Part I. Implementation

- Part 1

```python
# Begin your code (Part 1)
"""
graph: to create a graph
parents: to trace the path
dis: to record the distance from start_node to current_node
queue: nodes which can be visited
visited: nodes which have been visited

1. Use csv.reader to read the csv file and create a graph
   according to the csv file, then store into graph{}, whose
   format is graph[from_node][to_node] = distance
2. Search neighbors of start_node and set their parents and distance
   Then, put nodes into queue
3. Following FIFO, pop first element of queue. Check whether the node has
   been visited, then check if the node is end_node. If the node has not
   been visited and is not end_node, add it to visited and search its
   neighbors. Put its neighbors to queue and update their parents and distance
4. Trace the path according parents
"""
with open(edgeFile) as edges:
    rows = csv.reader(edges)
    headers = next(rows)
```

```python
    graph = {}
    parents = dict()
    dis = dict()
    queue = []
    visited = []

    for row in rows:
        if row[0] in graph:
            graph[row[0]][row[1]] = row[2]
        else:
            graph[row[0]] = dict()
            graph[row[0]][row[1]] = row[2]

    dis[start] = 0

    for node in graph[start]:
        if not parents.get(node):
            parents[node] = start
        if not dis.get(node):
            dis[node] = graph[start][node]
        queue.append(node)

    while queue:
        node = queue.pop(0)
        if not graph.get(node):
            continue
        else:
```

```python
        else:
            if node not in visited:
                if node == end:
                    break
                else:
                    visited.append(node)
                    for n in graph[node]:
                        if not parents.get(n):
                            parents[n] = node
                        if not dis.get(n):
                            dis[n] = float(graph[node][n]) + float(dis[node])
                        queue.append(n)

    path = []
    key = end

    while parents[key] != start:
        path.append(int(key))
        key = parents[key]
    path.append(int(key))
    path.append(int(start))
    path.reverse()

    return path, dis[end], len(visited)
```

- Part 2

```python
# Begin your code (Part 2)
"""
graph: to create a graph
parents: to trace the path
dis: to record the distance from start_node to current_node
queue: nodes which can be visited
visited: nodes which have been visited

1. Use csv.reader to read the csv file and create a graph
   according to the csv file, then store into graph{}, whose
   format is graph[from_node][to_node] = distance
2. Search neighbors of start_node and set their parents and distance
   Then, put nodes into queue
3. Following LIFO, pop last element of queue. Check whether the node has
   been visited, then check if the node is end_node. If the node has not
   been visited and is not end_node, add it to visited and search its
   neighbors. Put its neighbors to queue and update their parents and distance
4. Trace the path according parents
"""
with open(edgeFile) as edges:
    rows = csv.reader(edges)
    headers = next(rows)
```

```python
graph = {}
parents = dict()
dis = dict()
queue = []
visited = []

for row in rows:
    if row[0] in graph:
        graph[row[0]][row[1]] = row[2]
    else:
        graph[row[0]] = dict()
        graph[row[0]][row[1]] = row[2]

dis[start] = 0

for node in graph[start]:
    if not parents.get(node):
        parents[node] = start
    if not dis.get(node):
        dis[node] = graph[start][node]
    queue.append(node)

while queue:
    node = queue.pop()
    if not graph.get(node):
        continue
```

```python
    else:
        if node not in visited:
            if node == end:
                break
            else:
                visited.append(node)
                for n in graph[node]:
                    if not parents.get(n):
                        parents[n] = node
                    if not dis.get(n):
                        dis[n] = float(graph[node][n]) + float(dis[node])
                    queue.append(n)

path = []
key = end

while parents[key] != start:
    path.append(int(key))
    key = parents[key]

path.append(int(key))
path.append(int(start))
path.reverse()

return path, dis[end], len(visited)
```

- Part 3

```python
# Begin your code (Part 3)
"""
graph: to create a graph
parents: to trace the path
frontier: nodes which can be visited. A priority queue ordered by ucs_w
explored: nodes which have been visited

