Homework 2: Route Finding Report Template

Please keep the title of each section and delete examples. Note that please keep the questions listed in Part III.

Part I. Implementation (6%):

Please screenshot your code snippets of Part 1 ~ Part 4, and explain your implementation. Explanation inside the code.

Part 1. BFS

```python
import csv
edgeFile = 'edges.csv'

# to access the value from the file edges.csv
def graphfunc(efile):                          # can be used in other file
    graph={}                                   # dictionary to store a start node and its adjacent or end nodes
    distn={}                                   # dictionary to store distance between two nodes
    slimit={}                                  # dictionary to store the speed limit between two nodes
    with open(efile, 'r') as file:             # open and read the edges.csv file
        filereader = csv.reader(file)          # read the program using the csv library (csv.reader)
        rows = list(filereader)                # put result of the reader to list row
        for i in range(1,len(rows)):
            startnode = int(rows[i][0])        # to access the startnode of each row
            endnode = int(rows[i][1])
            distance = float(rows[i][2])
            speedlimit = float(rows[1][3])

            # if the startnode is not inside the graph, initialize the list and append the endnode
            # if the startnode is already inside the graph, just append the endnode to its list
            if startnode not in graph.keys():
                graph[startnode] = []
                graph[startnode].append(endnode)
            else:
                graph[startnode].append(endnode)

            # the same goes for distn and slimit
            if startnode not in distn.keys():
                distn[(startnode, endnode)] = distance              #key as tuple (a,b)
            if startnode not in slimit.keys():
                slimit[(startnode, endnode)] = speedlimit           #key as tuple (a,b)

    return graph, distn, slimit

```

```python
def bfs(start, end):
    # Begin your code (Part 1)
    # I started by finding the number of visited nodes to the path then lastly is the distance (visited -> path -> dist)
    graph, distn, slimit = graphfunc(edgeFile)

    visited = []                                        # list to keep track of node that has been visited
    queue = []                                          # we use queue for breadth-first search
    prev_node = {}                                      # to find the parent node (helps with finding the path)

    visited.append(start)                               # start by appending start node to both visited and queue
    queue.append(start)

    # traversing process: (exploring layer by layer)
    while queue:                                        # while queue is not empty
        x = queue.pop(0)                                # queue.pop(0) is removing the first value in the queue
        if x not in graph.keys(): continue              # means that it does not have any neighbour alr, so we skip it or continue
        for neighbour in graph[x]:                      # for all the adjacent nodes of x (which is neighbour here)
            if neighbour not in visited:                # if neighbour has not been visited yet
                prev_node[neighbour] = x                    # keep track of the parents of the node
                if end in visited:                          # if end is visited
                    while queue:
                        queue.pop()                                 # pop until the whole queue is empty
                    # biar krg satu
                    visited.pop(-1)                             # pop the current last node of visited
                    break                                   # break out of the for loop
                visited.append(neighbour)                   # append the neighbour to both visited and queue again
                queue.append(neighbour)                     # so we get to run through the whole graph by bfs

    # to find the path taken, we use backtrack (from end node to start node)
    path = []                                           # initialize path as list
    node = end                                          # let node be the end node
    while node != start:                                # so while the node is not start node yet, it will
        path.append(node)                                   # append the node to path list
        node = prev_node[node]                              # and backtrack to previous nodes
    path.append(start)                                  # after start node is found, we append start node to the list
    path.reverse()                                      # we reverse the whole path back, which it will return to normal (start to end)
    # path = [node for node in reversed(path)]          # same as previous line (in more complicated way)

    # to find the total distance of the path taken
    totaldist = 0
    # zip returns zip object(an iterator of tuples)
    # path[:-1] means all the sequence except the last
    # and path[:1] instead is all sequence except the first
    for i, j in zip(path[:-1], path[1:]):          # so the path pairs by taking (first, second) until (n-1, n) for (i, j) as distance between i
and j
        if (i, j) in distn.keys():
            totaldist += distn[(i,j)]              # add all the distances between (firstnode, secondnode) until the end node

    return path, totaldist, len(visited)
    # raise NotImplementedError("To be implemented")
    # End your code (Part 1)

if __name__ == '__main__':
    path, dist, num_visited = bfs(2270143902, 1079387396) #the location
    print(f'The number of path nodes: {len(path)}')
    print(f'Total distance of path: {dist}')
    print(f'The number of visited nodes: {num_visited}')
```

