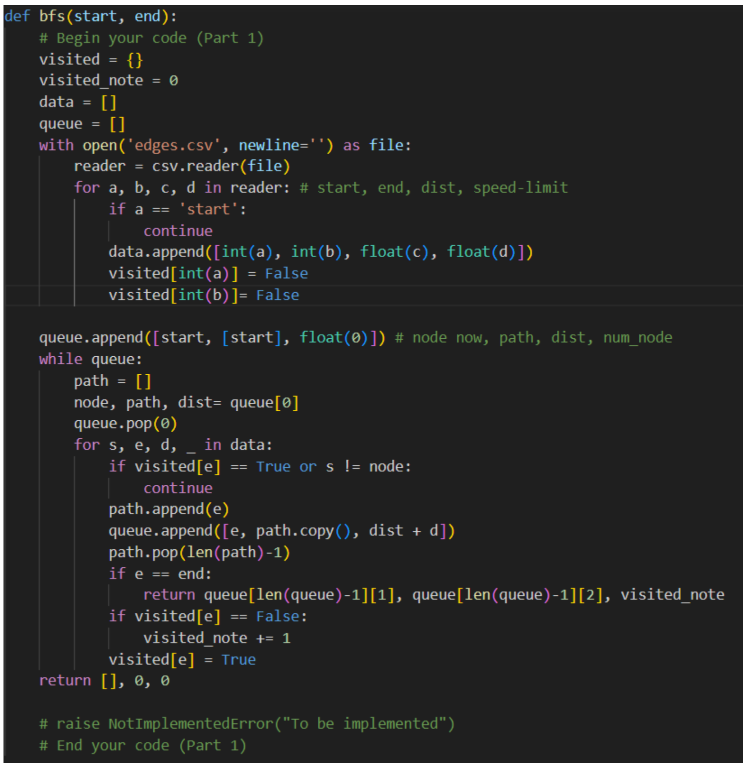
Homework 2: Route Finding

111550088 張育維

Part I. Implementation (6%):

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

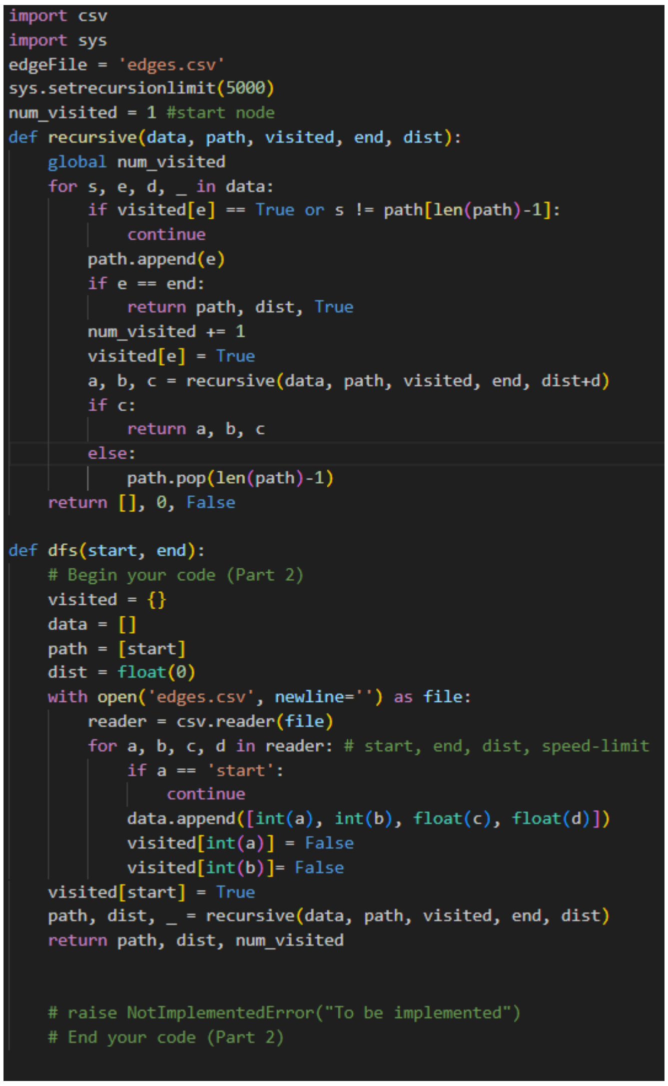
Part1 BFS



```python
def bfs(start, end):
    # Begin your code (Part 1)
    visited = {}
    visited_note = 0
    data = []
    queue = []
    with open('edges.csv', newline='') as file:
        reader = csv.reader(file)
        for a, b, c, d in reader: # start, end, dist, speed-limit
            if a == 'start':
                continue
            data.append([int(a), int(b), float(c), float(d)])
            visited[int(a)] = False
            visited[int(b)]= False

    queue.append([start, [start], float(0)]) # node now, path, dist, num_node
    while queue:
        path = []
        node, path, dist= queue[0]
        queue.pop(0)
        for s, e, d, _ in data:
            if visited[e] == True or s != node:
                continue
            path.append(e)
            queue.append([e, path.copy(), dist + d])
            path.pop(len(path)-1)
            if e == end:
                return queue[len(queue)-1][1], queue[len(queue)-1][2], visited_note
            if visited[e] == False:
                visited_note += 1
            visited[e] = True
    return [], 0, 0

    # raise NotImplementedError("To be implemented")
    # End your code (Part 1)
```

I use the 'open' and 'reader' to put data into the list called 'data', and create the list called 'queue' to perform BFS. Initially, the component in ‘queue’ including the node’s ID, a list of paths, and the distance of the path. In the while loop, I search all components in 'data' every time, and I store the data in the first component of 'queue' into variables named 'node', 'path', and 'dist', and then pop the first component. Next, if the start node of the component in ‘data’ equals ’node’, and the end node hasn’t been visited, I will append a new component into the 'queue', which is updated with the end node, path, and the distance. When the new end node (that hasn’t been visited) is found, the variable named 'visited_node' plus 1. Finally, if the ending node equals the variable named 'end', then return the result, including the list of paths, total distance, and the total number of nodes visited.

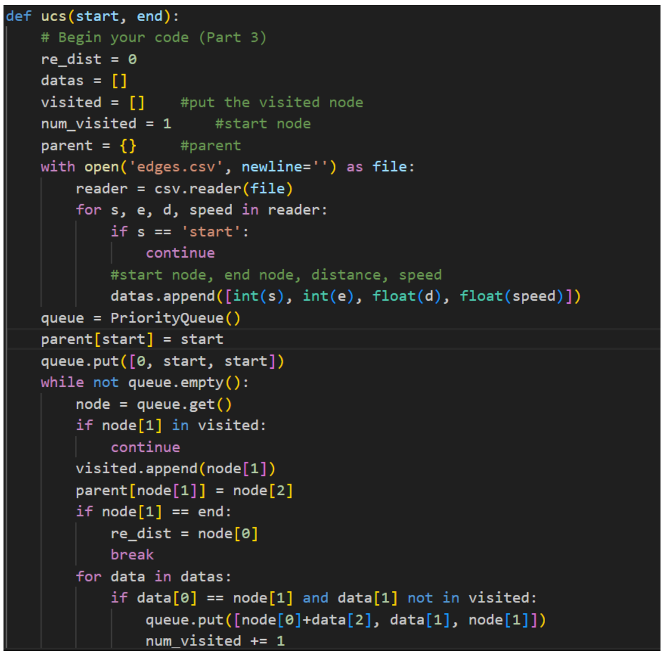
Part2 DFS(recursive)



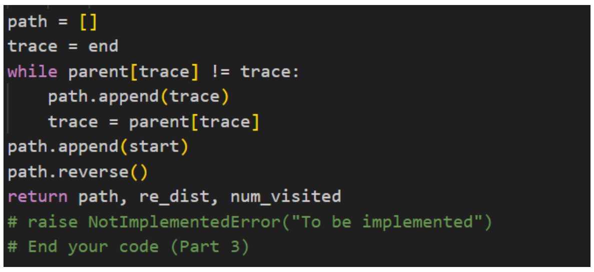
```python
import csv
import sys
edgeFile = 'edges.csv'
sys.setrecursionlimit(5000)
num_visited = 1 #start node
def recursive(data, path, visited, end, dist):
    global num_visited
    for s, e, d, _ in data:
        if visited[e] == True or s != path[len(path)-1]:
            continue
        path.append(e)
        if e == end:
            return path, dist, True
        num_visited += 1
        visited[e] = True
        a, b, c = recursive(data, path, visited, end, dist+d)
        if c:
            return a, b, c
        else:
            path.pop(len(path)-1)
    return [], 0, False

def dfs(start, end):
    # Begin your code (Part 2)
    visited = {}
    data = []
    path = [start]
    dist = float(0)
    with open('edges.csv', newline='') as file:
        reader = csv.reader(file)
        for a, b, c, d in reader: # start, end, dist, speed-limit
            if a == 'start':
                continue
            data.append([int(a), int(b), float(c), float(d)])
            visited[int(a)] = False
            visited[int(b)]= False
    visited[start] = True
    path, dist, _ = recursive(data, path, visited, end, dist)
    return path, dist, num_visited


    # raise NotImplementedError("To be implemented")
    # End your code (Part 2)
```

In DFS, I choose to implement it by recursion. I import 'sys' to change the upper bound of recursion. I use a similar way to BFS to get the data from a CSV file, and I set the number of visited nodes as a global variable. In the for loop, if I find a path where the ending node hasn't been visited and the starting node equals the ending of the list called 'path', I will append the ending node into the list and update the distance in the list, then call the recursive function again. When the ending of the list 'path' equals the 'end' ID, I return the path, distance, and 'True', which means that this path is feasible. When the for loop ends and no updating occurs, I return an empty list, 0, and 'False', which means that this is not the correct path. Whenever I search a node, the number of visited nodes increases by 1.