1. Use csv.reader to read the csv file and create a graph
   according to the csv file, then store into graph{}, whose
   format is graph[from_node][to_node] = distance
2. Pop first element of queue, which has smallest weight. Add the node to explored.
   Search its neighbors and check them whether they have been visited. If not, update
   their weight and parents
3. Sort frontier to ensure the first element has the smallest weight
4. Trace the path according parents
"""
with open(edgeFile) as edges:
    rows = csv.reader(edges)
    headers = next(rows)

    graph = {}
    parents = dict()
    frontier = []
    explored = []
```

```python
        for row in rows:
            if row[0] in graph:
                graph[row[0]][row[1]] = row[2]
            else:
                graph[row[0]] = dict()
                graph[row[0]][row[1]] = row[2]

        frontier.append([0, start])

        while frontier:

            tmp = frontier.pop(0)
            ucs_w = tmp[0]
            current_node = tmp[1]

            if not graph.get(current_node):
                continue
            else:
                explored.append(current_node)

                if current_node == end:
                    dis = float(ucs_w)
                    break

                for node in graph[current_node]:
                    if node not in explored:
                        new_weight = ucs_w + float(graph[current_node][node])
```

```python
        for node in graph[current_node]:
            if node not in explored:
                new_weight = ucs_w + float(graph[current_node][node])
                frontier.append([new_weight, node])
                parents[node] = current_node
        frontier = sorted(frontier)

    path = []
    key = end

    while parents[key] != start:
        path.append(int(key))
        key = parents[key]

    path.append(int(key))
    path.append(int(start))
    path.reverse()

    return path, dis, len(explored)
```

- Part 4

```python
# Begin your code (Part 4)
"""
graph: to create a graph
parents: to trace the path
frontier: nodes which can be visited. A priority queue ordered by h_n
explored: nodes which have been visited
h1_weight: the heuristic function which is from National Yang Ming Chiao
           University to Big City Shopping Mall
h2_weight: the heuristic function which is from Hsinchu Zoo to COSTCO Hsinchu Store
h3_weight: the heuristic function which is from National Experimental High School At
           Hsinchu Science Park to Nanliao Fighing Port
h_n: to store current heuristic function
