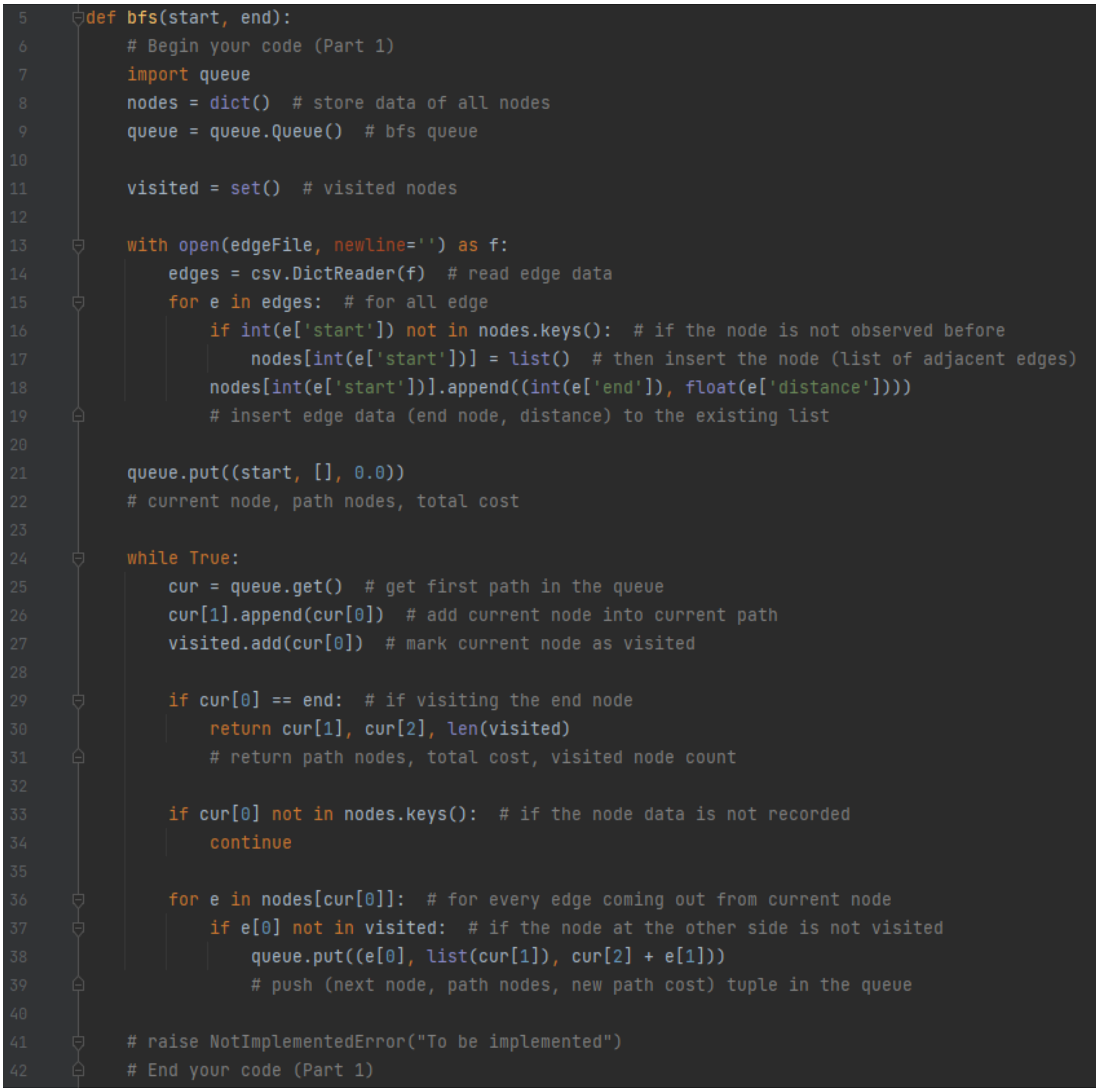
HW2 – Report

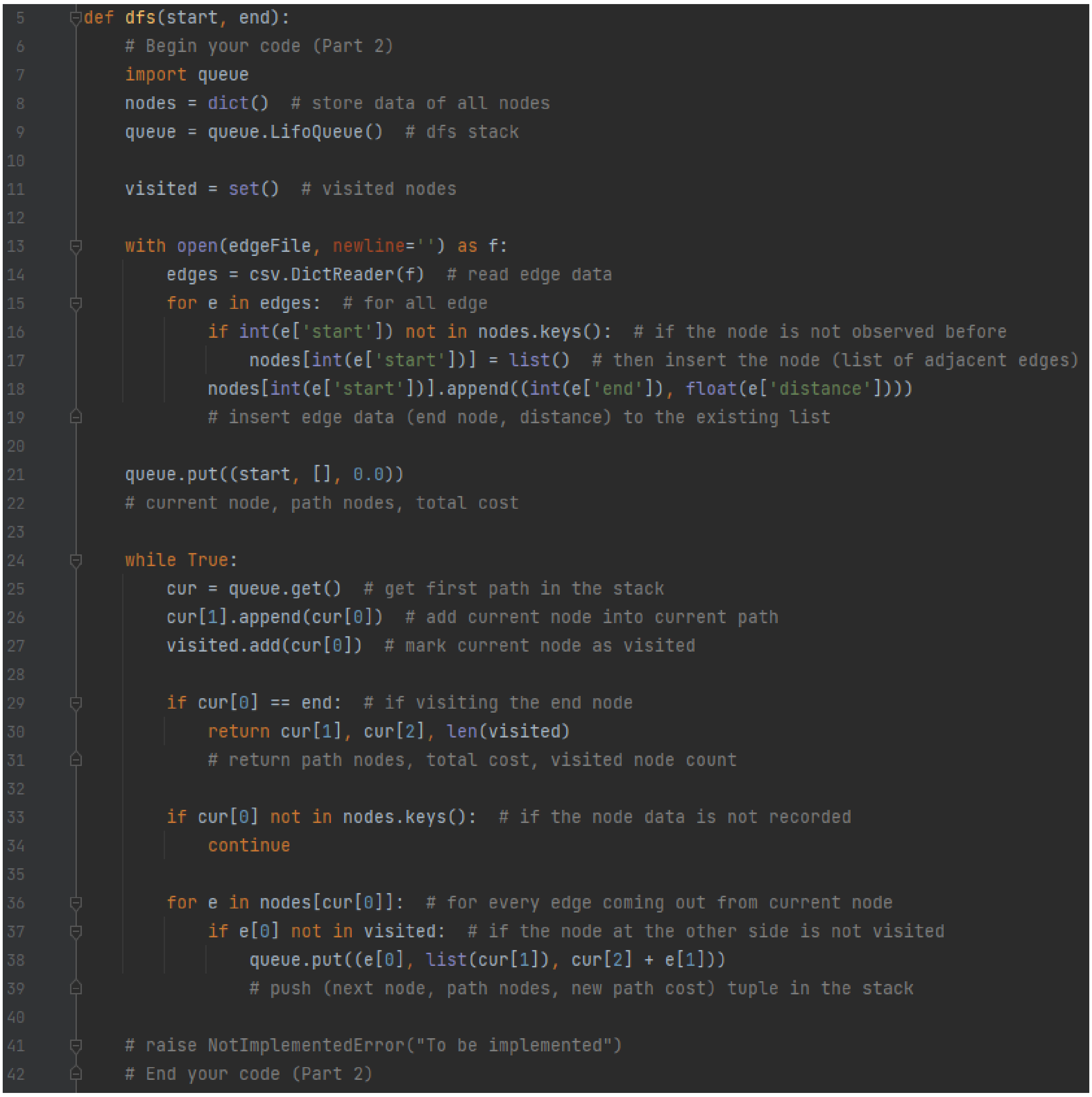
I. Implementation

- BFS



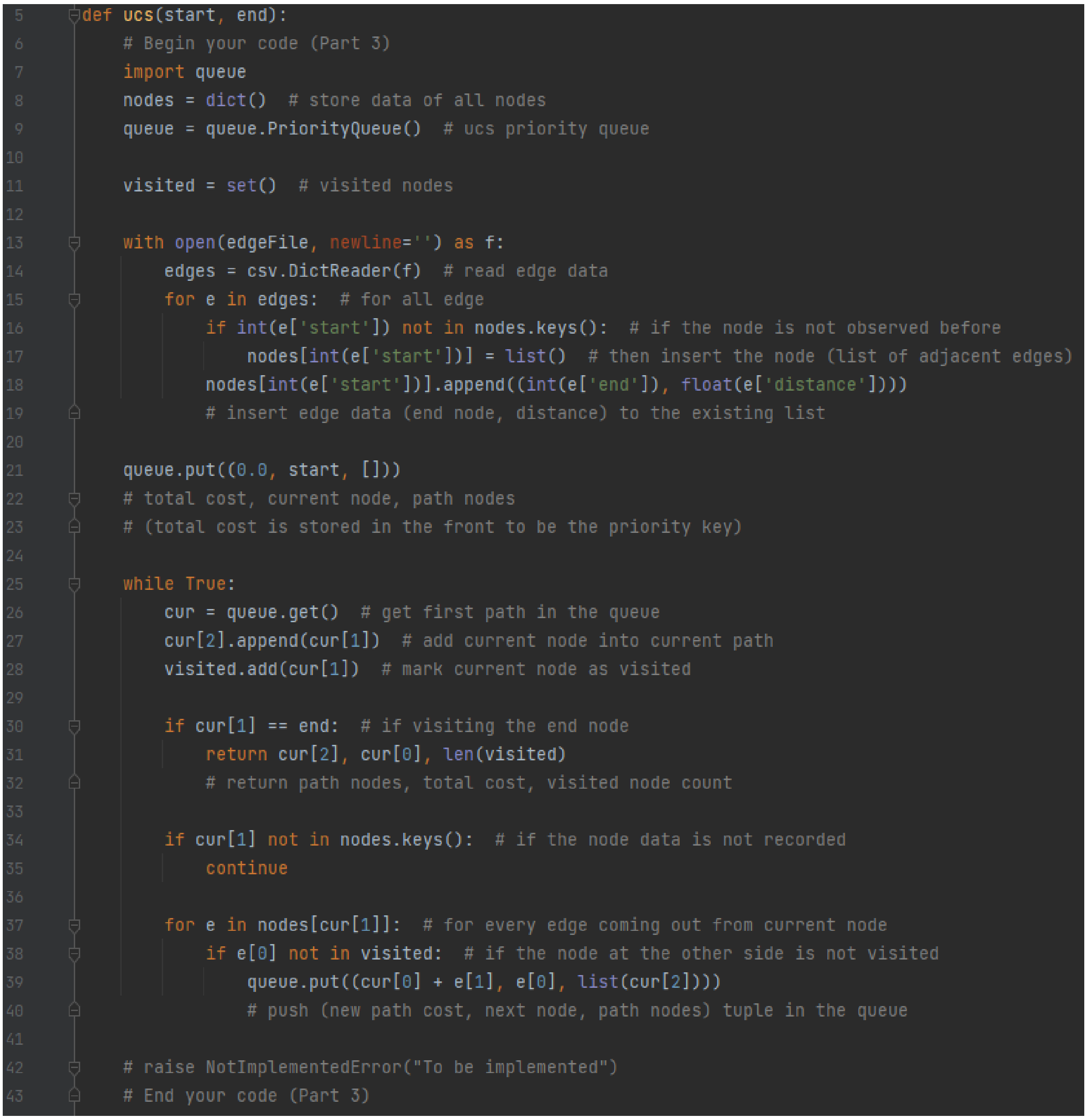
```python
def bfs(start, end):
    # Begin your code (Part 1)
    import queue
    nodes = dict()  # store data of all nodes
    queue = queue.Queue()  # bfs queue

    visited = set()  # visited nodes

    with open(edgeFile, newline='') as f:
        edges = csv.DictReader(f)  # read edge data
        for e in edges:  # for all edge