Part 2. DFS (Stack)

```
1  import csv
2  edgeFile = 'edges.csv'
3  from bfs import *                                           # graphfunc() from 'bfs.py' is used in line 7
4
5  def dfs(start, end):
6      # Begin your code (Part 2)
7      graph, distn, slimit = graphfunc(edgeFile)
8      visited = []                                            # list to keep track of node that has been visited
9      stack = []                                              # we use stack for depth-first search
10     prev_node = {}                                          # to find the parent node (helps with finding the path)
11
12     stack.append(start)                                     # start by appending start node to both visited and stack
13     visited.append(start)
14
15     # traversing process (iterative): (exploring layer by layer)
16     while stack:                                            # while stack is not empty
17         # pop vertex from start to visit next
18         # vertex = stack[-1]
19         vertex = stack.pop()                                # stack pop from last node which is first in last out
20         if vertex not in visited: visited.append(vertex)# we append the vertex to visited here instead
21         if vertex not in graph.keys(): continue             # if the vertex is not the start node, skip it or continue
22             # # bisa buat hasil recursive:
23             # for neighbour in reversed(graph[vertex]):
24             # hasil stack:
25         for neighbour in graph[vertex]:                     # for all the adjacent nodes of vertex (which is neighbour here)
26             if neighbour not in visited:                    # if the neighbour has not been visited yet
27                 prev_node[neighbour] = vertex                   # keep track of the parents of the node
28                 if end in visited:                              # if end is visited
29                     while stack:
30                         stack.pop()                                     # pop until the stack is empty
31                     # biar krg satu jg
32                     visited.pop(-1)                                 # pop the current last node of visited
33                     break                                       # break out of the for loop
34                 stack.append(neighbour)                         # only append to stack, unlike bfs to both queue and visited
35
36     # to find the path taken, we use backtrack (from end node to start node)
37     path = []                                               # initialize path as list
38     node = end                                              # let node be the end node
39     while node != start:                                    # so while the node is not start node yet, it will
40         path.append(node)                                       # append the node to path list
41         node = prev_node[node]                                  # and backtrack to previous nodes
42     path.append(start)                                      # after start node is found, we append start node to the list
43     path.reverse()                                          # we reverse the whole path back, which it will return to normal (start to end)
44
45     # to find the total distance of the path taken
46     totaldist = 0
47     # zip returns zip object(an iterator of tuples)
48     # path[:-1] means all the sequence except the last
49     # and path[:1] instead is all sequence except the first
50     for i, j in zip(path[:-1], path[1:]):                   # so the path pairs by taking (first, second) until (n-1, n) for (i, j) as distance
   between i and j
51         if (i, j) in distn.keys():
52             totaldist += distn[(i,j)]                       # add all the distances between (firstnode, secondnode) until the end node
53
54     return path, totaldist, len(visited)
55
56     raise NotImplementedError("To be implemented")
57     # End your code (Part 2)
58
59 if __name__ == '__main__':
60     path, dist, num_visited = dfs(2270143902, 1079387396)
61     print(f'The number of path nodes: {len(path)}')
62     print(f'Total distance of path: {dist}')
63     print(f'The number of visited nodes: {num_visited}')
```