g_n: to store the distance from start_node to current_node

1. Use csv.reader to read edges.csv and create a graph
   according to the csv file, then store the data into graph{},
   whose format is graph[from_node][to_node] = distance
2. Use csv.reader to read heuristic.csv and create dictionaries. Store the data into
   respective dictionary, whose format is hx_weight[node] = distance from node to end_node.
3. If clause determines which heuristic function is going to use
4. Pop first element of queue, which has smallest weight. Add the node to explored.
   Search its neighbors and check them whether they have been visited. If not, update
   their weight and parents
3. Sort frontier to ensure the first element has the smallest weight
4. Trace the path according parents
"""
```

```python
with open(edgeFile) as edges:
    rows = csv.reader(edges)
    headers = next(rows)

    graph = {}
    h1_weight = {}
    h2_weight = {}
    h3_weight = {}
    h_n = {}

    for row in rows:
        if row[0] in graph:
            graph[row[0]][row[1]] = row[2]
        else:
            graph[row[0]] = dict()
            graph[row[0]][row[1]] = row[2]

    with open(heuristicFile) as hFile:
        r = csv.reader(hFile)
        headers = next(r)

        for rr in r:
            h1_weight[rr[0]] = rr[1]
            h2_weight[rr[0]] = rr[2]
            h3_weight[rr[0]] = rr[3]
```

```python
        if end == '1079387396':
            h_n = h1_weight
        elif end == '1737223506':
            h_n = h2_weight
        elif end == '8513026827':
            h_n = h3_weight

        parents = {}
        g_n = {}
        frontier = []
        explored = []

        g_n[start] = 0
        frontier.append([g_n[start] + float(h_n[start]), start])

        while frontier:
            tmp = frontier.pop(0)
            weight = tmp[0]
            current_node = tmp[1]

            if not graph.get(current_node):
                continue
            else:
                explored.append(current_node)

                if current_node == end:
                    break
```

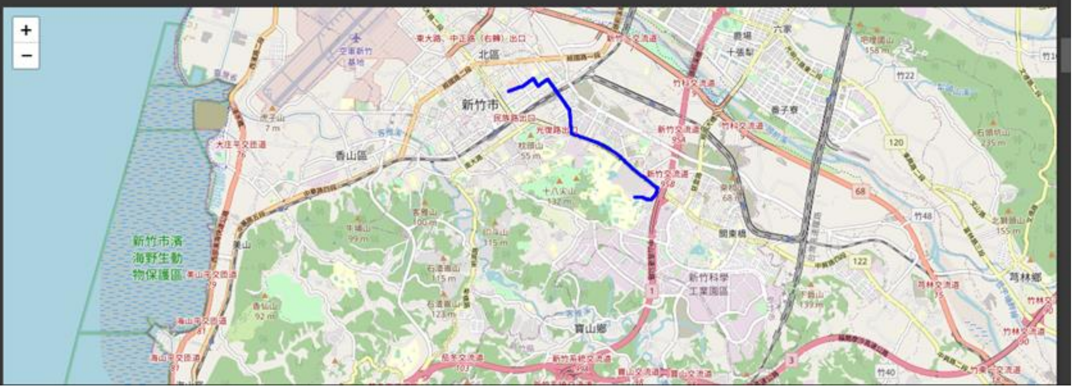
```python
                continue
            else:
                explored.append(current_node)

                if current_node == end:
                    break

                for node in graph[current_node]:
                    g_n[node] = float(g_n[current_node]) + float(graph[current_node][node])
                    f_n = g_n[node] + float(h_n[node])
                    if node not in explored:
                        frontier.append([f_n, node])
