109550182 莊婕妤

AI Homework2 Report

Part I. Implementation (6%)

Part 1 BFS

```
# Begin your code (Part 1)
"""
1. load the csv file into rows
2. store them into the dictionary edges in the format:
    {startID : list of attached (endID, distance)}
3. implement BFS
4. backtrack the path by using the list trace
5. return path, dist, num_visited
"""
# raise NotImplementedError("To be implemented")
```

```
edges = {}
with open(edgeFile, newline='') as csvfile:
    rows = csv.reader(csvfile, delimiter=',')
    headers = next(rows)
    # edges: {startID : list of attached (endID, distance) s}
    for row in rows:
        if edges.get(int(row[0])) != None:
            edges[int(row[0])].append((int(row[1]), float(row[2])))
        else: edges[int(row[0])] = [(int(row[1]), float(row[2]))]

queue = [(start, 0, 0)]
explored = [start]
trace = {}
num_visited = 0
```

```
while (True):
    flag = True
    if queue[0][0] in edges.keys():  # check if it is a leaf node
        for ID, distance in edges[queue[0][0]]:
            if ID in explored: continue  # check if the node is explored
            if ID == end:
                trace[end] = (queue[0][0], distance)
                num_visited += 1
                flag = False
                break
            queue.append((ID, distance, queue[0][0]))
            explored.append(ID)
            num_visited += 1
    trace[queue[0][0]] = (queue[0][2], queue[0][1])
    if flag == False: break
    queue.pop(0)
```

```
dist = 0
path = [end]
dist += trace[path[0]][1]
while trace[path[0]][0] != 0:
    path.insert(0, trace[path[0]][0])
    dist += trace[path[0]][1]

return path, dist, num_visited
# End your code (Part 1)
```

Part 2 DFS

```
"""
1. load the csv file into rows
2. store them into the dictionary edges in the format:
    {startID : list of attached (endID, distance)}
3. implement DFS with stack
4. backtrack the path by using the list trace
5. return path, dist, num_visited
"""
# raise NotImplementedError("To be implemented")
edges = {}
with open(edgeFile, newline='') as csvfile:
    rows = csv.reader(csvfile, delimiter=',')
    headers = next(rows)
    # edges: {startID : list of attached (endID, distance) s}
    for row in rows:
        if edges.get(int(row[0])) != None:
            edges[int(row[0])].append((int(row[1]), float(row[2])))
        else: edges[int(row[0])] = [(int(row[1]), float(row[2]))]
```

```
stack = [(start, 0, 0)]
explored = [start]
trace = {}
num_visited = 0
while (True):
    flag = True
    top = stack[-1]
    stack.pop()
    if top[0] in edges.keys():  # check if it is a leaf node
        for ID, distance in edges[top[0]]:
            if ID in explored: continue  # check if the node is explored
            if ID == end:
                num_visited += 1
                trace[end] = (top[0], distance)
                flag = False
                break
            stack.append((ID, distance, top[0]))
            explored.append(ID)
            num_visited += 1
    trace[top[0]] = (top[2], top[1])
    if flag == False: break
```

```
    dist = 0
    path = [end]
    dist += trace[path[0]][1]
    while trace[path[0]][0] != 0:
        path.insert(0, trace[path[0]][0])
        dist += trace[path[0]][1]

    return path, dist, num_visited
    # End your code (Part 2)
```

Part 3 UCS

```
    # Begin your code (Part 3)
    """
    1. load the csv file into rows
    2. store them into the dictionary edges in the format:
        {startID : list of attached (endID, distance)}
    3. implement UCS with priority quueue storing the accumulated weight
    4. backtrack the path by using the list trace
    5. return path, dist, num_visited
    """
    # raise NotImplementedError("To be implemented")
    edges = {}
    with open(edgeFile, newline='') as csvfile:
        rows = csv.reader(csvfile, delimiter=',')
        headers = next(rows)
        # edges: {startID : list of attached (endID, distance) s}
        for row in rows:
            if edges.get(int(row[0])) != None:
                edges[int(row[0])].append((int(row[1]), float(row[2])))
            else: edges[int(row[0])] = [(int(row[1]), float(row[2]))]
```

```
    queue = [[start, 0, 0]]  # ID, accumulated distance, parent
    trace = {}
    num_visited = 0
    while (True):
        flag = True
        # altered = False
        min = 0  # the index having the minimum accumulated distance
        for i in queue:
            if i[1] < queue[min][1]: min = queue.index(i)
```

109550182 莊婕妤

```
    if queue[min][0] in edges.keys():  # check if it is a leaf node
        for ID, distance in edges[queue[min][0]]:
            altered = False
            if ID in trace.keys(): continue  # check if the node is explored
            for i in range(len(queue)):
                if queue[i][0] == ID:
                    if queue[min][1] + distance < queue[i][1]:
                        queue[i][1] = queue[min][1] + distance
                        queue[i][2] = queue[min][0]
                    altered = True
            if altered == True: continue
            if ID == end:
                trace[end] = queue[min][0]
                dist = queue[min][1] + distance
                num_visited += 1
                flag = False
                break
            queue.append([ID, queue[min][1] + distance, queue[min][0]])
            num_visited += 1
```

```
    trace[queue[min][0]] = queue[min][2]
    if flag == False: break
    queue.pop(min)

path = [end]
while trace[path[0]] != 0:
    path.insert(0, trace[path[0]])

return path, dist, num_visited
# End your code (Part 3)
```

109550182 莊婕妤

Part 4 A*

```
# Begin your code (Part 4)
"""
1. determine the case number
2. load the csv files into dists & rows
3. store them into the dictionary edges in the format:
    {startID : list of attached (endID, distance)}
4. implement A*
5. backtrack the path by using the list trace
6. return path, dist, num_visited
"""
# raise NotImplementedError("To be implemented")
```

```
case = 0

with open(heuristicFile, mode='r') as input:
    reader = csv.reader(input)
    next(reader)
    dists = {int(rows[0]):[float(rows[1]),float(rows[2]),float(rows[3])] for rows in reader}

edges = {}
with open(edgeFile, newline='') as csvfile:
    rows = csv.reader(csvfile, delimiter=',')
    next(rows)
    # edges: {startID : list of attached (endID, distance) s}
    for row in rows:
        if edges.get(int(row[0])) != None:
            edges[int(row[0])].append((int(row[1]), float(row[2])))
        else: edges[int(row[0])] = [(int(row[1]), float(row[2]))]
```

```
queue = [[start, dists[start][case], 0, 0]]  # ID, accumulated distance, parent
trace = {}
num_visited = 0
while (True):
    flag = True
    # altered = False
    min = 0  # the index having the minimum accumulated distance
    for i in queue:
        if i[1] < queue[min][1]: min = queue.index(i)

    if queue[min][0] in edges.keys():  # check if it is a leaf node
        for ID, distance in edges[queue[min][0]]:
            altered = False
            if ID in trace.keys(): continue  # check if the node is explored
            for i in range(len(queue)):
                if queue[i][0] == ID:
                    if queue[min][2] + distance + dists[ID][case] < queue[i][1]:
                        queue[i][1] = queue[min][2] + distance + dists[ID][case]
                        queue[i][2] = queue[min][2] + distance
                        queue[i][3] = queue[min][0]
                    altered = True
            if altered == True: continue
```

```
            if ID == end:
                trace[end] = queue[min][0]
                dist = queue[min][2] + distance
                num_visited += 1
                flag = False
                break
            queue.append([ID, queue[min][2] + distance + dists[ID][case], queue[min][2] + distan
            num_visited += 1

    trace[queue[min][0]] = queue[min][3]
    if flag == False: break
    queue.pop(min)

path = [end]
while trace[path[0]] != 0:
    path.insert(0, trace[path[0]])

return path, dist, num_visited
# End your code (Part 4)
```