Part 3. UCS

```python
import csv
edgeFile = 'edges.csv'
from bfs import *                                           # graphfunc() from 'bfs.py' is used in line 7

def ucs(start, end):
    # Begin your code (Part 3)
    graph, distn, slimit = graphfunc(edgeFile)
    queue = []
    visited = []
    distt = {}                                              # dictionary that stores distance from start node to current node
    prev_nodes = {}                                         # to find the parent node (helps with finding the path)

    queue.append(start)                                     # append start to queue
    distt[start] = 0                                        # distance of start node from start is 0
    prev_nodes[start] = start                               # parent of start node is start
    path = []                                               # the shortest path from start to end node
    totaldist = 0                                           # save the value of total distance from start to end node

    while queue:                                            # while queue is not empty
        node = None                                         # initialize node
        for v in queue:                                     # for the node inside queue
            if node == None or distt[v] < distt[node]:      # if node still None or distt from v is smaller than that of current node
                node = v                                    # let node be v

        if node == None: # ganti ini                        # path does not exist if node is end
            break

        # if current node is end node, backtrack from the current node to start node (find path and total distance along the path)
        if node == end:                                     # if current node is end
            totaldist = distt[node]                         # distt is already updated by their weight because of the for loop below
            while node != start:                            # while the node is not start node yet, it will
                path.append(node)                               # append the node to path list
                node = prev_nodes[node]                         # and backtrack to previous nodes
            path.append(start)                              # after start node is found, we append start node to the list
            path.reverse()                                  # we reverse the whole path back, which it will return to normal (start to end)
            break                                           # then break out of the loop

        for m in graph[node]:                               # for all neighbours of current node
            if m not in graph.keys(): continue              # if the neighbour is not part of the graph.keys or start, continue or skip it
            weight = distn[(node, m)]                       # weight is the distance from current node to its neighbours
            if m not in queue and m not in visited:         # if neighbour not in both queue and visited
                queue.append(m)                             # add to queue and mark node as its parent
                prev_nodes[m] = node
                distt[m] = distt[node] + weight             # dist from neighbour to start is dist from node to start plus neighbour's weight
            elif m in queue and m!= node:                   # neighbour is in queue and not the node itself
                if distt[m] > distt[node] + weight:         # check if it's quicker to visit the node or m
                    distt[m] = distt[node] + weight         # update the distance for m and also the parent
                    prev_nodes[m] = node

        if node in queue:
            queue.remove(node)                              # remove the node from queue
        visited.append(node)                                # and add it to visited
    visited.append(end)                                     # when node is endnode, it doesn't run until the end (leads to endnode not appended)

    return path, totaldist, len(visited)
    raise NotImplementedError("To be implemented")
    # End your code (Part 3)

if __name__ == '__main__':
    path, dist, num_visited = ucs(2270143902, 1079387396)
    print(f'The number of path nodes: {len(path)}')
    print(f'Total distance of path: {dist}')
    print(f'The number of visited nodes: {num_visited}')
```