                        parents[node] = current_node
                frontier = sorted(frontier)

        path = []
        key = end

        while parents[key] != start:
            path.append(int(key))
            key = parents[key]

        path.append(int(key))
        path.append(int(start))
        path.reverse()

        return path, float(g_n[end]), len(explored)
```

Part II. Results & Analysis

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: 4978.882000000005 m
The number of visited nodes in BFS: 4266
```

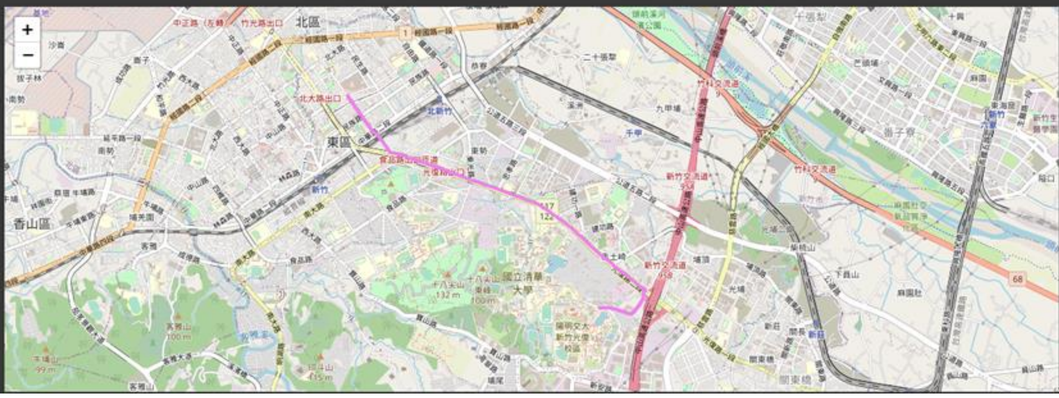


- DFS

```
The number of nodes in the path found by DFS: 222
Total distance of path found by DFS: 11061.247 m
The number of visited nodes in DFS: 4217
```

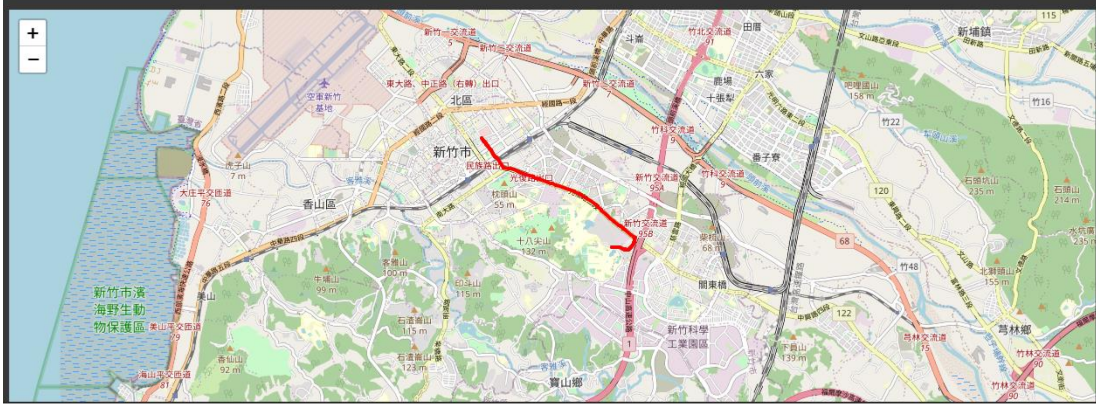
- UCS

```
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: 10897
```



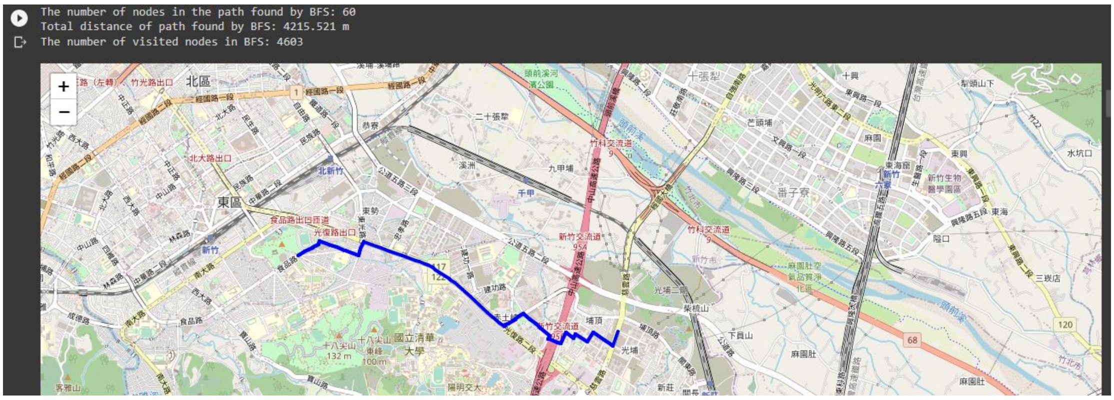
- A*

```
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: 272
```

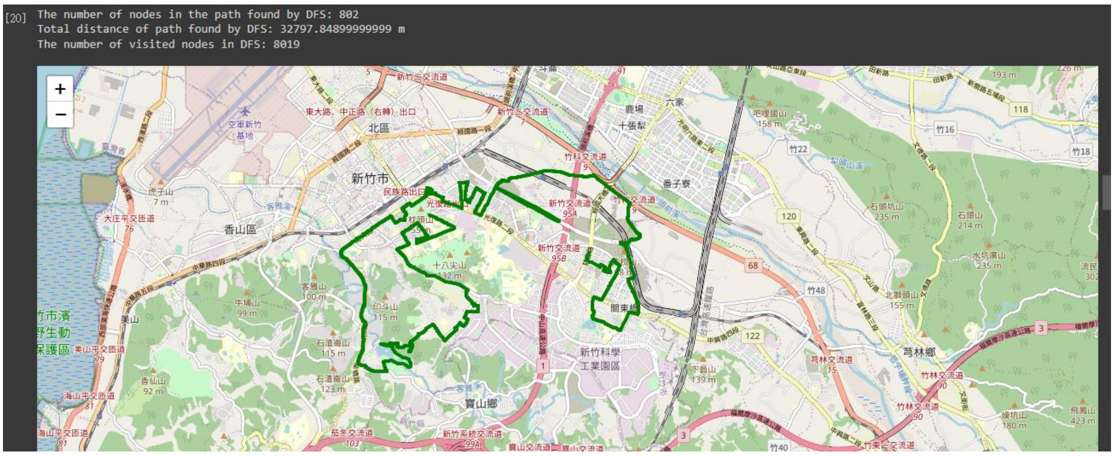


Test2: from Hsinchu Zoo (ID: 426882161) to COSTCO Hsinchu Store (ID: 1737223506)

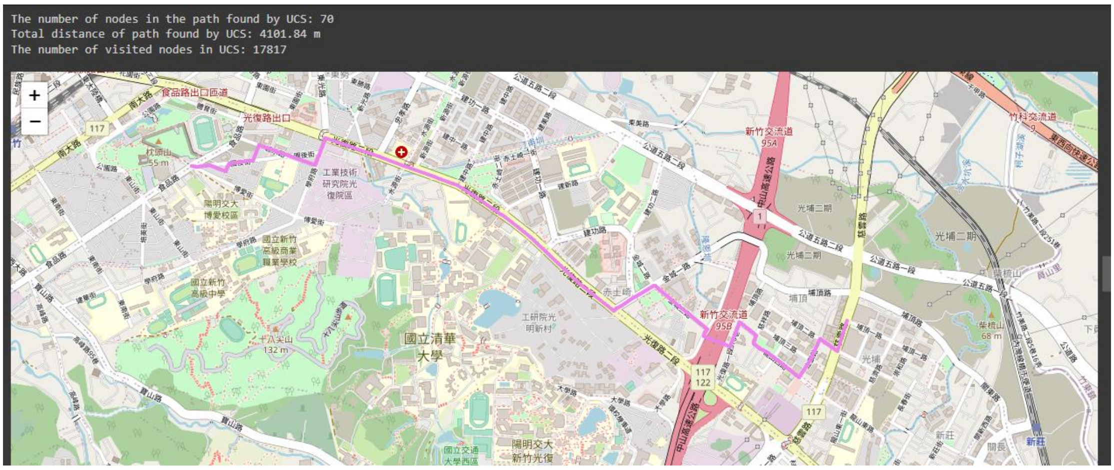
- BFS



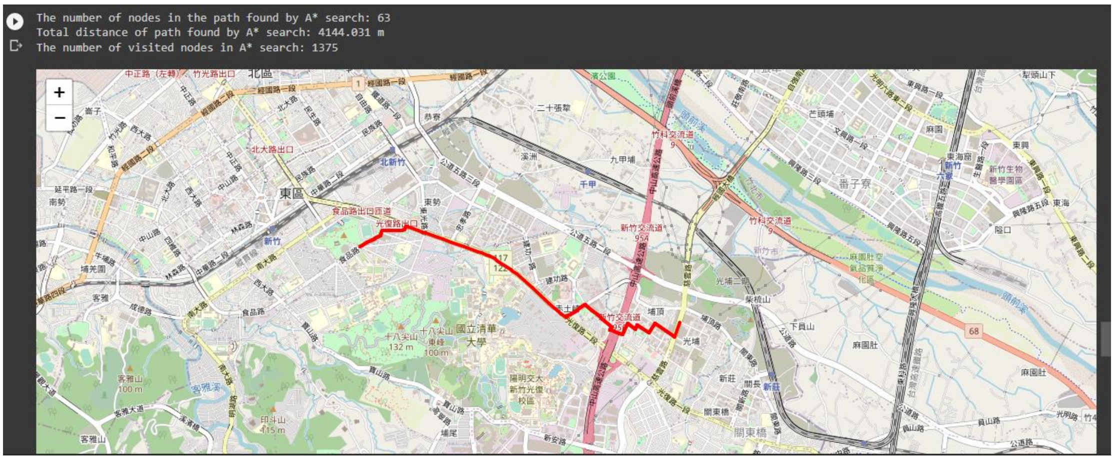
- DFS



- UCS

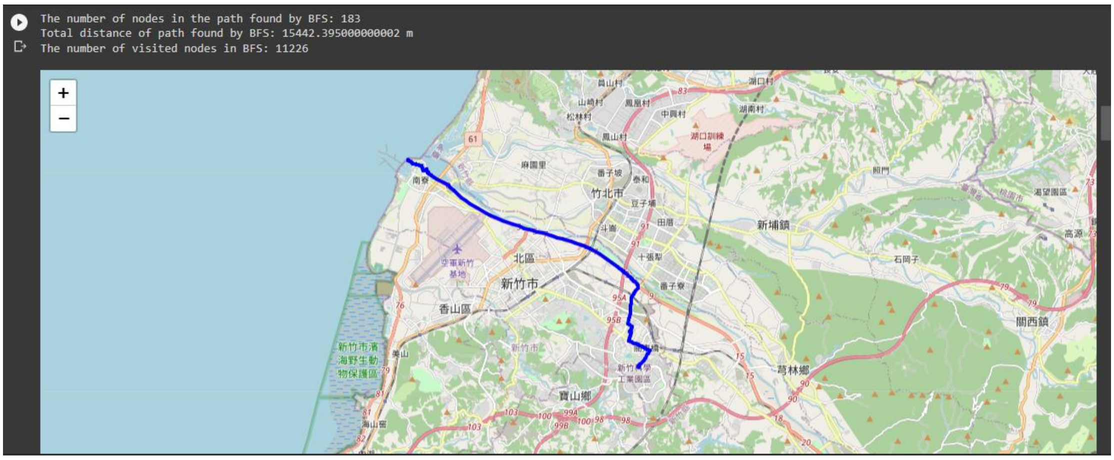


- A*

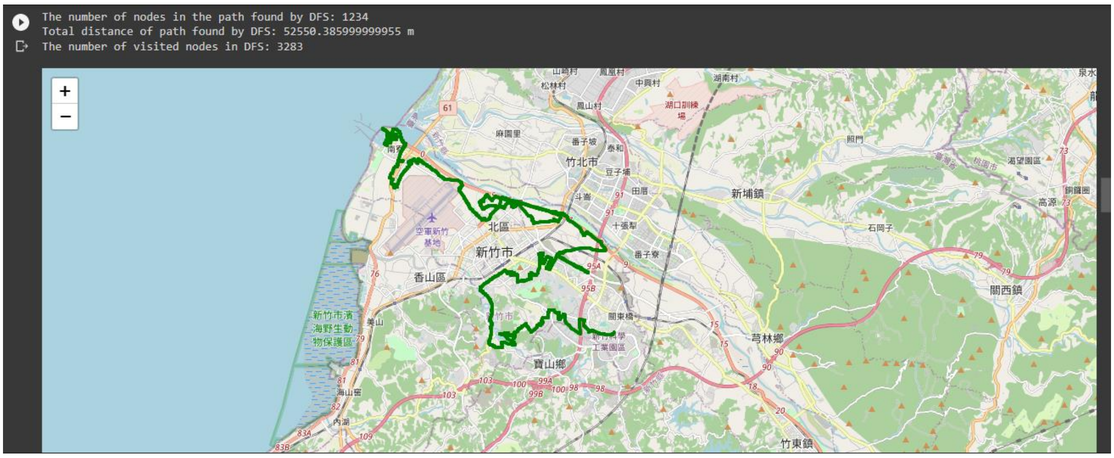