Part 6 Bonus

```
# Begin your code (Part 6)
"""
1. determine the case number
2. load the csv files into dists & rows
3. store them into the dictionary edges in the format:
    {startID : list of attached (endID, distance, speed limit)}
4. implement A* (time)
5. backtrack the path by using the list trace
6. return path, dist, num_visited
"""
# raise NotImplementedError("To be implemented")
case = 0

with open(heuristicFile, mode='r') as input:
    reader = csv.reader(input)
    next(reader)
    dists = {int(rows[0]):[math.sqrt(float(rows[1])),math.sqrt(float(rows[2])),math.sqrt(float(r
```

```
edges = {}
with open(edgeFile, newline='') as csvfile:
    rows = csv.reader(csvfile, delimiter=',')
    headers = next(rows)
    # edges: {startID : list of attached (endID, distance, speed limit)}
    for row in rows:
        if edges.get(int(row[0])) != None:
            edges[int(row[0])].append((int(row[1]), float(row[2]), float(row[3]) * 1000 / 3600))
        else: edges[int(row[0])] = [(int(row[1]), float(row[2]), float(row[3]) * 1000 / 3600)]
```

109550182 莊婕妤

```
    queue = [[start, dists[start][case], 0, 0]]  # ID, accumulated distance, parent
    trace = {}
    num_visited = 0
    while (True):
        flag = True
        # altered = False
        min = 0  # the index having the minimum accumulated distance
        for i in queue:
            if i[1] < queue[min][1]: min = queue.index(i)

        if queue[min][0] in edges.keys():  # check if it is a leaf node
            for ID, distance, speed in edges[queue[min][0]]:
                altered = False
                if ID in trace.keys(): continue  # check if the node is explored
                for i in range(len(queue)):
                    if queue[i][0] == ID:
                        if queue[min][2] + distance/speed + dists[ID][case] < queue[i][1]:
                            queue[i][1] = queue[min][2] + distance/speed + dists[ID][case]
                            queue[i][2] = queue[min][2] + distance/speed
                            queue[i][3] = queue[min][0]
                        altered = True
```

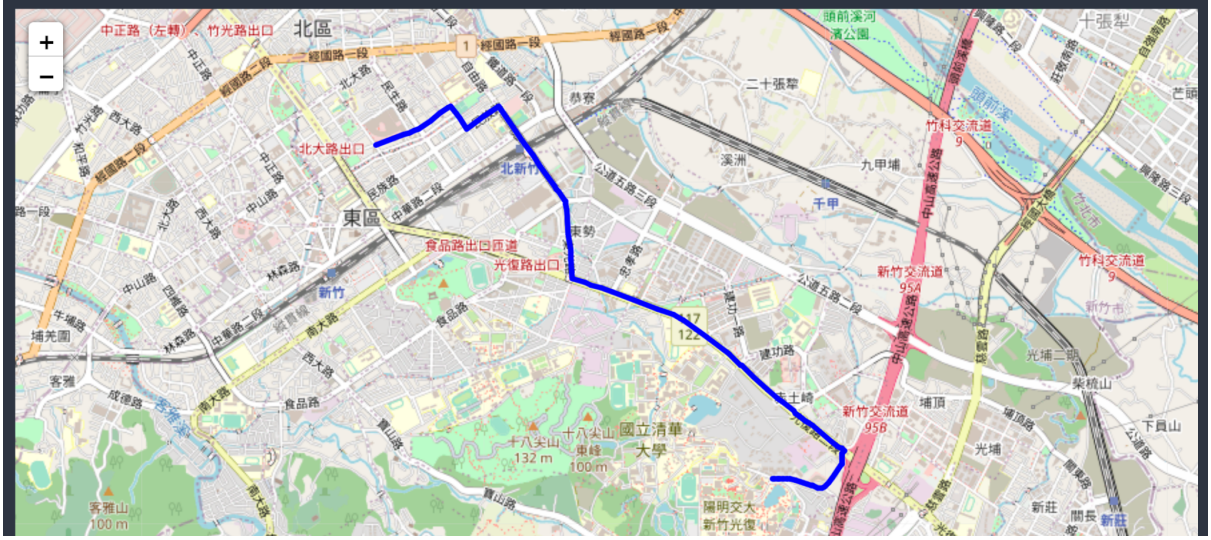
```
                if altered == True: continue
                if ID == end:
                    trace[end] = queue[min][0]
                    dist = queue[min][2] + distance/speed
                    num_visited += 1
                    flag = False
                    break
                queue.append([ID, queue[min][2] + distance/speed + dists[ID][case], queue[min][2] +
                num_visited += 1

        trace[queue[min][0]] = queue[min][3]
        if flag == False: break
        queue.pop(min)

    path = [end]
    while trace[path[0]] != 0:
        path.insert(0, trace[path[0]])

    return path, dist, num_visited
    # End your code (Part 6)
```

Part II. Results & Analysis (12%)