Part3 ucs



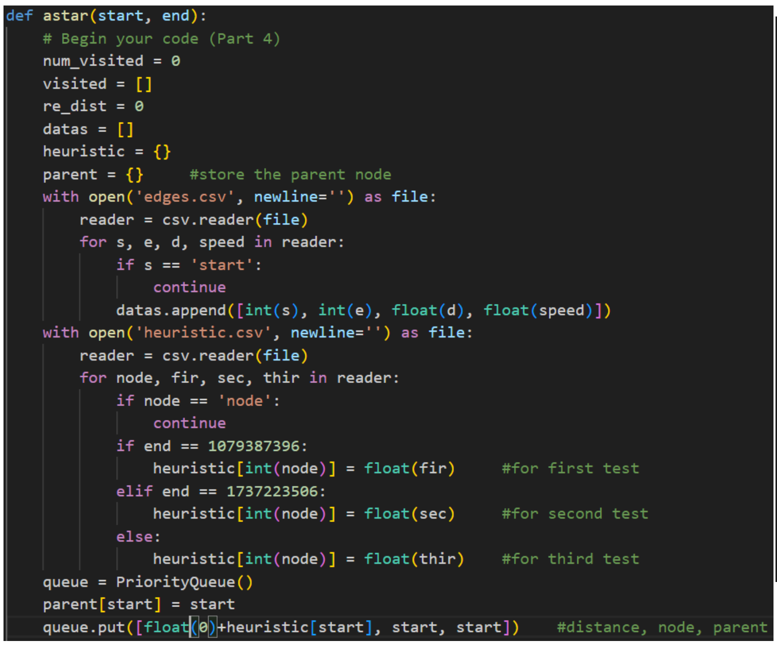
```python
def ucs(start, end):
    # Begin your code (Part 3)
    re_dist = 0
    datas = []
    visited = []    #put the visited node
    num_visited = 1     #start node
    parent = {}     #parent
    with open('edges.csv', newline='') as file:
        reader = csv.reader(file)
        for s, e, d, speed in reader:
            if s == 'start':
                continue
            #start node, end node, distance, speed
            datas.append([int(s), int(e), float(d), float(speed)])
    queue = PriorityQueue()
    parent[start] = start
    queue.put([0, start, start])
    while not queue.empty():
        node = queue.get()
        if node[1] in visited:
            continue
        visited.append(node[1])
        parent[node[1]] = node[2]
        if node[1] == end:
            re_dist = node[0]
            break
        for data in datas:
            if data[0] == node[1] and data[1] not in visited:
                queue.put([node[0]+data[2], data[1], node[1]])
                num_visited += 1
```



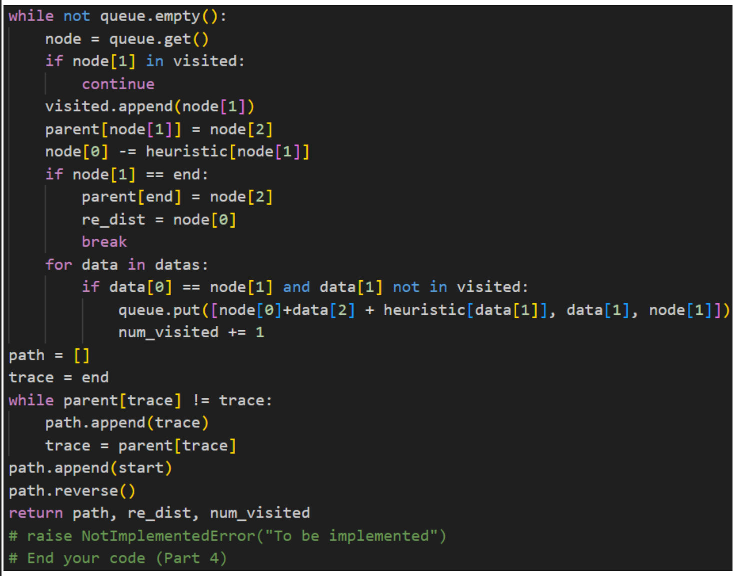
```python
path = []
trace = end
while parent[trace] != trace:
    path.append(trace)
    trace = parent[trace]
path.append(start)
path.reverse()
return path, re_dist, num_visited
# raise NotImplementedError("To be implemented")
# End your code (Part 3)
```

I import the data from CSV into a list called 'datas' and create a priority queue called 'queue', sorting the elements by distance. Each element in the queue contains the distance, node's ID, and the ID of the previous one. In the while loop, I extract the component with the smallest distance, mark it as visited, and store the relationship between its node's ID and the ID of the previous one in a dictionary called 'parent'. If the data in 'datas' starts from this node's ID, I set its distance as the sum of its own distance and the distance from the extracted one's, and assign the previous node as the extracted one, then add it to the 'queue'. When the ending node is reached, I break out of the while loop and trace all paths using the 'parent' dictionary. Finally, I return the list of paths, total distance, and the number of nodes searched in this function.

Part4 A*



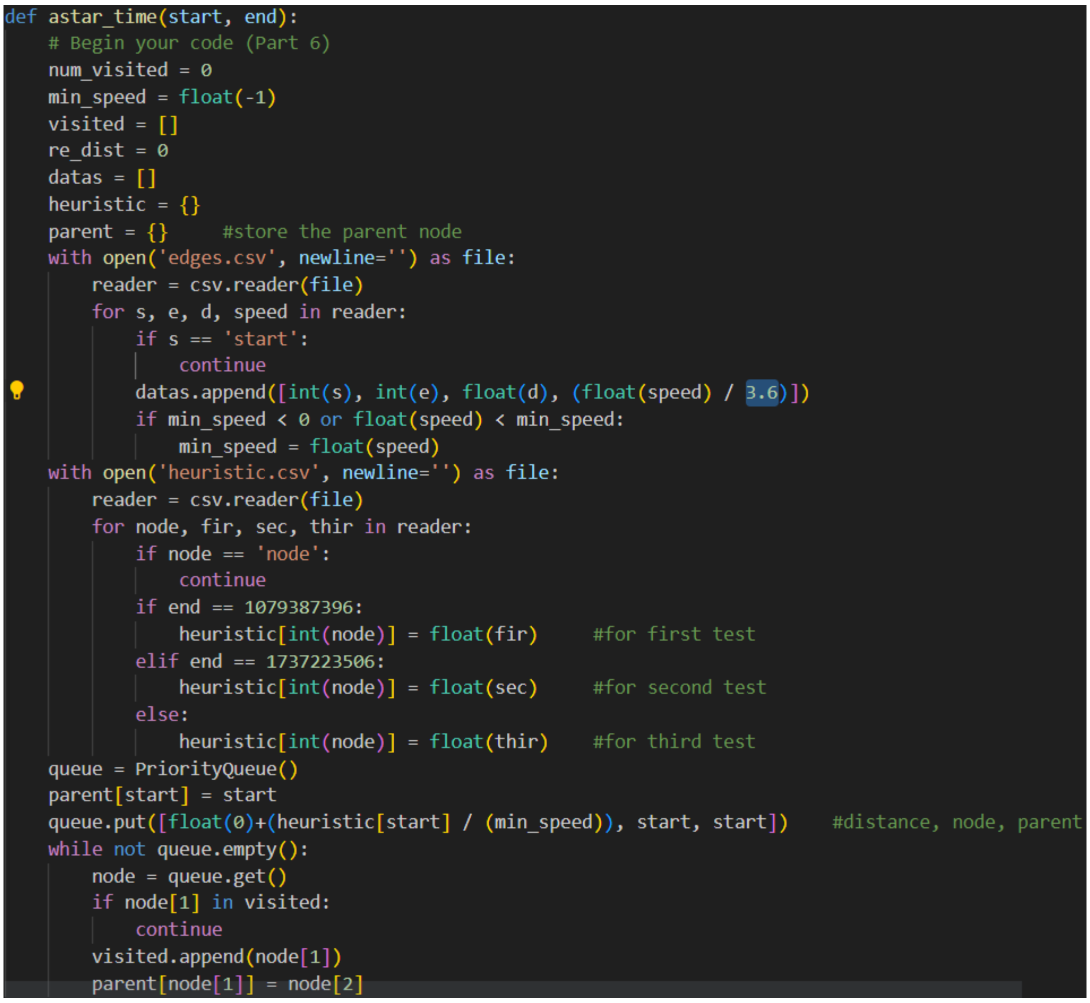
```python
def astar(start, end):
    # Begin your code (Part 4)
    num_visited = 0
    visited = []
    re_dist = 0
    datas = []
    heuristic = {}
    parent = {}     #store the parent node
    with open('edges.csv', newline='') as file:
        reader = csv.reader(file)
        for s, e, d, speed in reader:
            if s == 'start':
                continue
            datas.append([int(s), int(e), float(d), float(speed)])
    with open('heuristic.csv', newline='') as file:
        reader = csv.reader(file)
        for node, fir, sec, thir in reader:
            if node == 'node':
                continue
            if end == 1079387396:
                heuristic[int(node)] = float(fir)     #for first test
            elif end == 1737223506:
                heuristic[int(node)] = float(sec)     #for second test
            else:
                heuristic[int(node)] = float(thir)    #for third test
    queue = PriorityQueue()
    parent[start] = start
    queue.put([float(0)+heuristic[start], start, start])    #distance, node, parent
```