Part 4. A*

```python
import csv
from sklearn import neighbors
edgeFile = 'edges.csv'
heuristicFile = 'heuristic.csv'
from bfs import *                                  # graphfunc() from 'bfs.py' is used in line 7

def heurfunc(lefile):                              # each id differs by test case (id1 for test1, etc)
    heur1 = {}                                     # heur1 as dictionary for node in each row to call its value (distance from node to id1)
    heur2 = {}
    heur3 = {}
    distid = {}                                    # dictionary to store the distance from each row differs by id
    with open(heuristicFile, 'r') as heurfile:     # open and read the heuristic.csv file
        heurreader = csv.reader(heurfile)          # read the program using the csv library (csv.reader)
        rows = list(heurreader)                    # put result of the reader to list row
        for i in range(1, len(rows)):
            node = int(rows[i][0])
            distid1 = float(rows[i][1])            # store heuristic or straight-line distance from the row by id1
            distid2 = float(rows[i][2])
            distid3 = float(rows[i][3])

            if node not in heur1.keys():           # initialize and append the heuristic distance as the value of dictionary
                heur1[node] = []
                heur1[node].append(distid1)

            if node not in heur2.keys():
                heur2[node] = []
                heur2[node].append(distid2)

            if node not in heur3.keys():
                heur3[node] = []
                heur3[node].append(distid3)

    return heur1, heur2, heur3

def astar(start, end):
    # Begin your code (Part 4)
    graph, distn, slimit = graphfunc(edgeFile)
    heur1, heur2, heur3 = heurfunc(heuristicFile)

    opened = []                                # list of nodes which have been visited, but neighbors havent all been inspected
    closed = []                                # list of nodes which have been visited and neigbors have been inspected
    distt = {}                                 # current distances from start node to all other nodes
    prev_nodes = {}                            # to find the parent node (helps with finding the path)

    opened.append(start)                       # append start to opened
    distt[start] = 0                           # distance of start node from start is 0
    prev_nodes[start] = start                  # parent of start node is start
    path = [] # pindah                         # the shortest path from start to end node
    totaldist = 0                              # save the value of total distance from start to end node

    while len(opened) > 0:                     # while opened is not empty
        node = None

        # f(n) = g(n) + h(n)
        # g(n): value of shortest path from start node to node n (here is distt)
        # h(n): heuristic approx. of the node's value (here is heur)

        # find node with lowest value of f(n)
        for v in opened:                       # for the node inside opened
            if (start, end) == (2270143902, 1079387396):        # for test1, thus use heur1
                if node == None or distt[v] + heur1[v][0] < distt[node] + heur1[node][0]:
                    node = v
            elif (start, end) == (426882161, 1737223506):       # for test2, thus use heur2
                if node == None or distt[v] + heur2[v][0] < distt[node] + heur2[node][0]:
                    node = v
            elif (start, end) == (1718165260, 8513026827):      # for test3, thus use heur3
                if node == None or distt[v] + heur3[v][0] < distt[node] + heur3[node][0]:
                    node = v
```

```python
        if node == None: # ganti ini                    # path does not exist if n is end
            break

        # if current node is end node, backtrack from the current node to start node (find path and total distance along the path)
        if node == end:                                 # if current node is end
            totaldist = distt[node]                     # distt is already updated by their weight because of the for loop below
            while node != start:                        # while the node is not start node yet, it will
            # while prev_nodes[node] != node:           # while parent of the node still node the node itself
                path.append(node)                           # append the node to path list
                node = prev_nodes[node]                     # and backtrack to previous nodes
            path.append(start)                          # after start node is found, we append start node to the list
            path.reverse()                              # we reverse the whole path back, which it will return to normal (start to end)
            break                                       # then break out of the loop

        for m in graph[node]:                           # for all neighbors of current node
            if m not in graph.keys(): continue          # if the neighbour is not part of the graph.keys or start, continue or skip it
            weight = distn[(node, m)]                   # weight is the distance from current node to its neighbours
            if m not in opened and m not in closed:     # if neighbour not in opened and closed list
                opened.append(m)                        # add to opened list and mark node as its parent
                prev_nodes[m] = node
                distt[m] = distt[node] + weight         # dist from neighbour to start is dist from node to start plus neighbour's weight
            else:
                if distt[m] > distt[node] + weight:     # check if it's quicker to visit the node or m
                    distt[m] = distt[node] + weight     # update the distance for m and also the parent
                    prev_nodes[m] = node
                    if m in closed:                     # if neighbour in the closed list
                        closed.remove(m)                # remove it and append to opened list
                        opened.append(m)

        if node in opened: # nambah ini
            opened.remove(node) # uncommen ini          # remove current node from opened list
        closed.append(node)                             # append th enode to closed list
    closed.append(end)                                  # when node is endnode, it doesn't run until the end (leads to endnode not appended)

    return path, totaldist, len(closed)
    raise NotImplementedError("To be implemented")
    # End your code (Part 4)

if __name__ == '__main__':
```

Part II. Results & Analysis (12%):

Please screenshot the results.