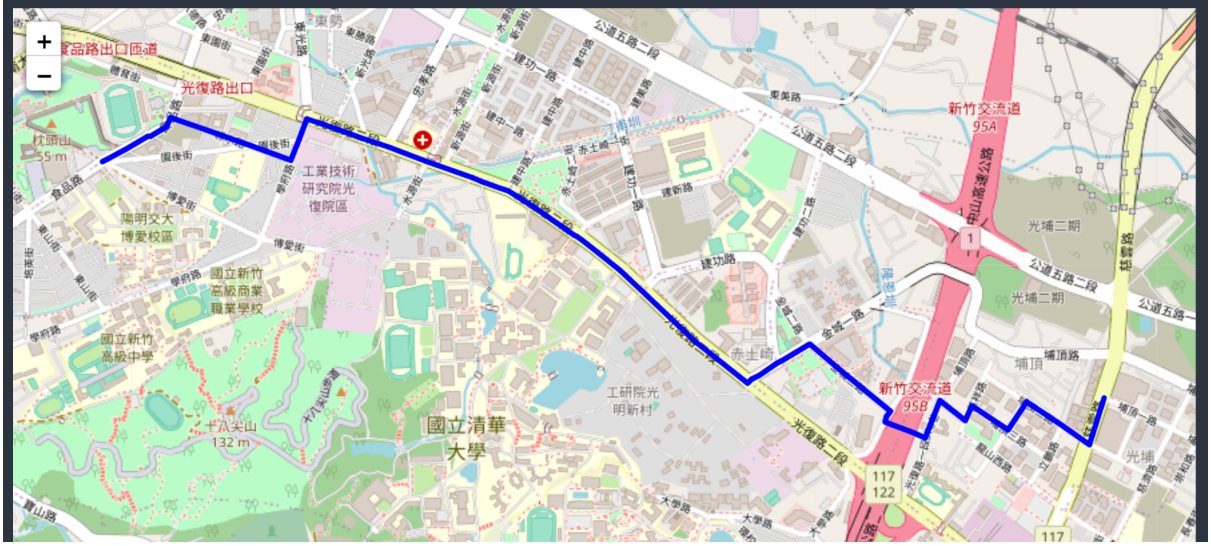
BFS

Compared to other route finding methods, BFS seems to be a better approach than DFS, but not that fast as UCS and A* search.

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

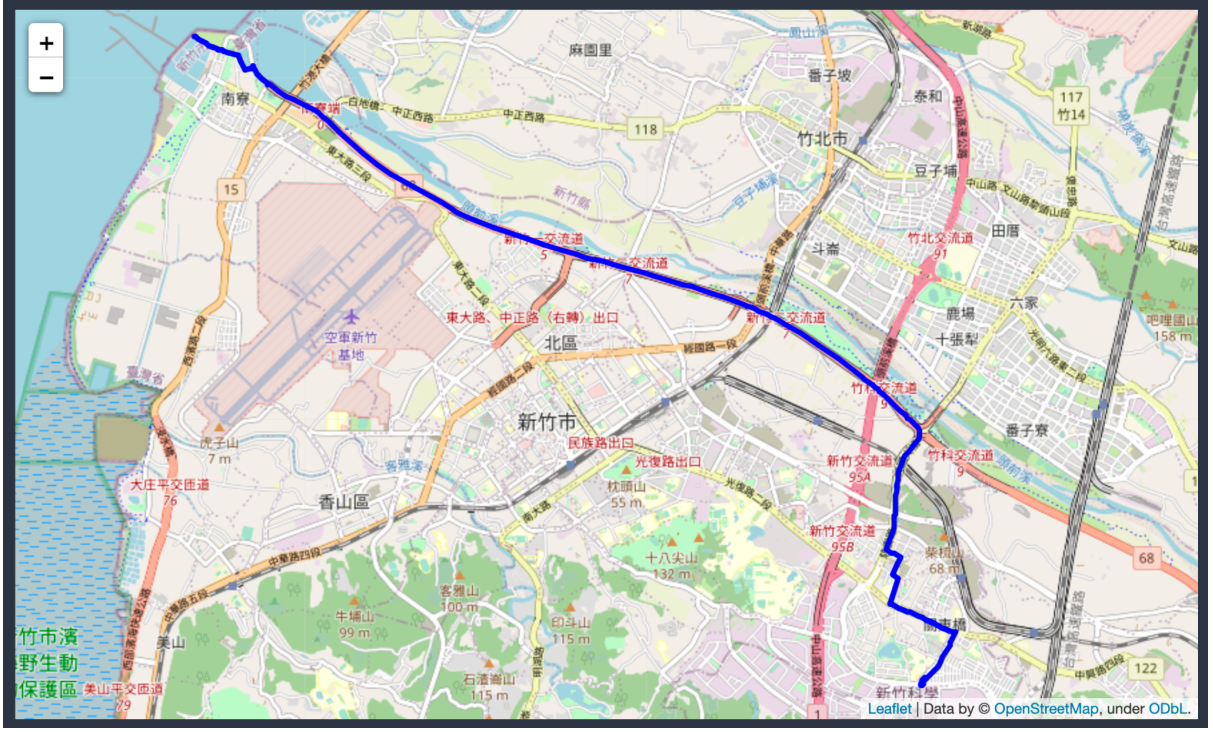


The number of nodes in the path found by BFS: 60
Total distance of path found by BFS: 4215.521000000001 m
The number of visited nodes in BFS: 4606



109550182 莊婕妤

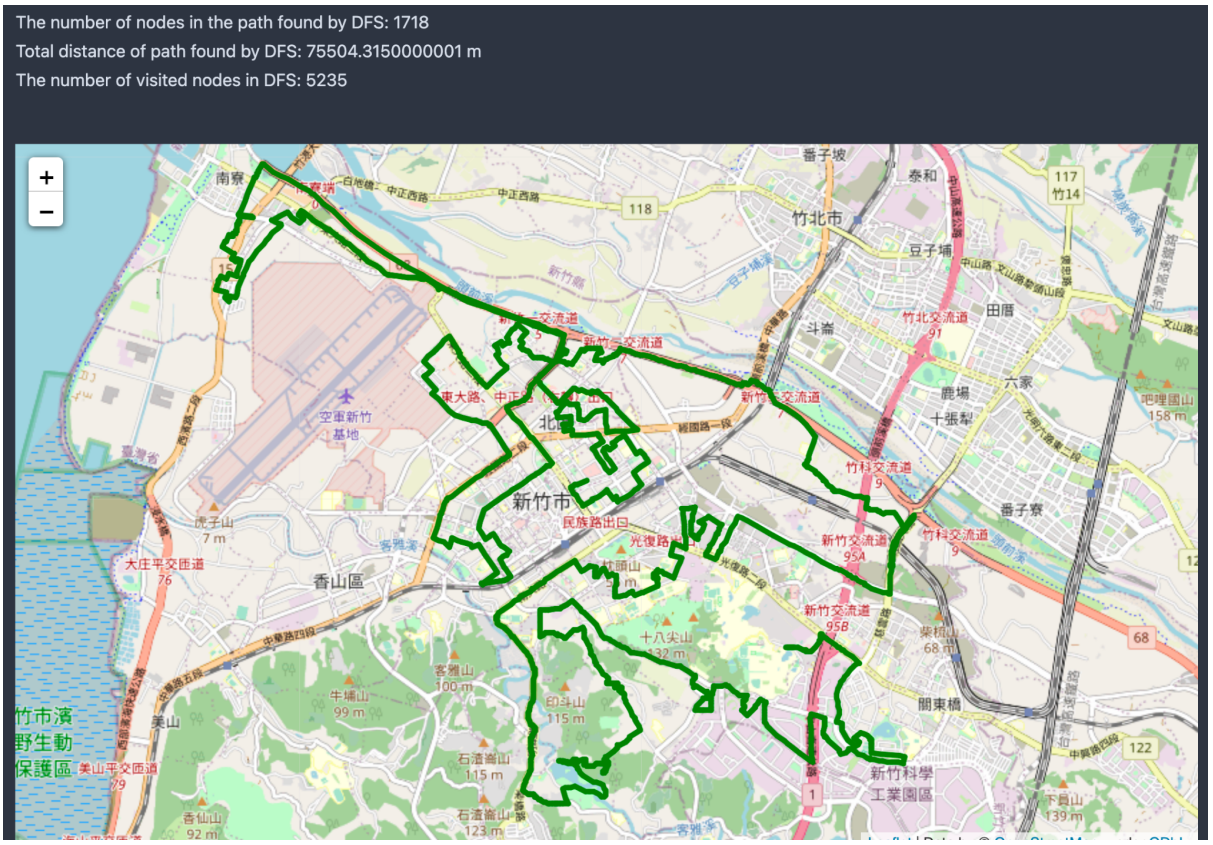
The number of nodes in the path found by BFS: 183
Total distance of path found by BFS: 15442.394999999995 m
The number of visited nodes in BFS: 11241