Test3: from National Experimental High School At Hsinchu Science Park (ID: 1718165260) to Nanliao Fighing Port (ID: 8513026827)

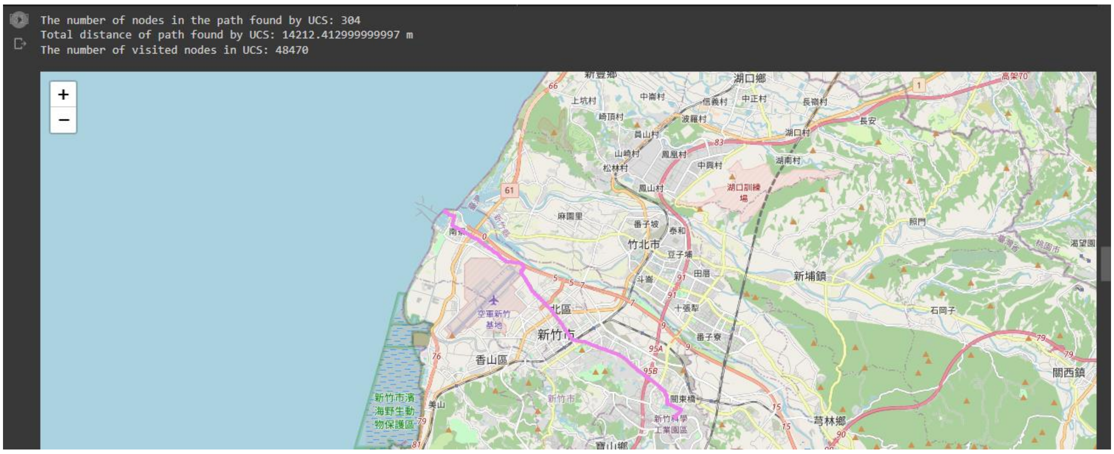
- BFS



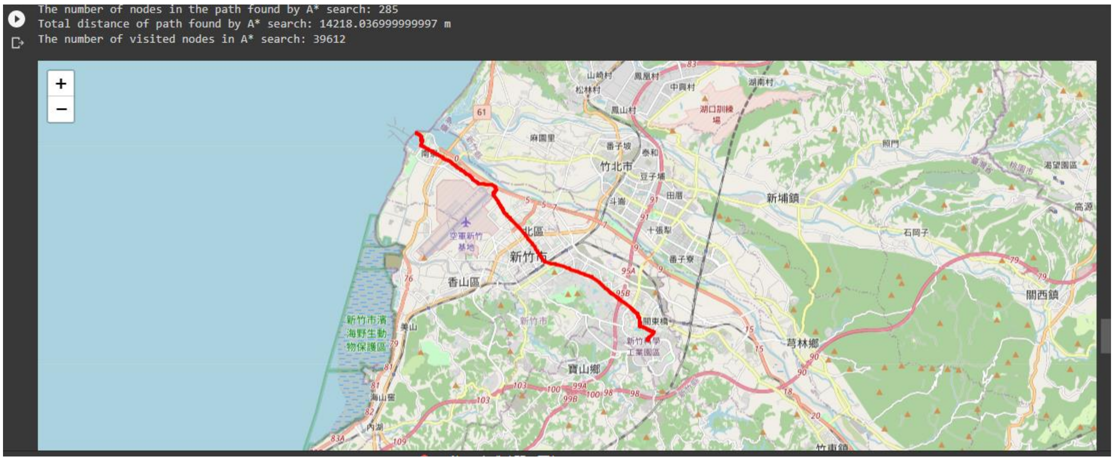
- DFS



- UCS



- A*



Analysis:

Total distance of path: UCS = A* > BFS > DFS

I think I did somewhere wrong in A*, so in test2 and test3, the distance of path found by A* is longer than that by USC.

Part III. Question Answering

1. When I tried to read the csv file, I save each row in a dictionary at the beginning. After I finished bfs, I ran my code and found error when I create my graph. It turned out to be that I save the title in the csv file, so it caused keyerror. I use next the pointer to solve the problem.

2. I think traffic flow is another attribute that is essential for route finding in the real world. If we take traffic flow into consideration, we could avoid the traffic jam. Take another route with light traffic flow, sometimes we can arrive our destination more quickly.
3. Mapping: Satelite
   Localization: GPS
4. A food delivery would accept not merely one order simultaneously. It is one attribute can be taken into consideration. Another attribute is also important, which is the time a store takes to make food.