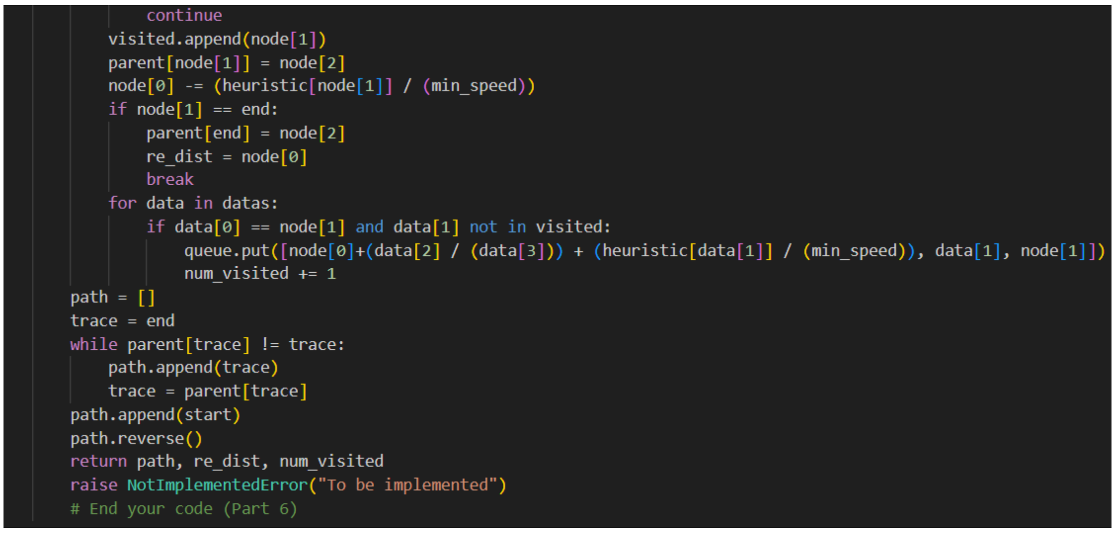
```python
while not queue.empty():
    node = queue.get()
    if node[1] in visited:
        continue
    visited.append(node[1])
    parent[node[1]] = node[2]
    node[0] -= heuristic[node[1]]
    if node[1] == end:
        parent[end] = node[2]
        re_dist = node[0]
        break
    for data in datas:
        if data[0] == node[1] and data[1] not in visited:
            queue.put([node[0]+data[2] + heuristic[data[1]], data[1], node[1]])
            num_visited += 1
path = []
trace = end
while parent[trace] != trace:
    path.append(trace)
    trace = parent[trace]
path.append(start)
path.reverse()
return path, re_dist, num_visited
# raise NotImplementedError("To be implemented")
# End your code (Part 4)
```

I do the same thing when I retrieve data from 'edges.csv', and I set up a dictionary called 'heuristic'. In 'heuristic.csv', I set the node's ID as the key and the distance, depending on the value of 'end', as the value, storing the relationship in the 'heuristic' dictionary. After that, my approach is similar to Part 3. The only difference is that when I add an element to the 'queue', I add the value stored in 'heuristic' with the key equal to its ID to the distance. When I remove an element from the 'queue', I subtract this value from the distance before adding the new elements to the 'queue'. Finding the path is also the same as in Part 3.

Bonus A*(time)



```python
def astar_time(start, end):
    # Begin your code (Part 6)
    num_visited = 0
    min_speed = float(-1)
    visited = []
    re_dist = 0
    datas = []
    heuristic = {}
    parent = {}     #store the parent node
    with open('edges.csv', newline='') as file:
        reader = csv.reader(file)
        for s, e, d, speed in reader:
            if s == 'start':
                continue
            datas.append([int(s), int(e), float(d), (float(speed) / 3.6)])
            if min_speed < 0 or float(speed) < min_speed:
                min_speed = float(speed)
    with open('heuristic.csv', newline='') as file:
        reader = csv.reader(file)
        for node, fir, sec, thir in reader:
            if node == 'node':
                continue
            if end == 1079387396:
                heuristic[int(node)] = float(fir)     #for first test
            elif end == 1737223506:
                heuristic[int(node)] = float(sec)     #for second test
            else:
                heuristic[int(node)] = float(thir)    #for third test
    queue = PriorityQueue()
    parent[start] = start
    queue.put([float(0)+(heuristic[start] / (min_speed)), start, start])    #distance, node, parent
    while not queue.empty():
        node = queue.get()
        if node[1] in visited:
            continue
        visited.append(node[1])
        parent[node[1]] = node[2]
```


```
            continue
        visited.append(node[1])
        parent[node[1]] = node[2]
        node[0] -= (heuristic[node[1]] / (min_speed))
        if node[1] == end:
            parent[end] = node[2]
            re_dist = node[0]
            break
        for data in datas:
            if data[0] == node[1] and data[1] not in visited:
                queue.put([node[0]+(data[2] / (data[3])) + (heuristic[data[1]] / (min_speed)), data[1], node[1]])
                num_visited += 1
    path = []
    trace = end
    while parent[trace] != trace:
        path.append(trace)
        trace = parent[trace]
    path.append(start)
    path.reverse()
    return path, re_dist, num_visited
    raise NotImplementedError("To be implemented")
    # End your code (Part 6)
```

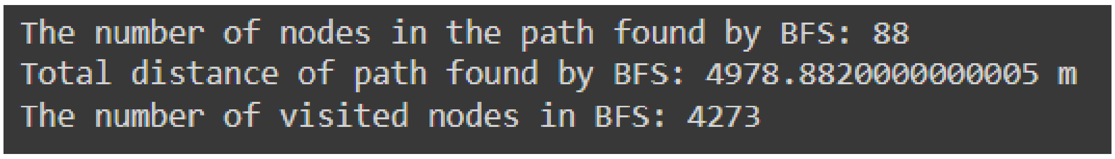
In this part, it's similar to the part 4, but I change the value in priority queue from distance to time(using distance / (speed limit / 3.6)). And when I read the data from csv to the list, I store the minimum speed limit to use in heuristic function(distance read from heuristic.csv / (minimum speed limit / 3.6))

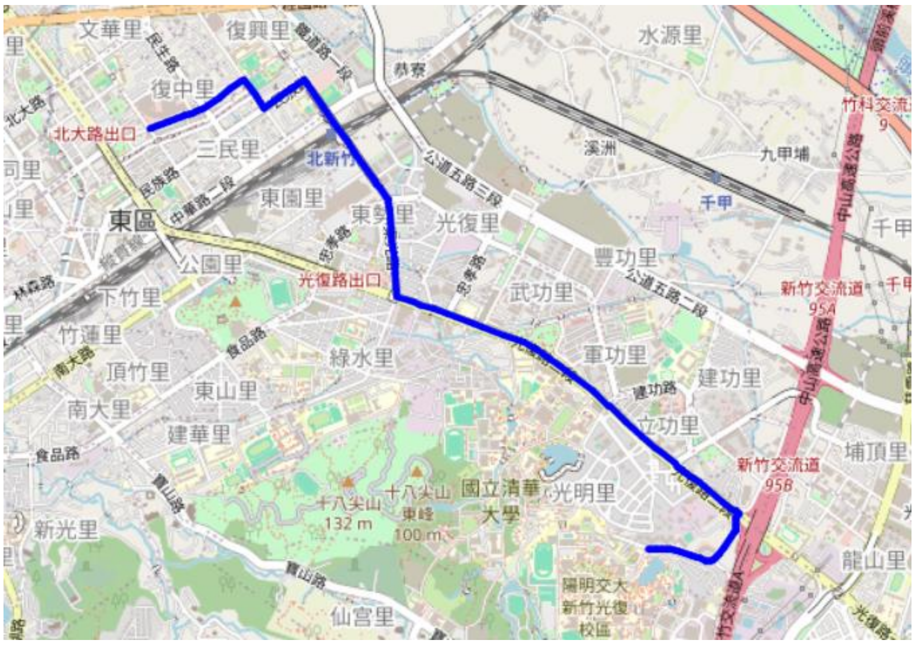
Part II. Results & Analysis (12%):

Please screenshot the results.