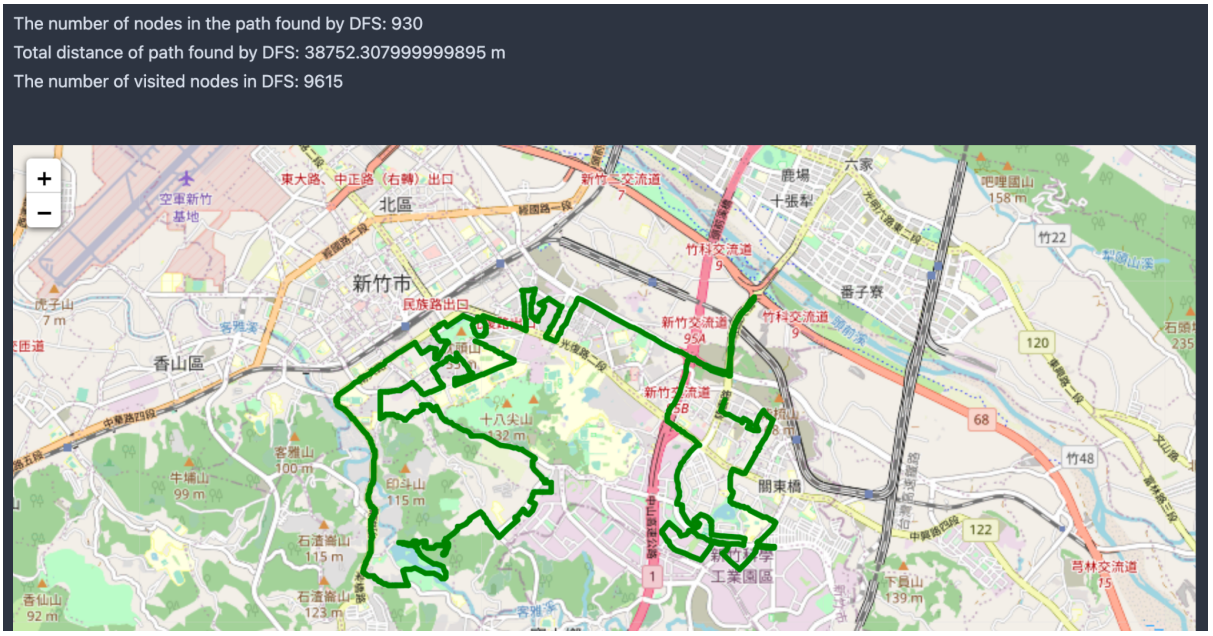


109550182 莊婕妤

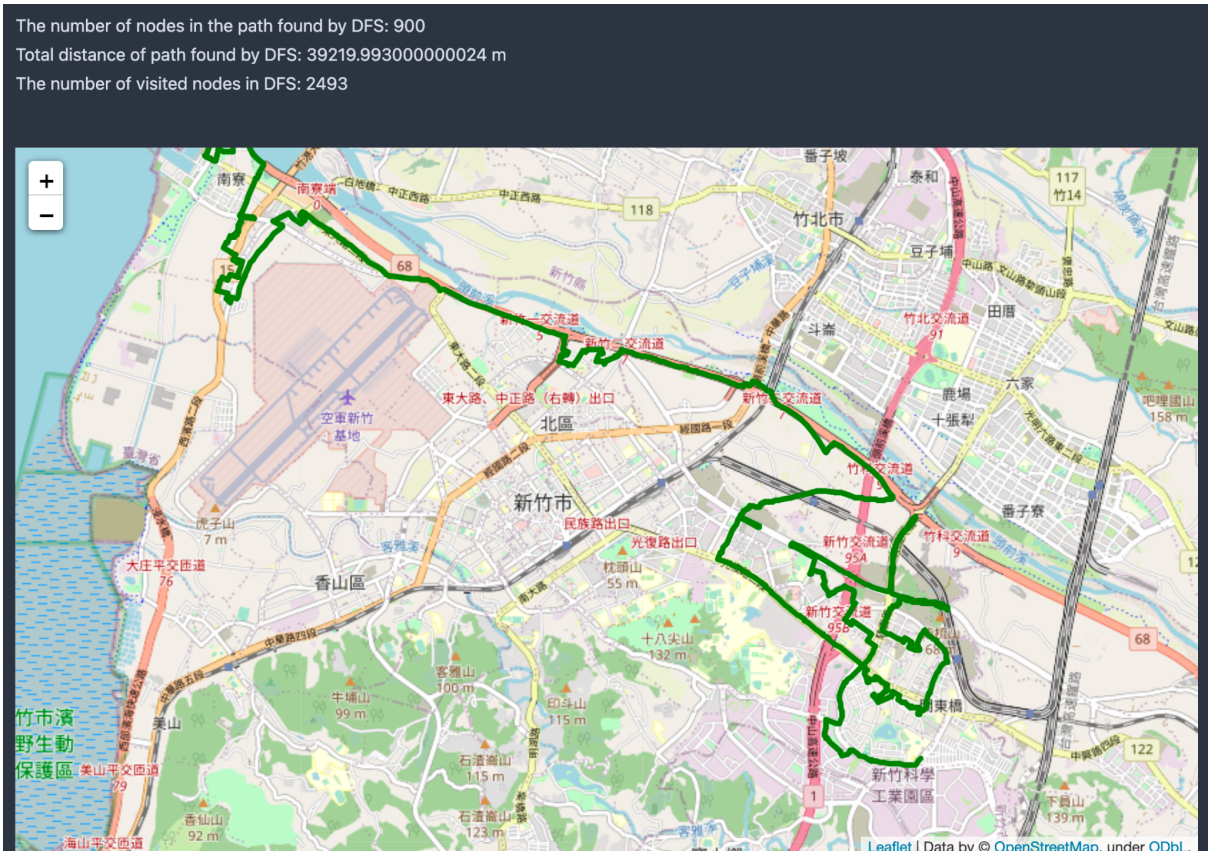
DFS (Stack)

Compared to other route finding methods, the distance of path found by DFS is much longer than other methods, which is not that effective.