1. 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: 4978.8820000000005 m
The number of visited nodes in BFS: 4273
```

- DFS (stack):

```
The number of nodes in the path found by DFS: 1232
Total distance of path found by DFS: 57208.987000000045 m
The number of visited nodes in DFS: 4210
```

- 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: 5077
```

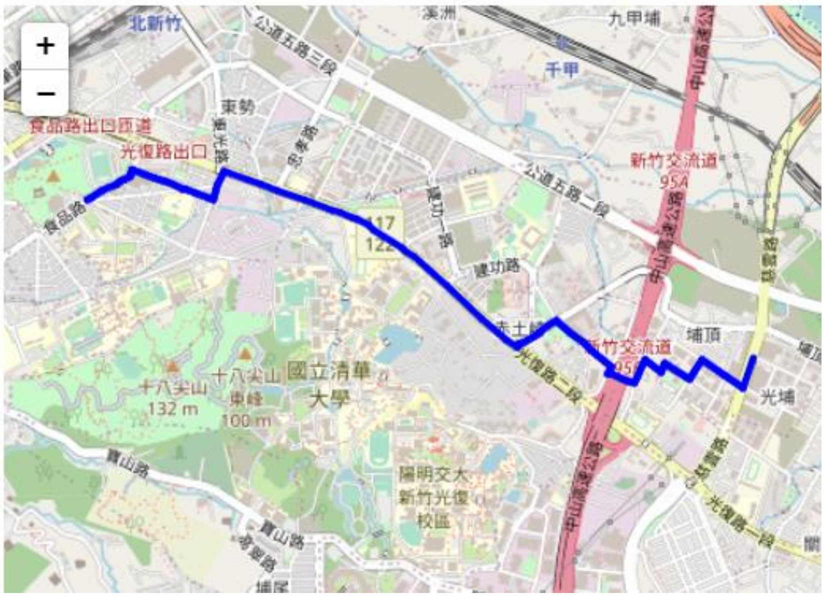
- A*:

```
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
```

2. 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: 4215.521 m
The number of visited nodes in BFS: 4606
```

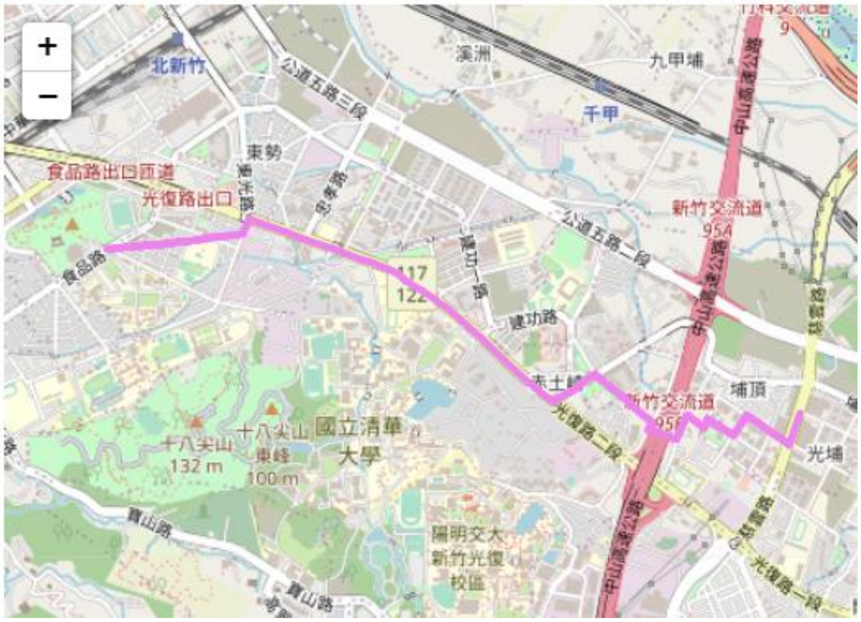


- DFS (stack):

```
The number of nodes in the path found by DFS: 998
Total distance of path found by DFS: 41094.65799999992 m
The number of visited nodes in DFS: 8030
```

- 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: 7207
```