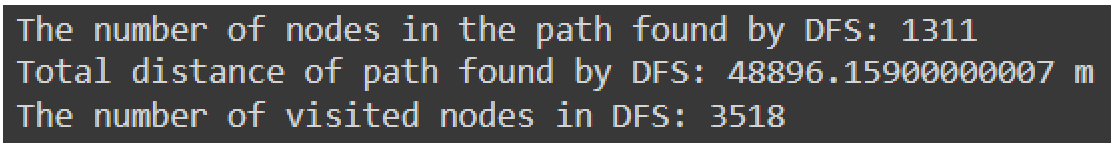
Test1: from National Yang Ming Chiao Tung University (ID: 2270143902) to Big City Shopping Mall (ID: 1079387396)

BFS:

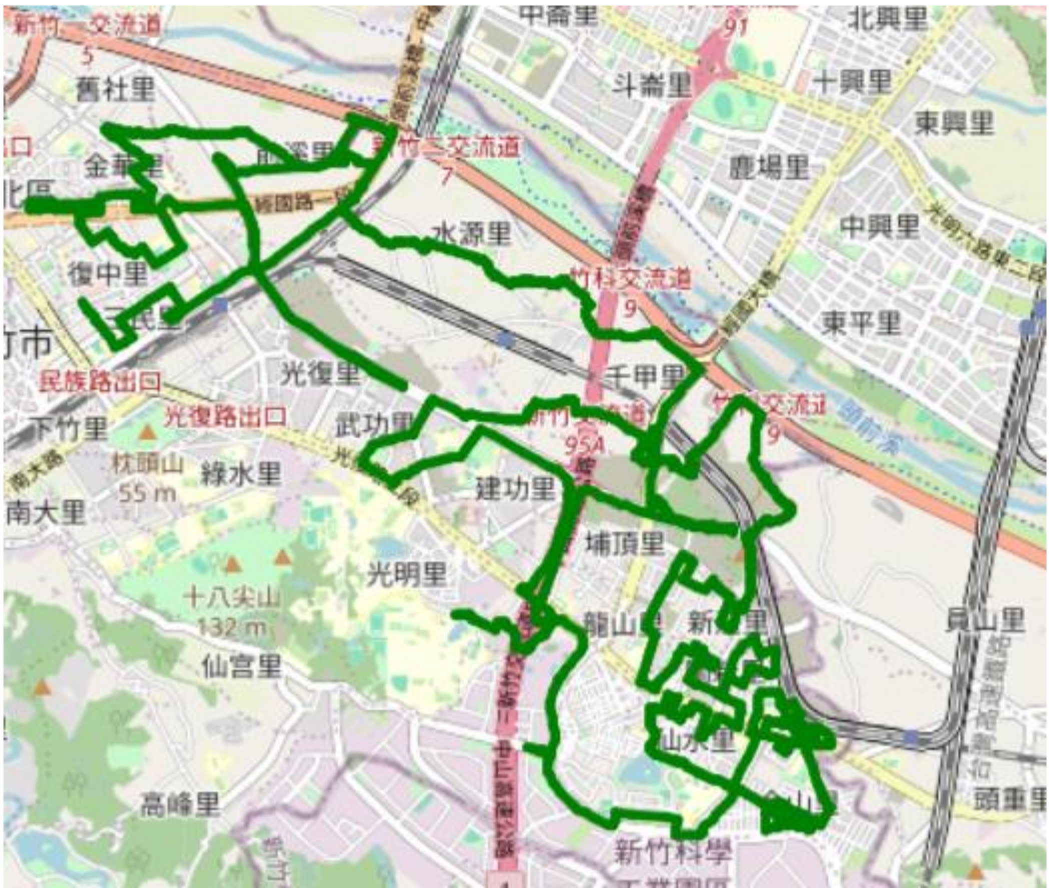

```
The number of nodes in the path found by BFS: 88
Total distance of path found by BFS: 4978.8820000000005 m
The number of visited nodes in BFS: 4273
```



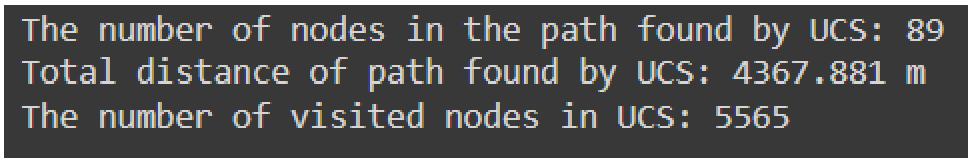
DFS(recursive):



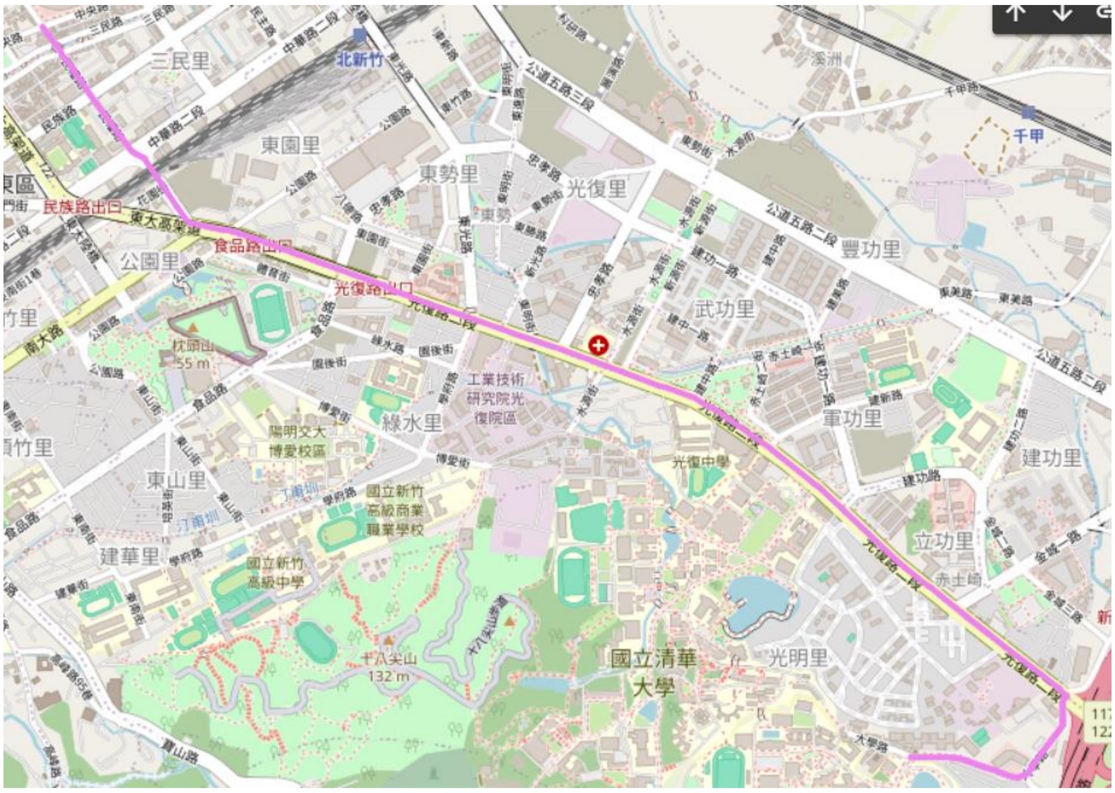
```
The number of nodes in the path found by DFS: 1311
Total distance of path found by DFS: 48896.15900000007 m
The number of visited nodes in DFS: 3518
```



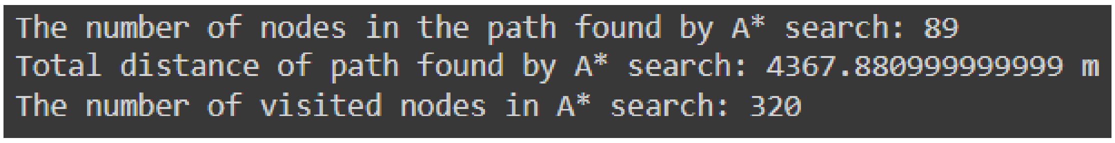
UCS:



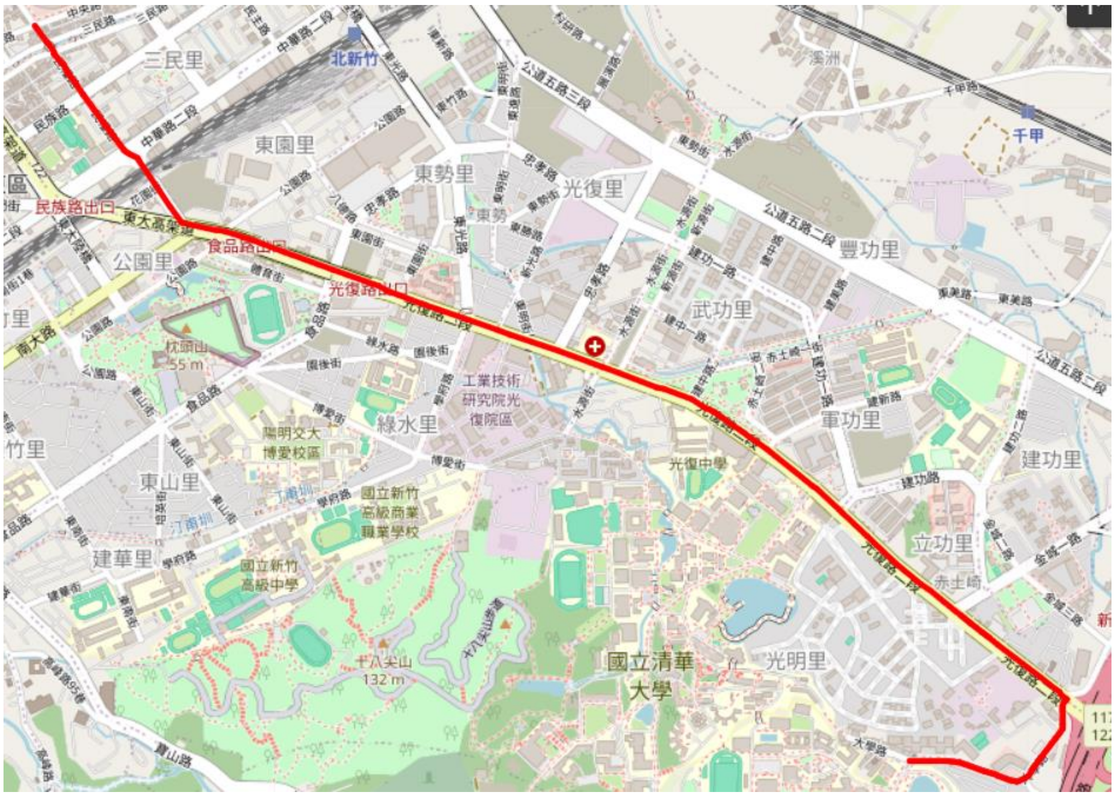
```
The number of nodes in the path found by UCS: 89
Total distance of path found by UCS: 4367.881 m
The number of visited nodes in UCS: 5565
```