109550182 莊婕妤

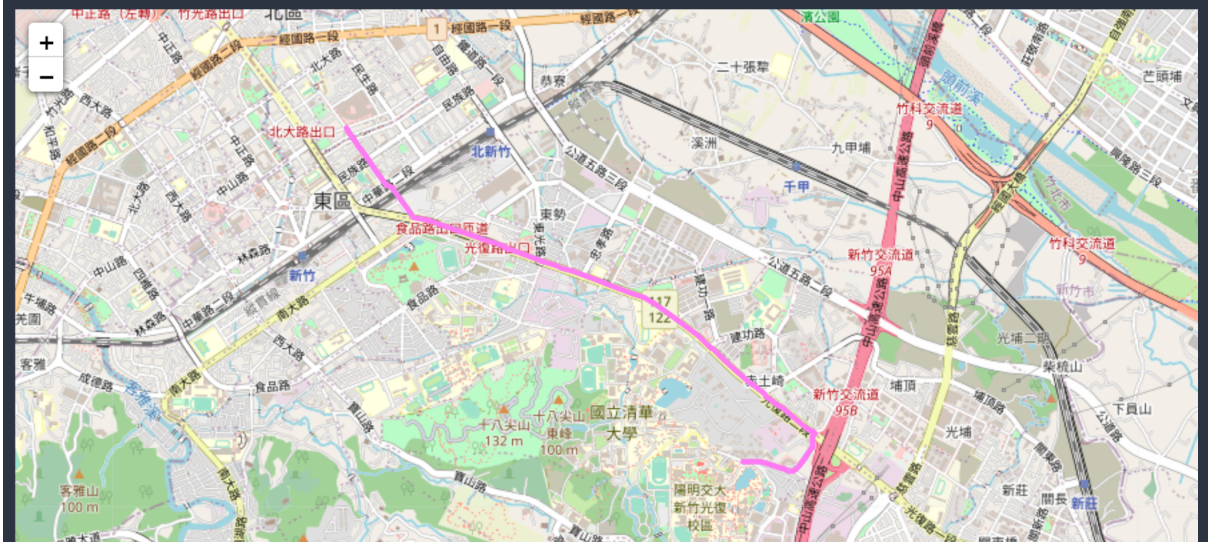


109550182 莊婕妤

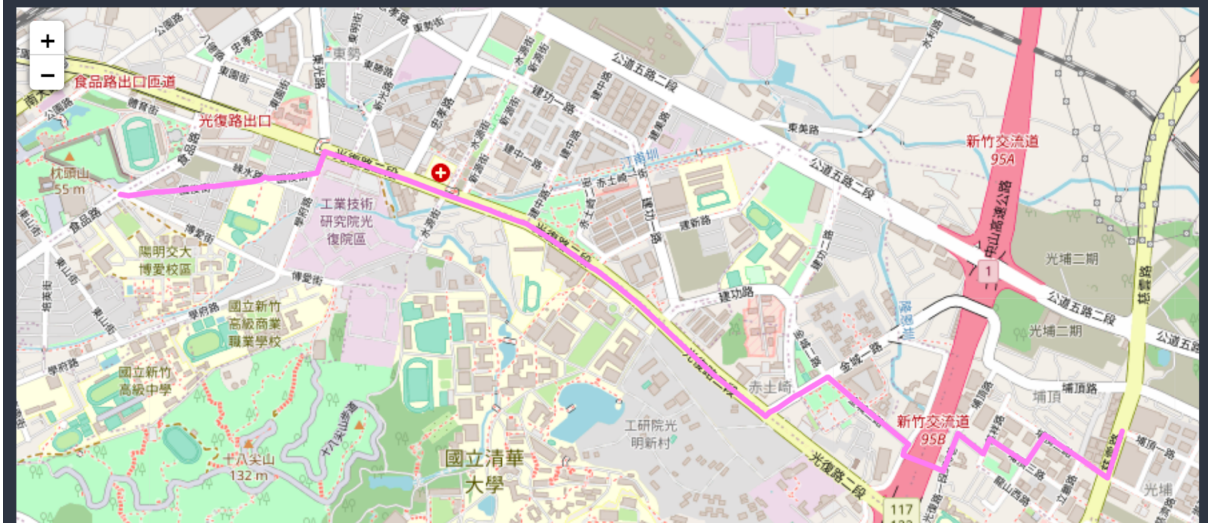
UCS

The number of nodes in the path and the path distance is almost the same with A* search. However, it visited much more nodes than A* search. Compared to BFS and DFS, UCS still got a better performance.