            if int(e['start']) not in nodes.keys():  # if the node is not observed before
                nodes[int(e['start'])] = list()  # then insert the node (list of adjacent edges)
            nodes[int(e['start'])].append((int(e['end']), float(e['distance'])))
            # insert edge data (end node, distance) to the existing list

    queue.put((start, [], 0.0))
    # current node, path nodes, total cost

    while True:
        cur = queue.get()  # get first path in the queue
        cur[1].append(cur[0])  # add current node into current path
        visited.add(cur[0])  # mark current node as visited

        if cur[0] == end:  # if visiting the end node
            return cur[1], cur[2], len(visited)
            # return path nodes, total cost, visited node count

        if cur[0] not in nodes.keys():  # if the node data is not recorded
            continue

        for e in nodes[cur[0]]:  # for every edge coming out from current node
            if e[0] not in visited:  # if the node at the other side is not visited
                queue.put((e[0], list(cur[1]), cur[2] + e[1]))
                # push (next node, path nodes, new path cost) tuple in the queue

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

- DFS (stack)



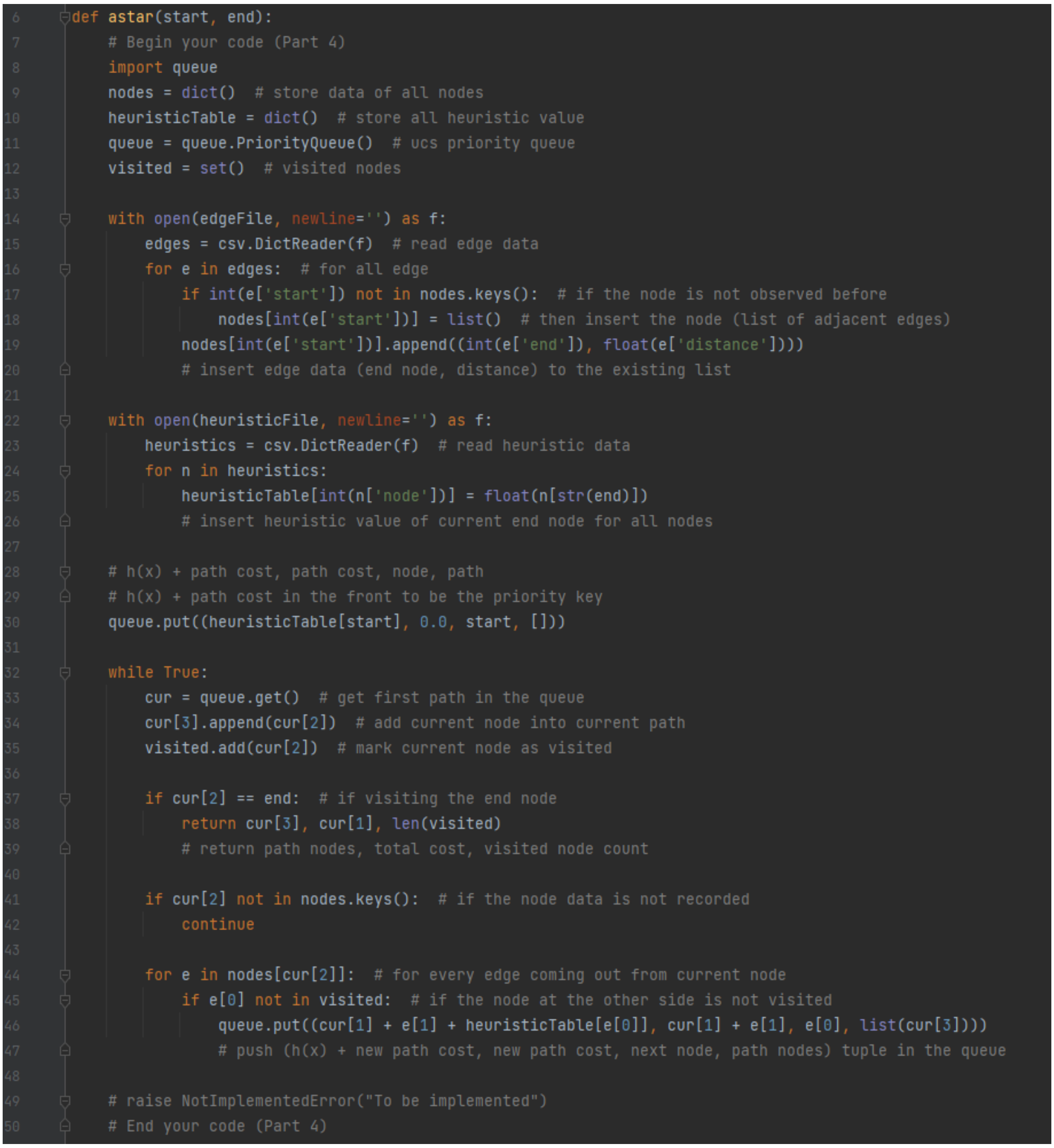
```python
def dfs(start, end):
    # Begin your code (Part 2)
    import queue
    nodes = dict()  # store data of all nodes
    queue = queue.LifoQueue()  # dfs stack