- 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: 1171
```

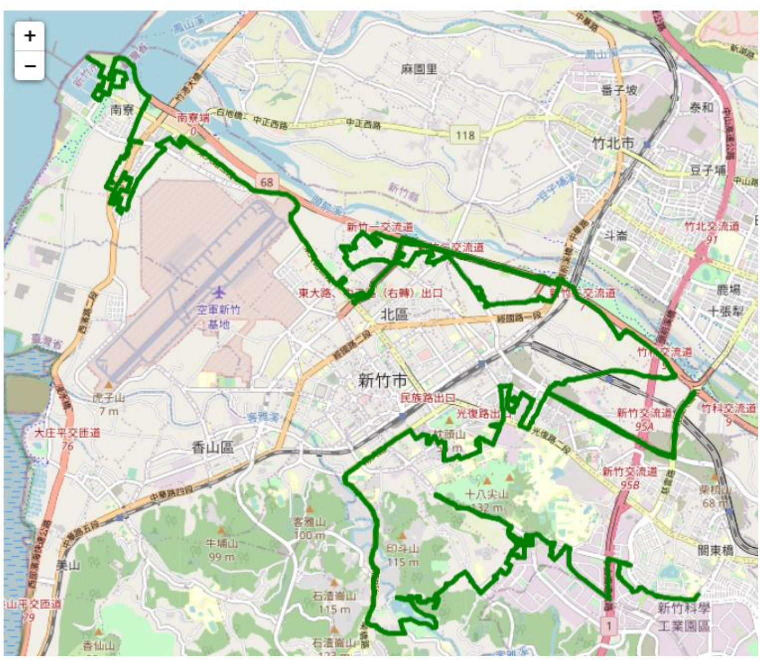
3. Test 3:
   from National Experimental High School At Hsinchu Science Park (ID: 1718165260)
   to Nanliao Fishing Port (ID: 8513026827)

- 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: 11241
```

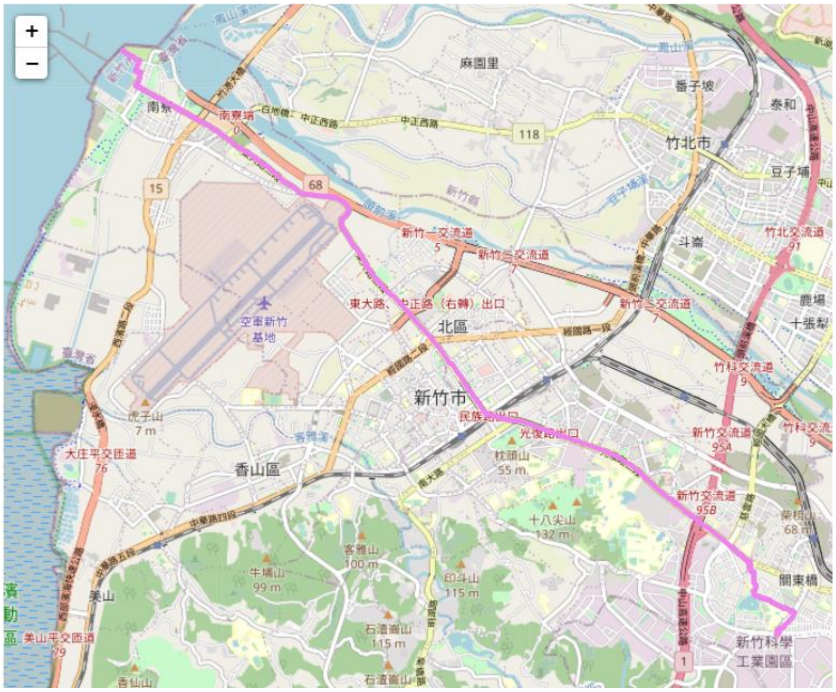
- DFS (stack):

```
The number of nodes in the path found by DFS: 1521
Total distance of path found by DFS: 64821.60399999987 m
The number of visited nodes in DFS: 3291
```