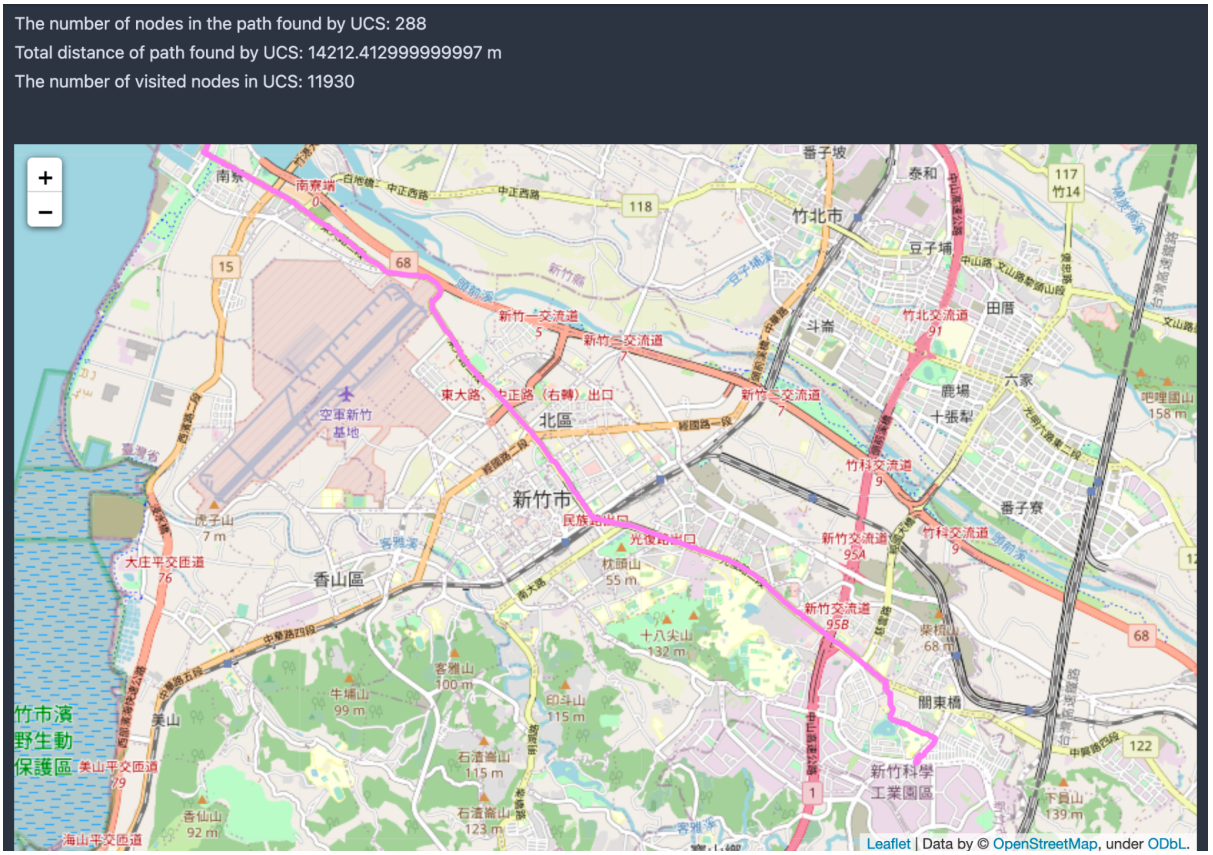
The number of nodes in the path found by UCS: 89
Total distance of path found by UCS: 4367.881 m
The number of visited nodes in UCS: 5076