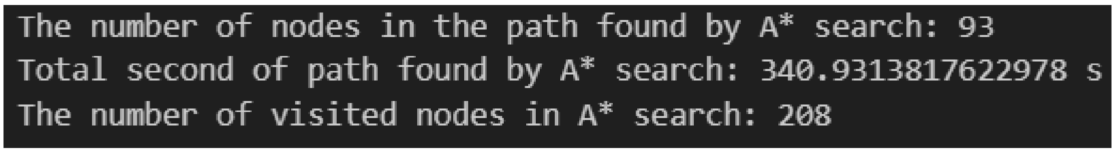
A*:



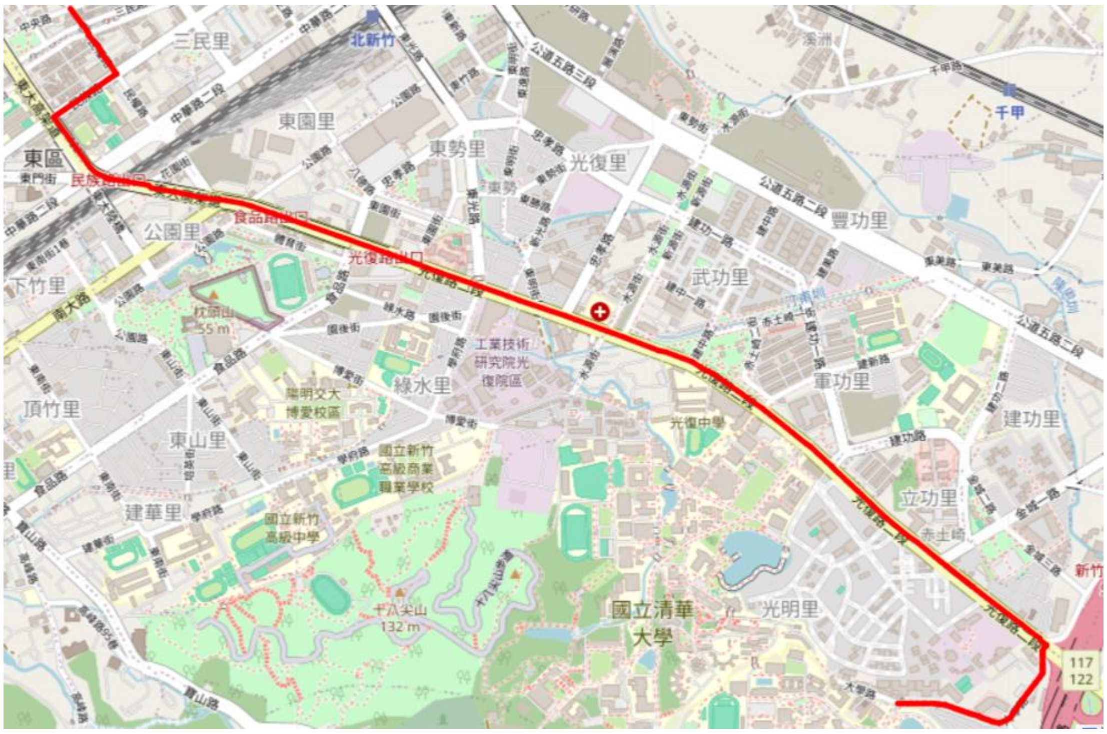
```
The number of nodes in the path found by A* search: 89
Total distance of path found by A* search: 4367.880999999999 m
The number of visited nodes in A* search: 320
```



Bonus:

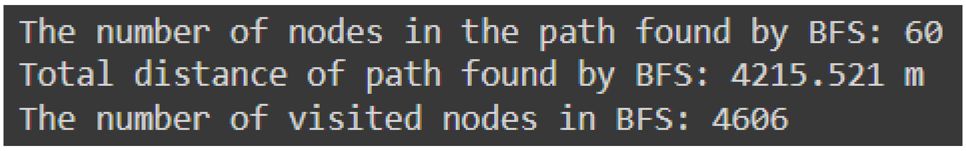


```
The number of nodes in the path found by A* search: 93
Total second of path found by A* search: 340.9313817622978 s
The number of visited nodes in A* search: 208
```

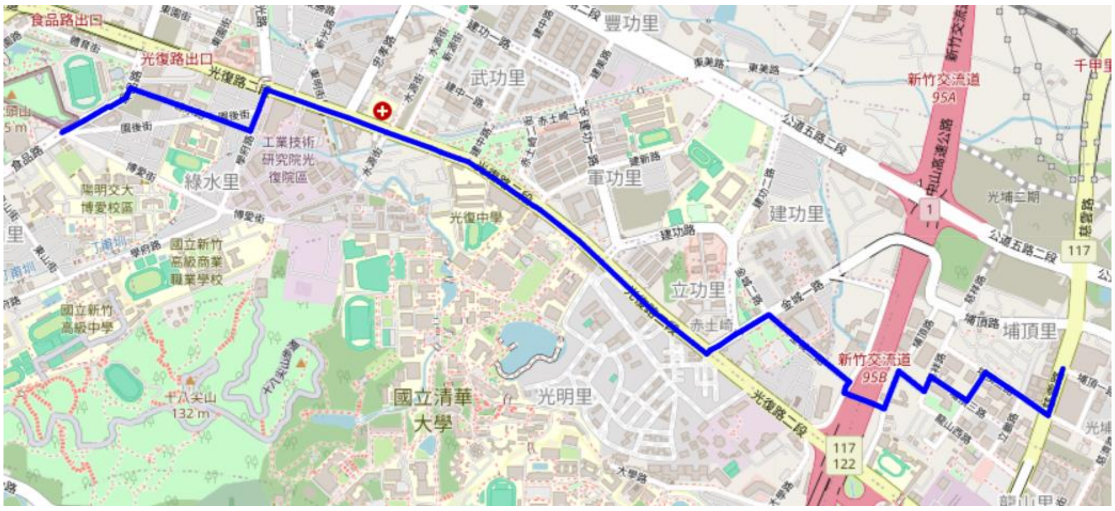


Test2: from Hsinchu Zoo (ID: 426882161) to COSTCO Hsinchu Store (ID: 1737223506)

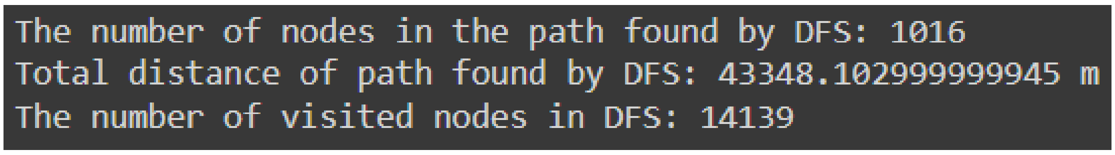
BFS:



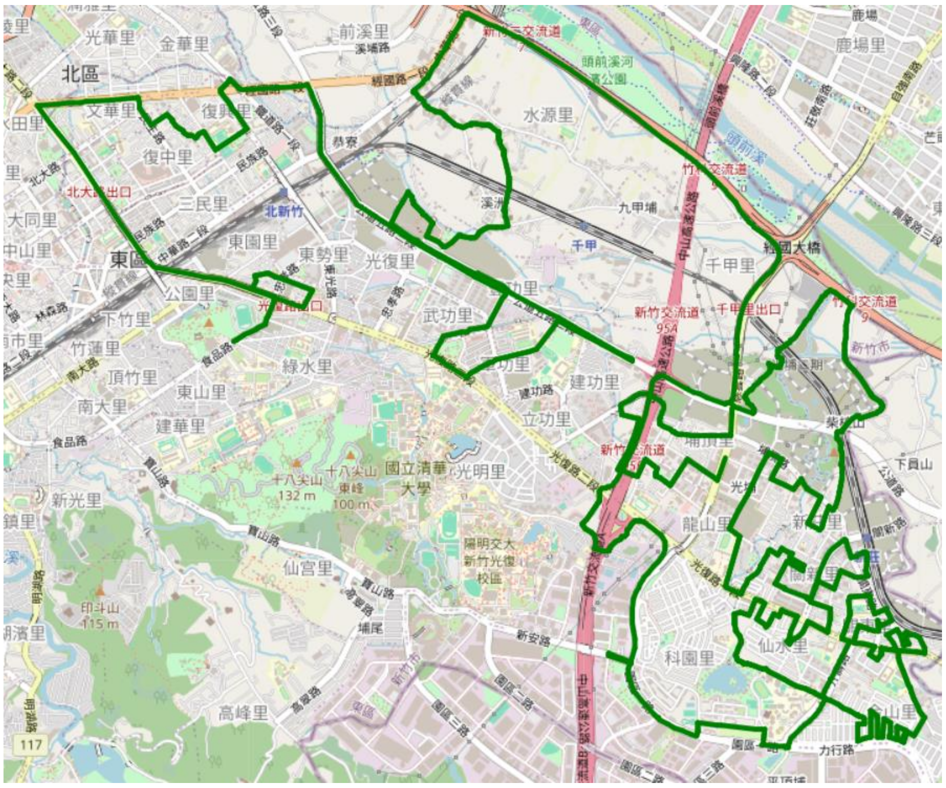
```
The number of nodes in the path found by BFS: 60
Total distance of path found by BFS: 4215.521 m
The number of visited nodes in BFS: 4606
```



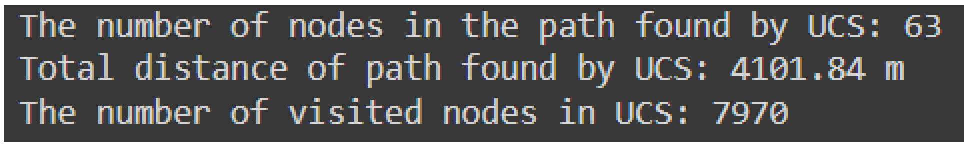
DFS(recursive):



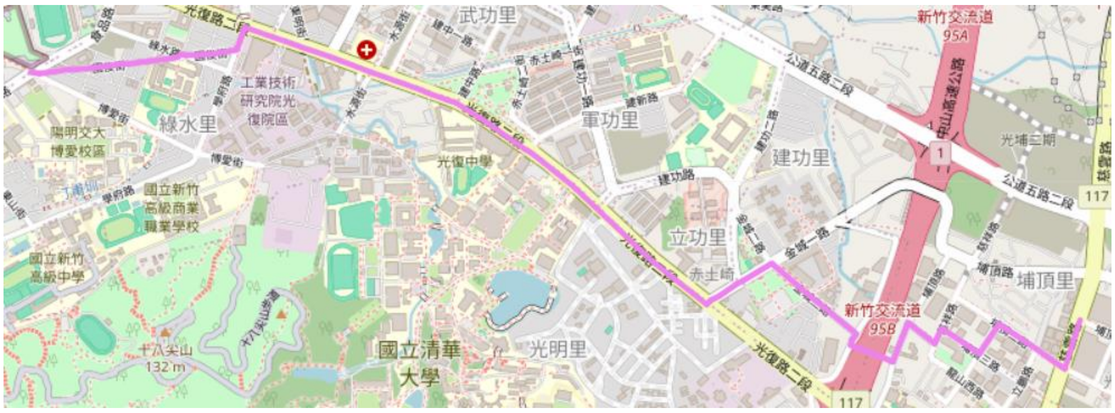
```
The number of nodes in the path found by DFS: 1016
Total distance of path found by DFS: 43348.102999999945 m
The number of visited nodes in DFS: 14139
```



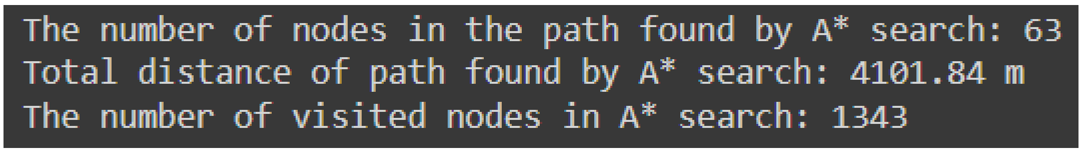
UCS



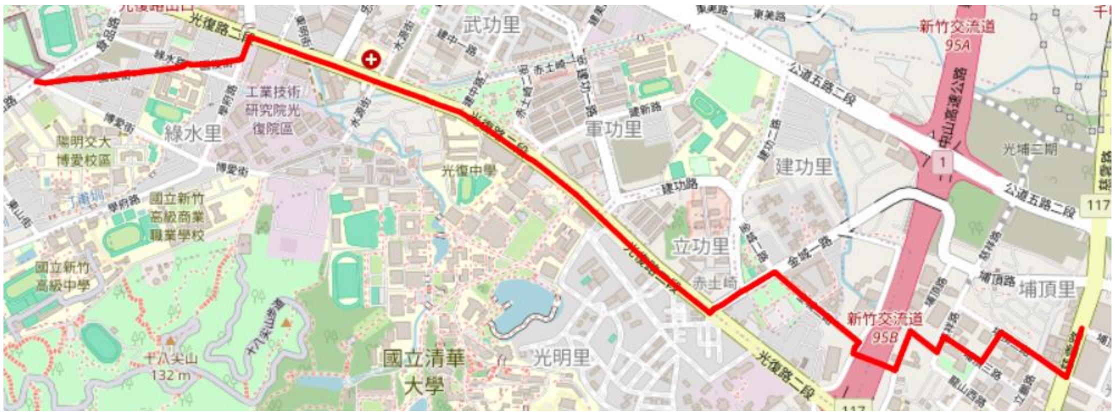
```
The number of nodes in the path found by UCS: 63
Total distance of path found by UCS: 4101.84 m
The number of visited nodes in UCS: 7970
```



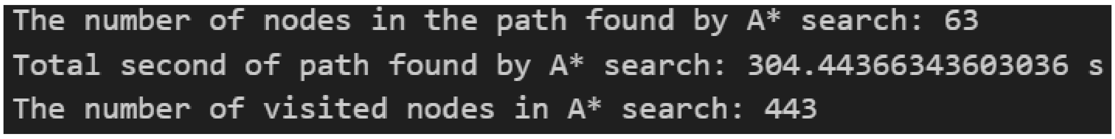
A*



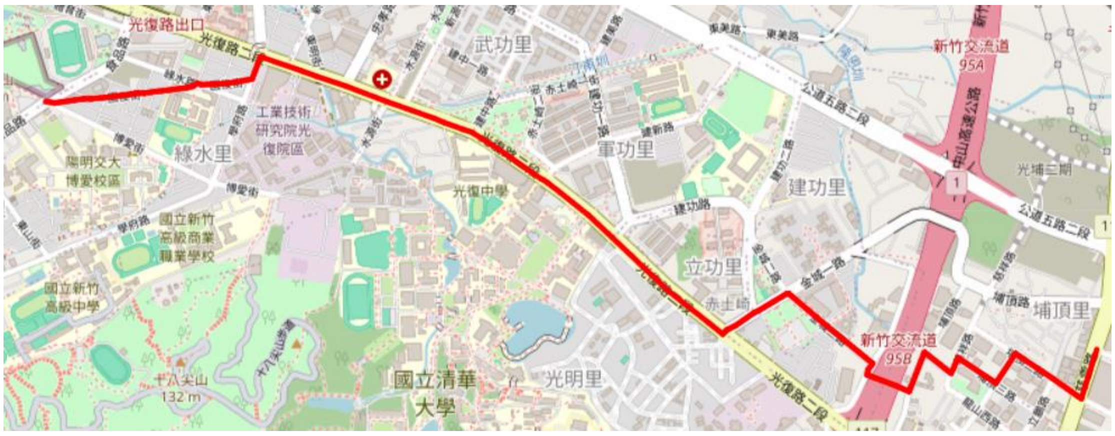
```
The number of nodes in the path found by A* search: 63
Total distance of path found by A* search: 4101.84 m
The number of visited nodes in A* search: 1343
```



Bonus

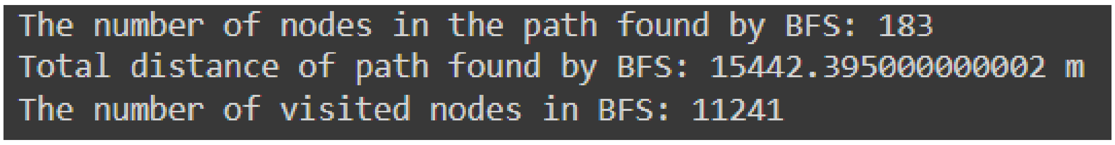


```
The number of nodes in the path found by A* search: 63
Total second of path found by A* search: 304.44366343603036 s
The number of visited nodes in A* search: 443
```