    visited = set()  # visited nodes

    with open(edgeFile, newline='') as f:
        edges = csv.DictReader(f)  # read edge data
        for e in edges:  # for all edge
            if int(e['start']) not in nodes.keys():  # if the node is not observed before
                nodes[int(e['start'])] = list()  # then insert the node (list of adjacent edges)
            nodes[int(e['start'])].append((int(e['end']), float(e['distance'])))
            # insert edge data (end node, distance) to the existing list

    queue.put((start, [], 0.0))
    # current node, path nodes, total cost

    while True:
        cur = queue.get()  # get first path in the stack
        cur[1].append(cur[0])  # add current node into current path
        visited.add(cur[0])  # mark current node as visited

        if cur[0] == end:  # if visiting the end node
            return cur[1], cur[2], len(visited)
            # return path nodes, total cost, visited node count

        if cur[0] not in nodes.keys():  # if the node data is not recorded
            continue

        for e in nodes[cur[0]]:  # for every edge coming out from current node
            if e[0] not in visited:  # if the node at the other side is not visited
                queue.put((e[0], list(cur[1]), cur[2] + e[1]))
                # push (next node, path nodes, new path cost) tuple in the stack

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

- UCS



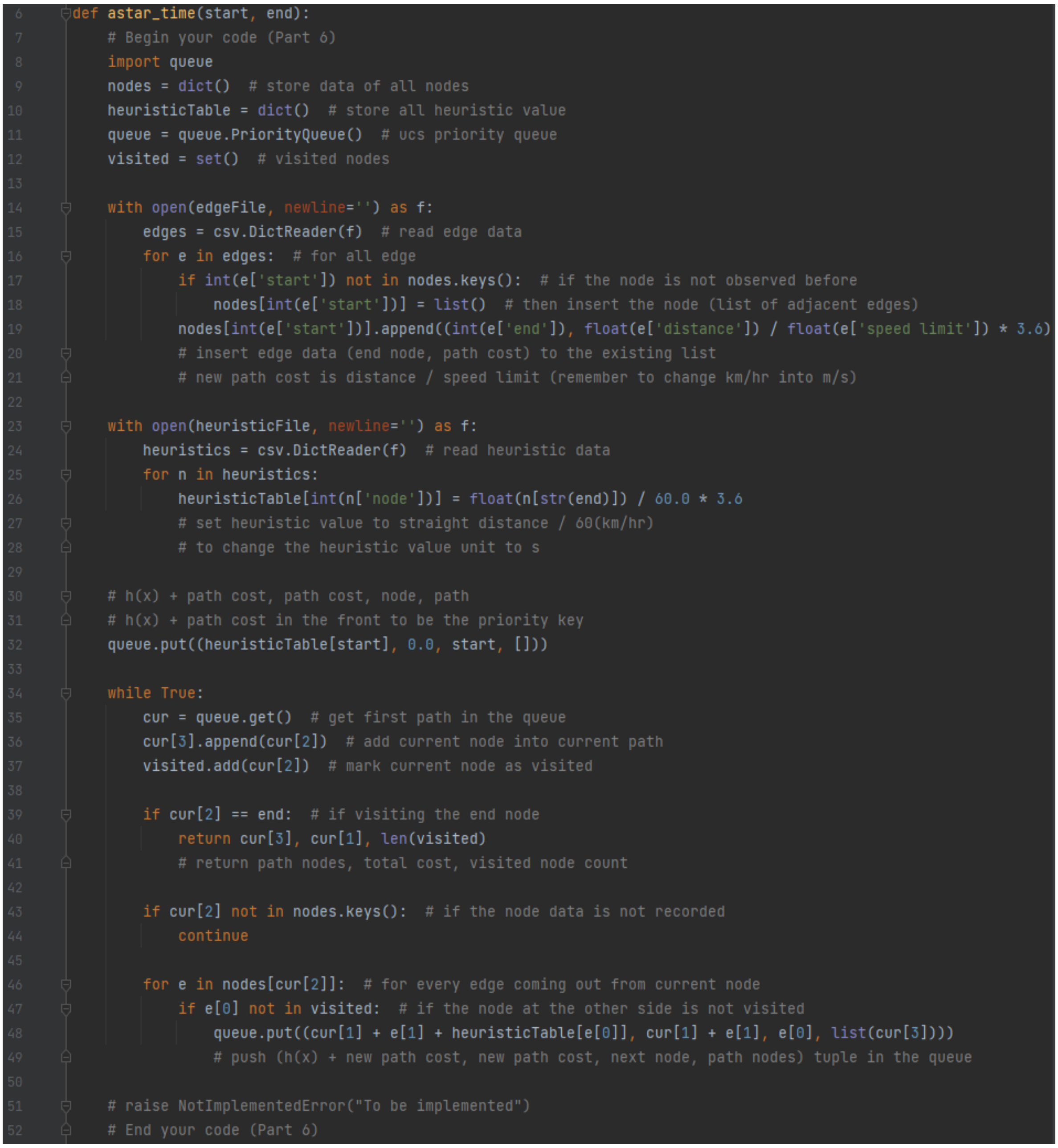
```python
def ucs(start, end):
    # Begin your code (Part 3)
    import queue
    nodes = dict()  # store data of all nodes
    queue = queue.PriorityQueue()  # ucs priority queue

    visited = set()  # visited nodes

    with open(edgeFile, newline='') as f:
        edges = csv.DictReader(f)  # read edge data
        for e in edges:  # for all edge
            if int(e['start']) not in nodes.keys():  # if the node is not observed before
                nodes[int(e['start'])] = list()  # then insert the node (list of adjacent edges)
            nodes[int(e['start'])].append((int(e['end']), float(e['distance'])))
            # insert edge data (end node, distance) to the existing list

    queue.put((0.0, start, []))
    # total cost, current node, path nodes
    # (total cost is stored in the front to be the priority key)

    while True:
        cur = queue.get()  # get first path in the queue
        cur[2].append(cur[1])  # add current node into current path
        visited.add(cur[1])  # mark current node as visited

        if cur[1] == end:  # if visiting the end node
            return cur[2], cur[0], len(visited)
            # return path nodes, total cost, visited node count

        if cur[1] not in nodes.keys():  # if the node data is not recorded
            continue

        for e in nodes[cur[1]]:  # for every edge coming out from current node
            if e[0] not in visited:  # if the node at the other side is not visited
                queue.put((cur[0] + e[1], e[0], list(cur[2])))
                # push (new path cost, next node, path nodes) tuple in the queue

    # raise NotImplementedError("To be implemented")
    # End your code (Part 3)
```