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: 6901



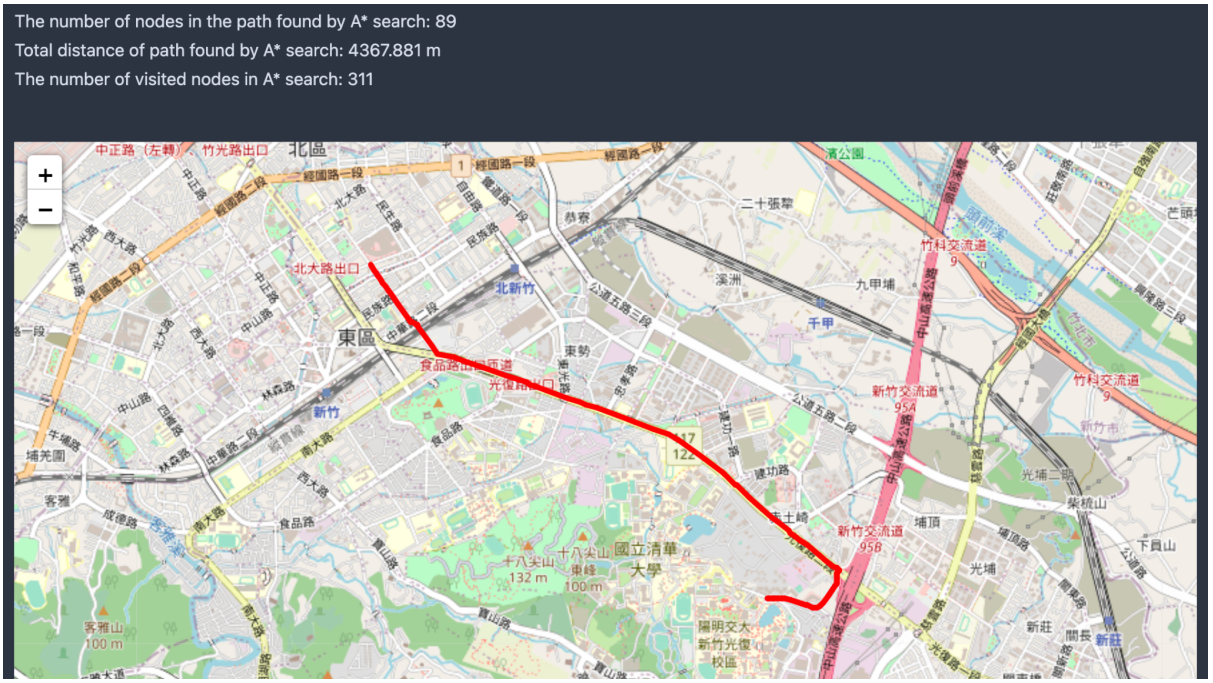
109550182 莊婕妤

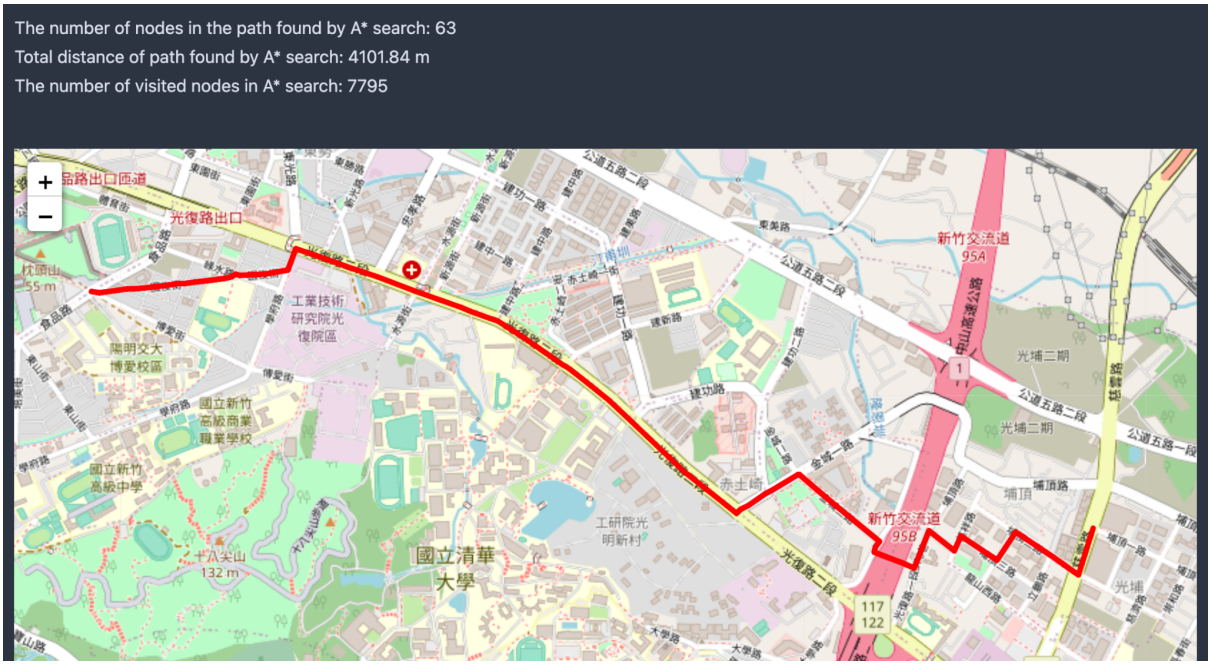


109550182 莊婕妤

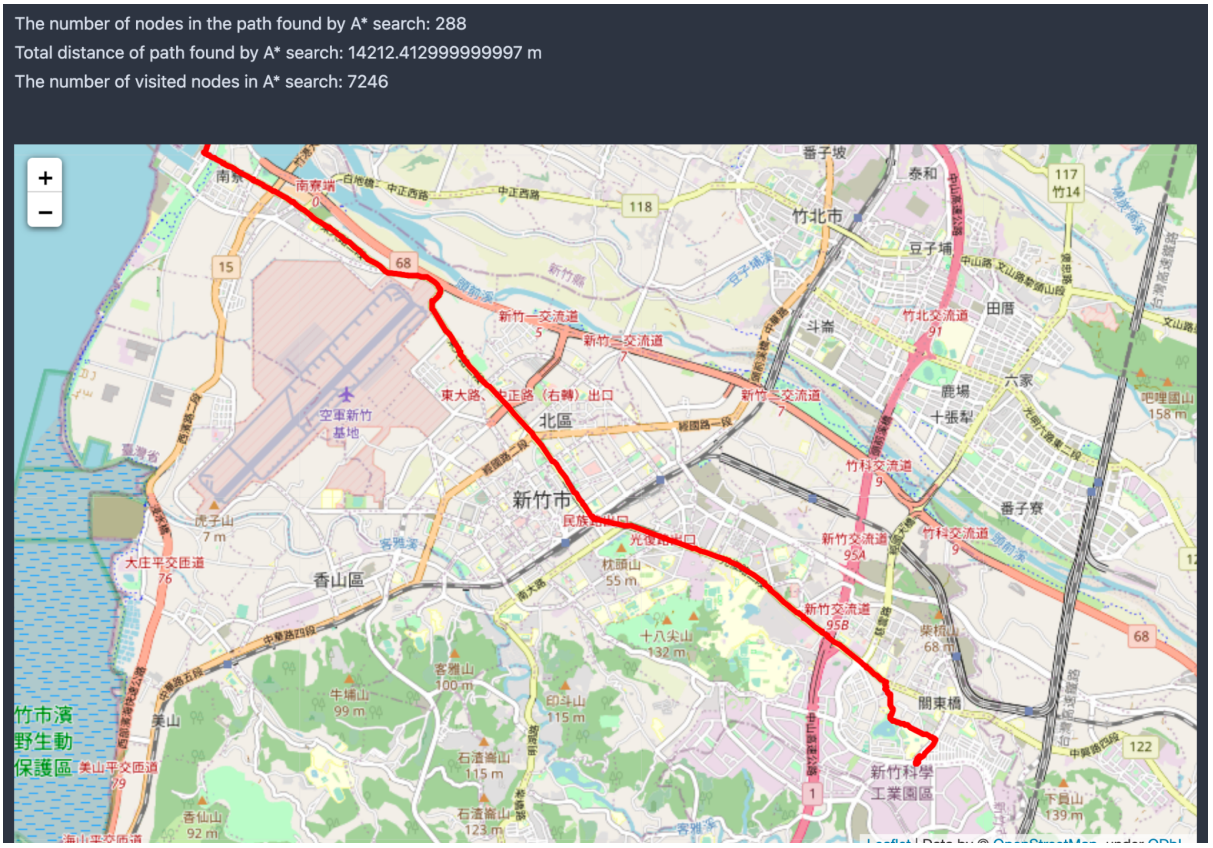
A*

The number of nodes in the path and the path distance is almost the same with UCS, while visiting less nodes. Compared to the other three searching methods, this is the best route finding algorithm.