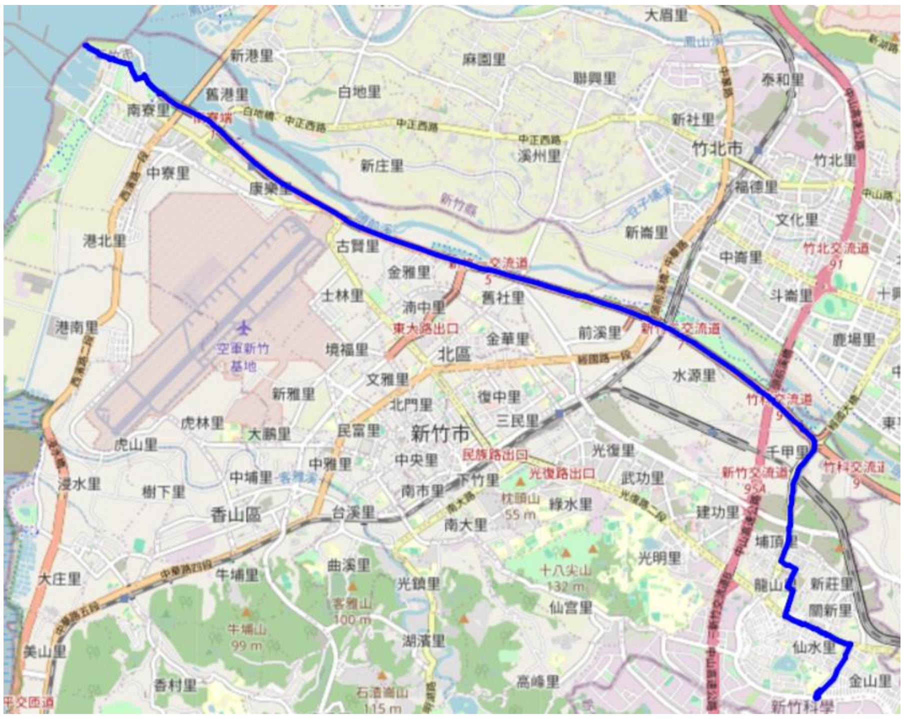


Test3: from National Experimental High School At Hsinchu Science Park (ID: 1718165260) to Nanliao Fighing Port (ID: 8513026827)

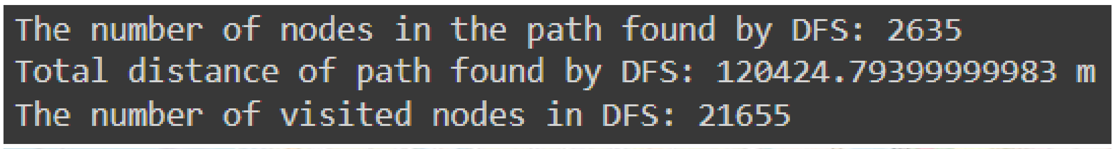
BFS:



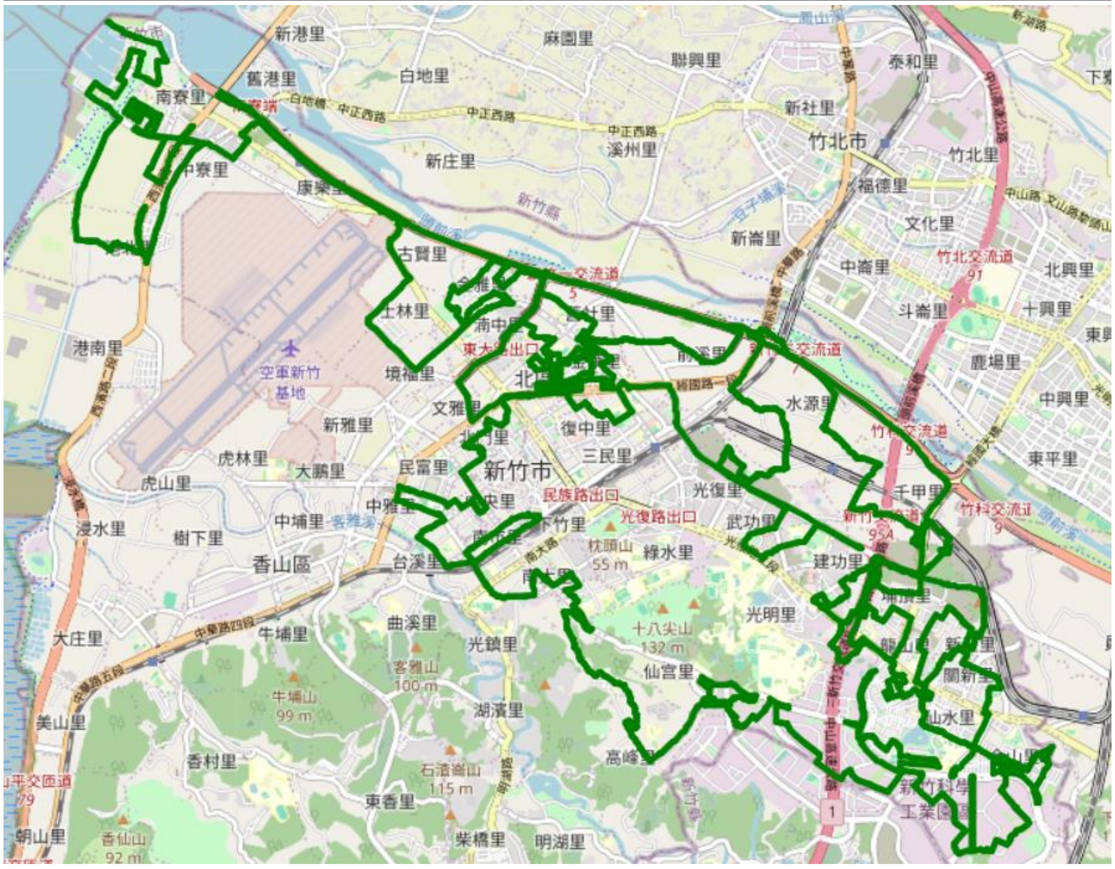
```
The number of nodes in the path found by BFS: 183
Total distance of path found by BFS: 15442.395000000002 m
The number of visited nodes in BFS: 11241
```



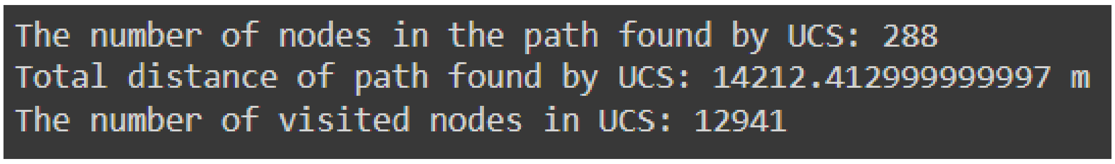
DFS(recursive)



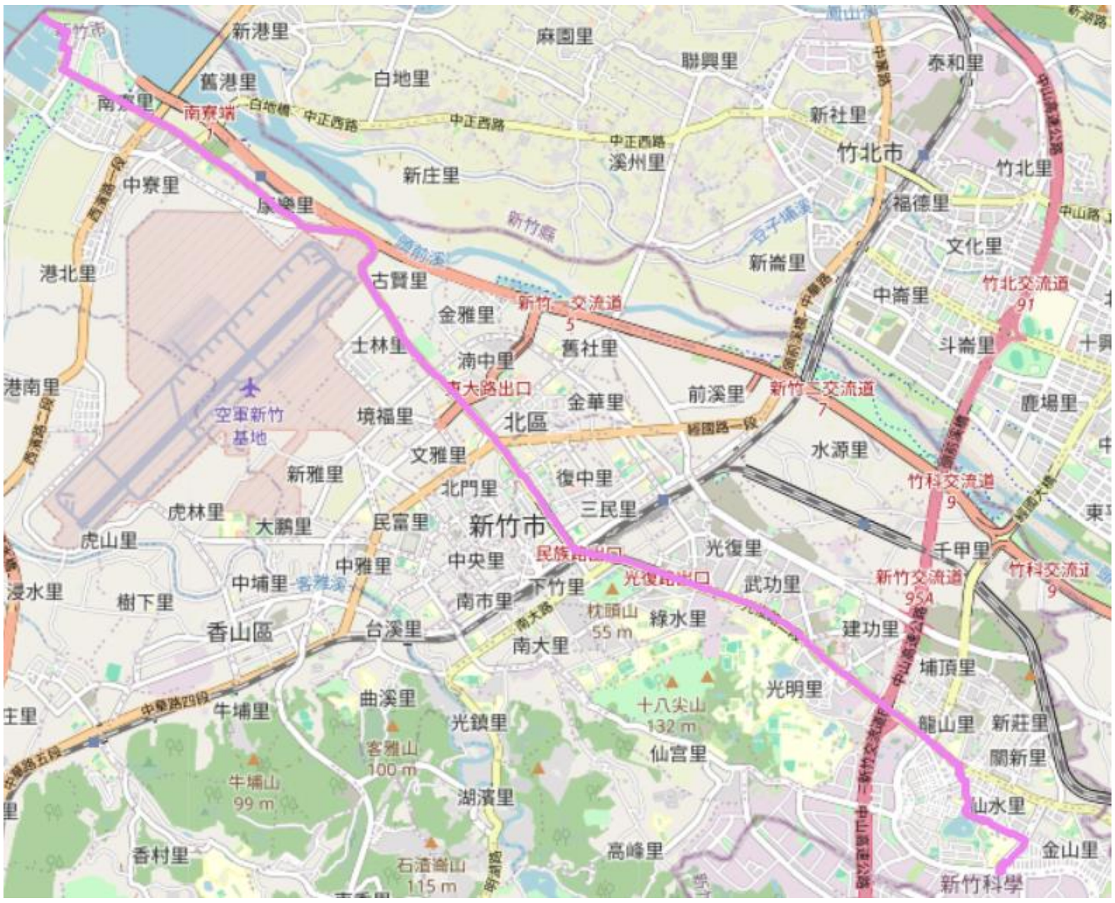
```
The number of nodes in the path found by DFS: 2635
Total distance of path found by DFS: 120424.79399999983 m
The number of visited nodes in DFS: 21655
```