- A*



```
def astar(start, end):
    # Begin your code (Part 4)
    import queue
    nodes = dict()  # store data of all nodes
    heuristicTable = dict()  # store all heuristic value
    queue = queue.PriorityQueue()  # ucs priority queue
    visited = set()  # visited nodes

    with open(edgeFile, newline='') as f:
        edges = csv.DictReader(f)  # read edge data
        for e in edges:  # for all edge
            if int(e['start']) not in nodes.keys():  # if the node is not observed before
                nodes[int(e['start'])] = list()  # then insert the node (list of adjacent edges)
            nodes[int(e['start'])].append((int(e['end']), float(e['distance'])))
            # insert edge data (end node, distance) to the existing list

    with open(heuristicFile, newline='') as f:
        heuristics = csv.DictReader(f)  # read heuristic data
        for n in heuristics:
            heuristicTable[int(n['node'])] = float(n[str(end)])
            # insert heuristic value of current end node for all nodes

    # h(x) + path cost, path cost, node, path
    # h(x) + path cost in the front to be the priority key
    queue.put((heuristicTable[start], 0.0, start, []))

    while True:
        cur = queue.get()  # get first path in the queue
        cur[3].append(cur[2])  # add current node into current path
        visited.add(cur[2])  # mark current node as visited

        if cur[2] == end:  # if visiting the end node
            return cur[3], cur[1], len(visited)
            # return path nodes, total cost, visited node count

        if cur[2] not in nodes.keys():  # if the node data is not recorded
            continue

        for e in nodes[cur[2]]:  # for every edge coming out from current node
            if e[0] not in visited:  # if the node at the other side is not visited
                queue.put((cur[1] + e[1] + heuristicTable[e[0]], cur[1] + e[1], e[0], list(cur[3])))
                # push (h(x) + new path cost, new path cost, next node, path nodes) tuple in the queue

    # raise NotImplementedError("To be implemented")
    # End your code (Part 4)
```

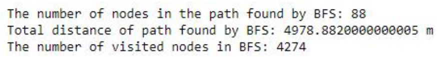
- A* (time): I set the new heuristic value to be the original one / 60(km/hr), because 60 km/h is the highest speed limit of all roads, so that it can transform the Euclidean distance heuristic to estimated arrive time heuristic.



