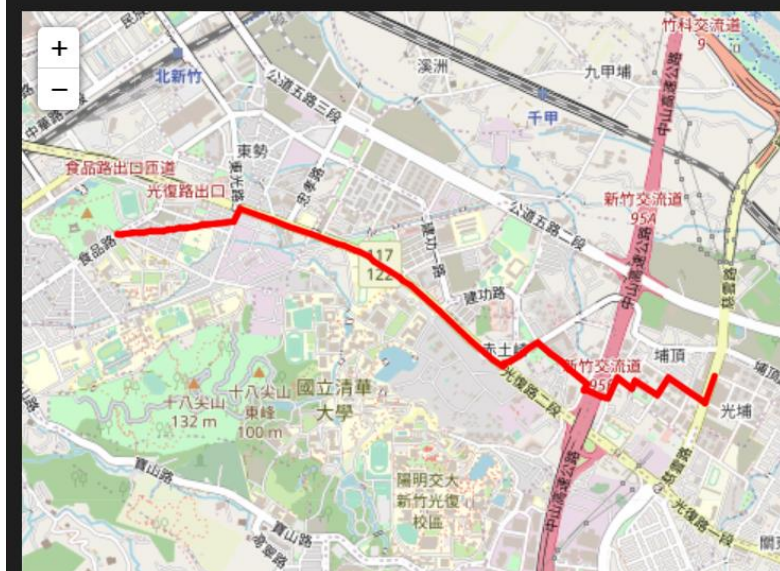


A\* search :

```
The number of nodes in the path found by A* search: 288
Total distance of path found by A* search: 14212.413000000002 m
The number of visited nodes in A* search: 7073
```







Test 3 :

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

```
The number of nodes in the path found by A* search: 209  
Total second of path found by A* search: 779.527922836848 s  
The number of visited nodes in A* search: 8458
```



**Compare the results with results obtained in Part 4 :**

Compared with the A\* search algorithm we have done in part 4, this program calculates the total second of path. Thus, we have to consider the speed limit of every road. In this program, the number of visited nodes when searching is far bigger than the one of part 4's.

### Part III. Question Answering

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

I had problem executing jupyter notebook in the beginning, my surrounding of my computer was not set up well so I can't execute it in Spyder successfully. Finally, I execute my programs on Visual Studio Code, and it work successfully.

**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.**

Besides speed limit and distance, I think time is also an attribute. There are different traffic flow between day and night. In the morning, people are likely to go to work, so traffic jam may occurs on the roads lead to these business district. So the route may no the shortest-distance path. Thus, we also have to consider what time it is to find a route in the real world.

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

For mapping, we can get materials of map online and turn the whole map into a directed weighted map. Intersections become nodes and every road is an edge, the speed limit is inversely proportional to the weight of an edge.

For localization, we can use GPS system to confirm our current position precisely. Sometime it has to be connected to the internet to execute the system.

**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.**

We can first construct a list that represents the distance of each intersection and the destination. Then as we heading for the destination, we calculate the time we need to reach the destination dynamically, which means we have to consider real-time condition of the path we will go through. It could contain many attributes such as weather, traffic flow, road accident etc.