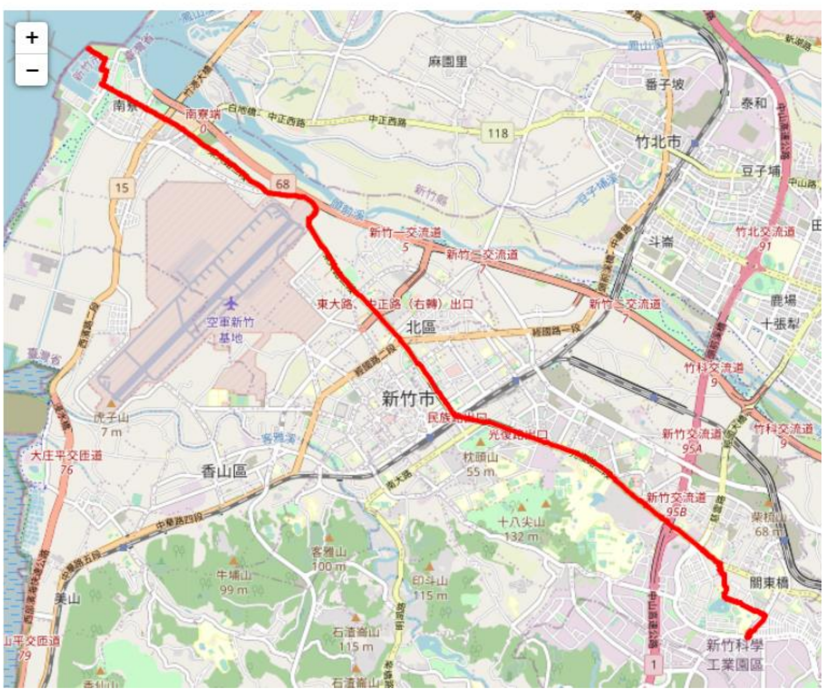
- UCS:

```
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: 11909
```



- A*:

```
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: 7067
```



From the observation of part 5, I notice that UCS and A* have the shortest path (total distance) compared to the two other methods of search. Despite of having different number of visited nodes, UCS and A* have similar path as well as the total distance of their path. However, A* has less visited nodes than UCS does, which means it is more effective and less time consuming. The reason that UCS and A* has such similar results might be that A* is the combination of BFS and UCS. I also notice that DFS is the least effective search method because it goes through lots of nodes and resulting in distance that is many times away from other searching algorithm. In conclusion, we can rank them based on their effectiveness and what we observe from the data above: A*, UCS, BFS, DFS (least effective).

Part III. Question Answering (12%):

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

<u>Ans.</u>

One of the problems I encountered while doing this homework is while doing the UCS, I get the expected value and path for only test 1 and test 3. Thus, I tried to find sources from slides to internet, trying to figure out where the problem is. I solved this problem by modifying and rewriting mostly the code and now it works for all test cases.

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.

<u>Ans.</u>

I think that some other attributes that might be essential for route finding are road traffic and weather. Without data about traffic, we can't say that the route finding is correct or complete. Moreover, traffic took a significant part of our life nowadays. We face traffic almost every day. Just like traffic, weather play an important role in route finding too.

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

<u>Ans.</u>

For mapping, I think some components we need maybe libraries for loading data, geo-plotting, projecting, visualizing, and analyzing the data (for example street networks). For localization, I think geolocation APIs to receive data from mapping, addresses, library to convert addresses to geocoordinates.

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.

<u>Ans.</u>

Dynamic heuristic function for estimated time of arrival includes the changes and data regarding traffic, weather, speed limit, and many more others possible changes that might happen and affects the arrival time. These factors can dynamically change the function along the process of delivering. Let's say if the weather is raining heavily and the traffic is very bad due to working hours, the heuristic function is going to cost more than other situation.