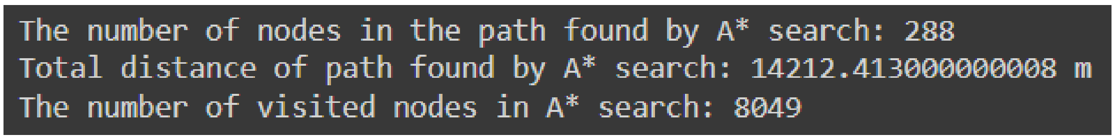
UCS



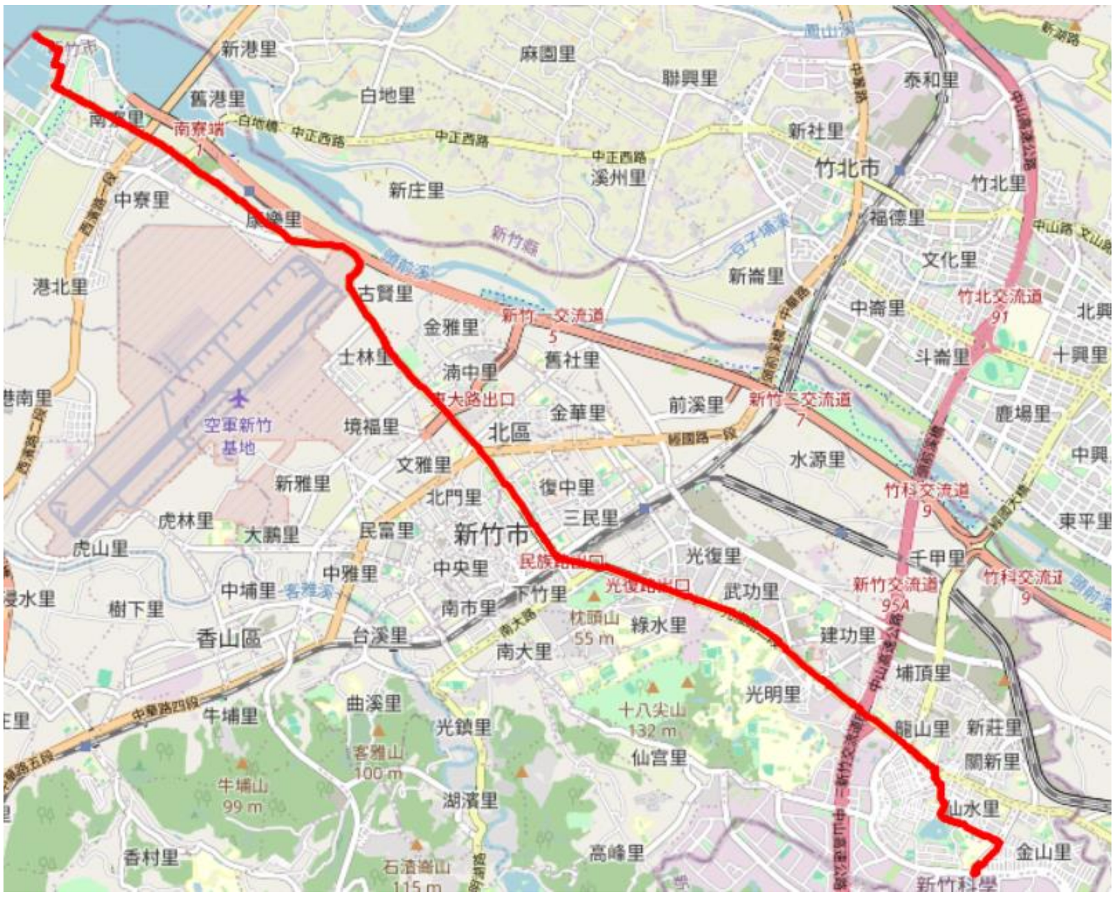
```
The number of nodes in the path found by UCS: 288
Total distance of path found by UCS: 14212.412999999997 m
The number of visited nodes in UCS: 12941
```



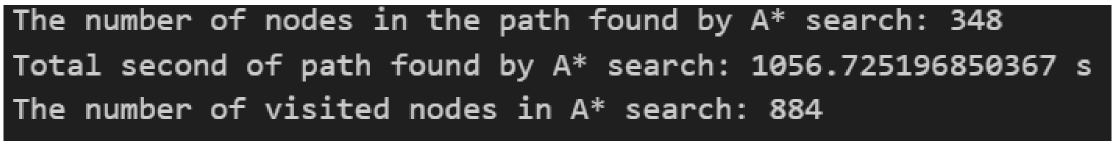
A*

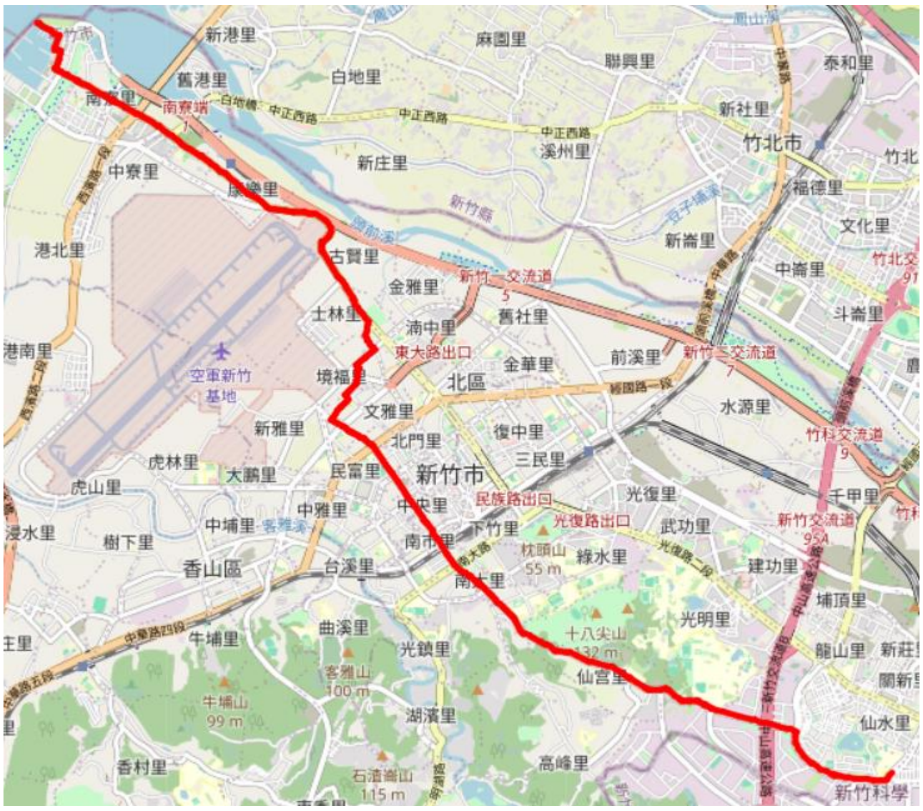


```
The number of nodes in the path found by A* search: 288
Total distance of path found by A* search: 14212.413000000008 m
The number of visited nodes in A* search: 8049
```



Bonus


```
The number of nodes in the path found by A* search: 348
Total second of path found by A* search: 1056.725196850367 s
The number of visited nodes in A* search: 884
```



Part III. Question Answering (12%):

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

The problem I encountered is that when I was performing BFS, I wanted to store the list of paths in the component. Initially, I coded it as 'queue.append([e, path, dist + d])' (where 'path' already had the new node appended to it). However, it didn’t work as expected. When I debugged it, I found that there was nothing in the path of the new element. Finally, I solved it by using the 'copy()' function of the list, like this: 'queue.append([e, path.copy(), dist + d])', and it worked.

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 conditions can affect travel time, as a high volume of vehicles on the road may prevent drivers from reaching the speed limit.
Government policies also influence travel, especially during important festivals when high-speed roads may have restrictions on the number of passengers allowed. If there are only one or two persons in the car, they may not be permitted to use the high-speed road.

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 utilize existing maps such as Google Maps or leverage user-generated data to continuously update and enhance the map.
For localization components, we can rely on GPS signals to determine the location. Alternatively, we can install transmitters at known locations and use signal strength to estimate the distance between transmitters and vehicles, enabling accurate 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. Hint: You can consider meal prep time, delivery priority, multiple orders, etc
This is a possible ETA equation:
ETA = (Base Time + Meal preptime) * Delivery Priority * Multiple Orders
Base Time : The time required for delivery.
Meal preptime: The preparation time for the meal at the restaurant.
Delivery Priority: The importance of the order.
Multiple Orders: The total number of the meal in the same importance.

I define the ETA in this equation because I believe that the total time for an order is the sum of the times for orders with equal or higher importance. Therefore, I multiply the total time for preparation and delivery by the number of orders that must be delivered before this meal.