109550182 莊婕妤

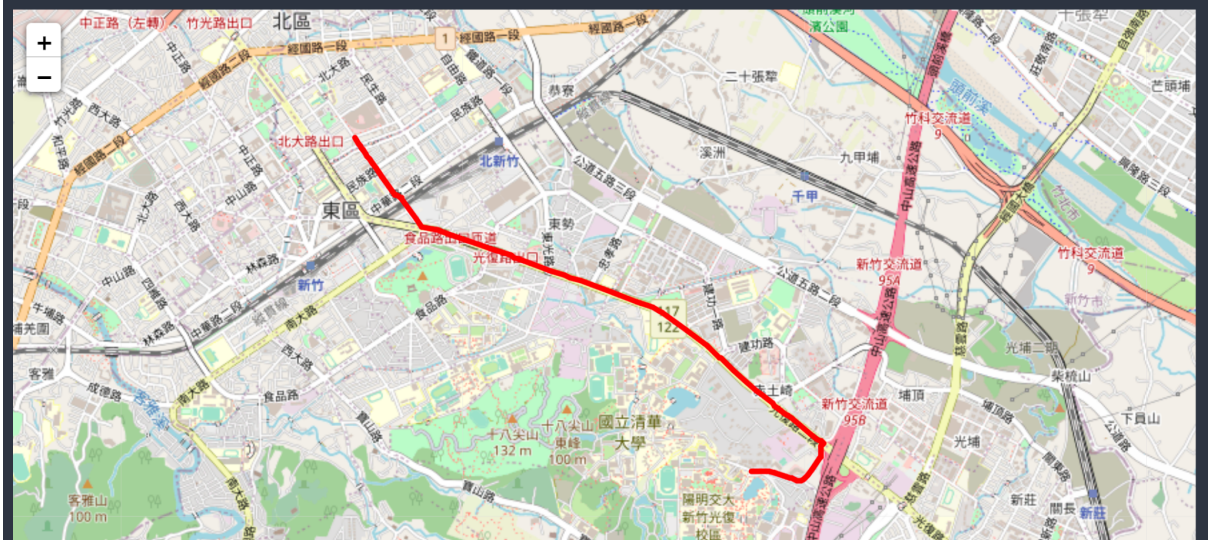


109550182 莊婕妤

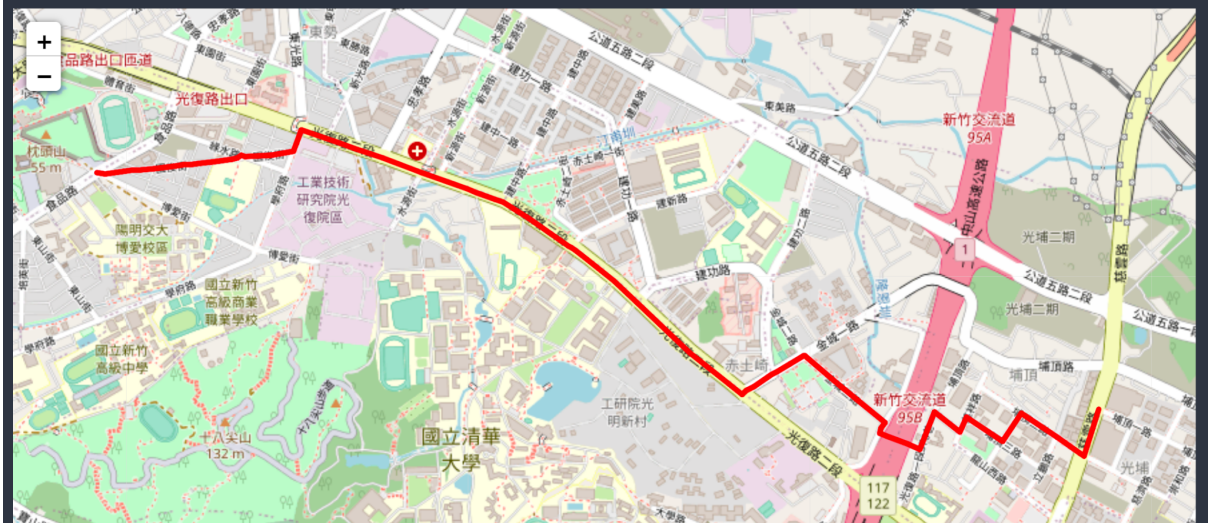
Bonus: A* Time

The new heuristic function I defined is taking the square root of the straight-line distance. The number of nodes in the path remains unchanged comparing with the original A* search, while it visited much more nodes.

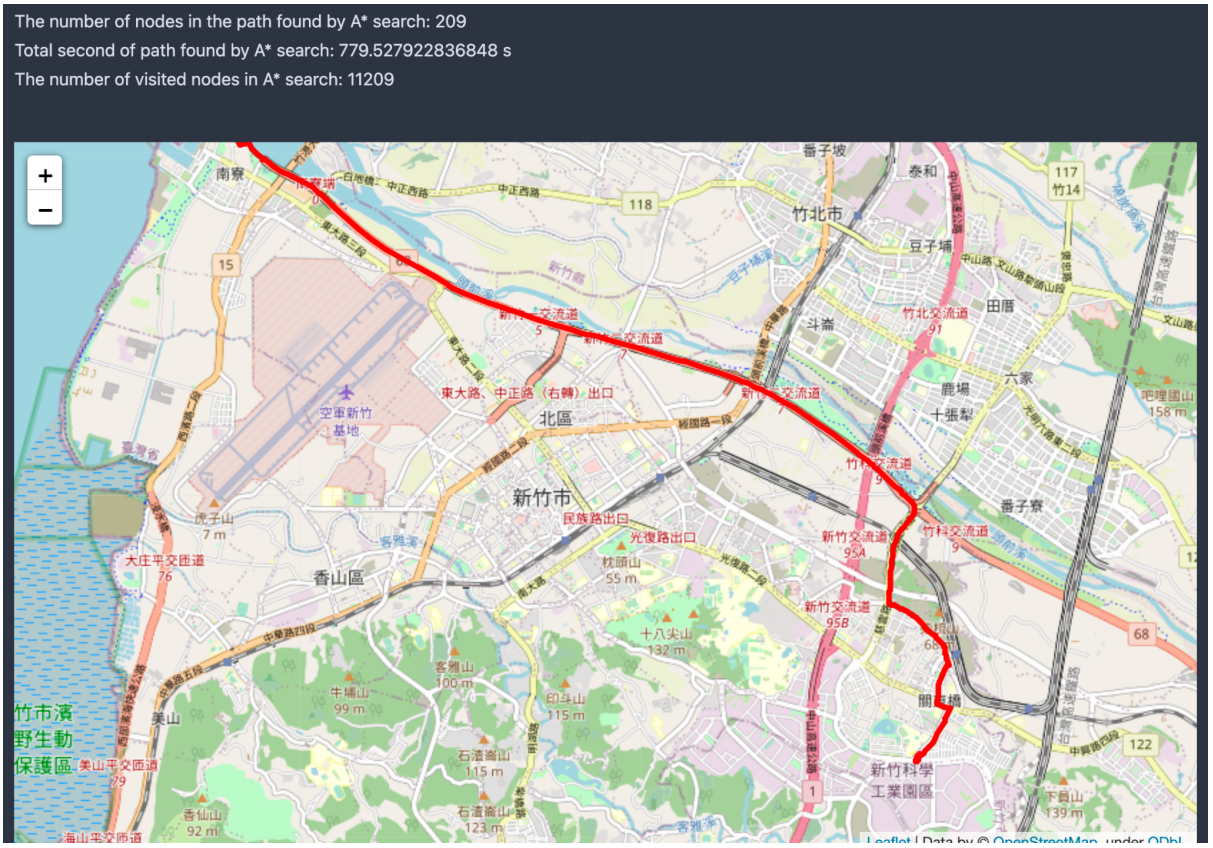
The number of nodes in the path found by A* search: 89
Total second of path found by A* search: 320.87823163083164 s
The number of visited nodes in A* search: 3996



The number of nodes in the path found by A* search: 63
Total second of path found by A* search: 304.44366343603014 s
The number of visited nodes in A* search: 7377



109550182 莊婕妤



109550182 莊婕妤

Part III. Answer the questions (12%)

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

   One problem I encountered is that I kept getting the wrong answer on UCS for the second and third test data, while the first one was correct. I reviewed my code again and tried to simulate all circumstances I might encounter. Then I found out that there are some situations where I may store a larger accumulated distance to the priority queue. I corrected my code and solved the problem.

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.

   Apart from speed limit and distance, I think we should also take the current traffic flow into consideration, since it highly affects the traveling time. For instance, no matter how high the speed limit is, if you're stuck in a traffic jam, it still takes a lot of time.

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, I suggest that we can use the longitude and latitude to identify the nodes. For localization, I think we can combine data from different sensors such as LIDAR and camera in order to detect the vehicle's surroundings.

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.

   With respect to what I've mentioned in problem 2, I'd like to take the traffic flow into consideration. Therefore, I think we can design a heuristic function as below :

   (Speed Limit + Number of Cars) / Distance

   This way, if there are many cars, the weight will be higher.