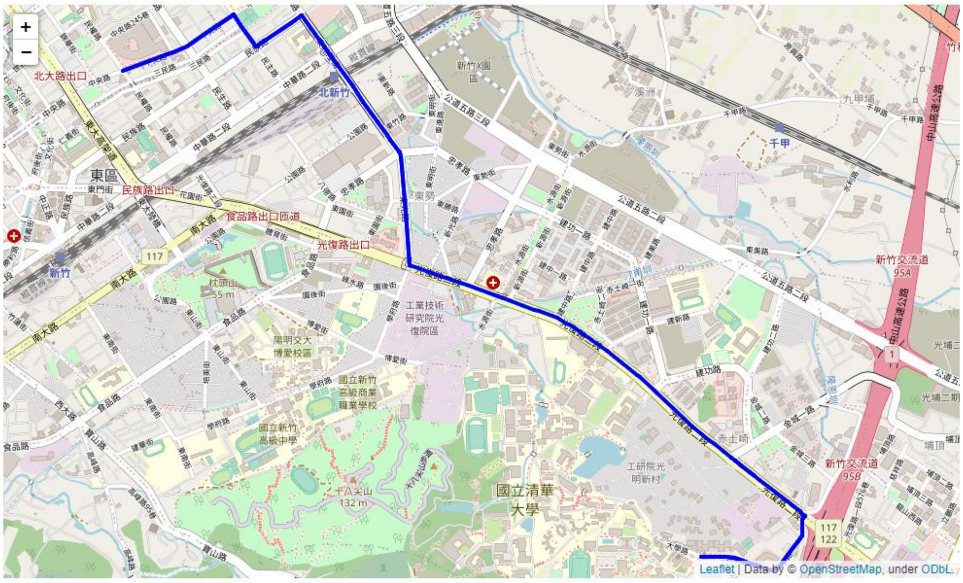
```python
def astar_time(start, end):
    # Begin your code (Part 6)
    import queue
    nodes = dict()  # store data of all nodes
    heuristicTable = dict()  # store all heuristic value
    queue = queue.PriorityQueue()  # ucs priority queue
    visited = set()  # visited nodes

    with open(edgeFile, newline='') as f:
        edges = csv.DictReader(f)  # read edge data
        for e in edges:  # for all edge
            if int(e['start']) not in nodes.keys():  # if the node is not observed before
                nodes[int(e['start'])] = list()  # then insert the node (list of adjacent edges)
            nodes[int(e['start'])].append((int(e['end']), float(e['distance']) / float(e['speed limit']) * 3.6)
            # insert edge data (end node, path cost) to the existing list
            # new path cost is distance / speed limit (remember to change km/hr into m/s)

    with open(heuristicFile, newline='') as f:
        heuristics = csv.DictReader(f)  # read heuristic data
        for n in heuristics:
            heuristicTable[int(n['node'])] = float(n[str(end)]) / 60.0 * 3.6
            # set heuristic value to straight distance / 60(km/hr)
            # to change the heuristic value unit to s

    # h(x) + path cost, path cost, node, path
    # h(x) + path cost in the front to be the priority key
    queue.put((heuristicTable[start], 0.0, start, []))

    while True:
        cur = queue.get()  # get first path in the queue
        cur[3].append(cur[2])  # add current node into current path
        visited.add(cur[2])  # mark current node as visited

        if cur[2] == end:  # if visiting the end node
            return cur[3], cur[1], len(visited)
            # return path nodes, total cost, visited node count

        if cur[2] not in nodes.keys():  # if the node data is not recorded
            continue

        for e in nodes[cur[2]]:  # for every edge coming out from current node
            if e[0] not in visited:  # if the node at the other side is not visited
                queue.put((cur[1] + e[1] + heuristicTable[e[0]], cur[1] + e[1], e[0], list(cur[3])))
                # push (h(x) + new path cost, new path cost, next node, path nodes) tuple in the queue

    # raise NotImplementedError("To be implemented")
    # End your code (Part 6)
```

II. Result & Analysis

- Test 1: from National Yang Ming Chiao Tung University (ID: 2270143902) to Big City Shopping Mall (ID: 1079387396)
    - BFS:

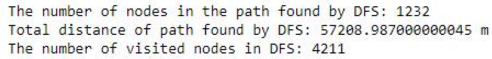


```
The number of nodes in the path found by BFS: 88
Total distance of path found by BFS: 4978.8820000000005 m
The number of visited nodes in BFS: 4274
```

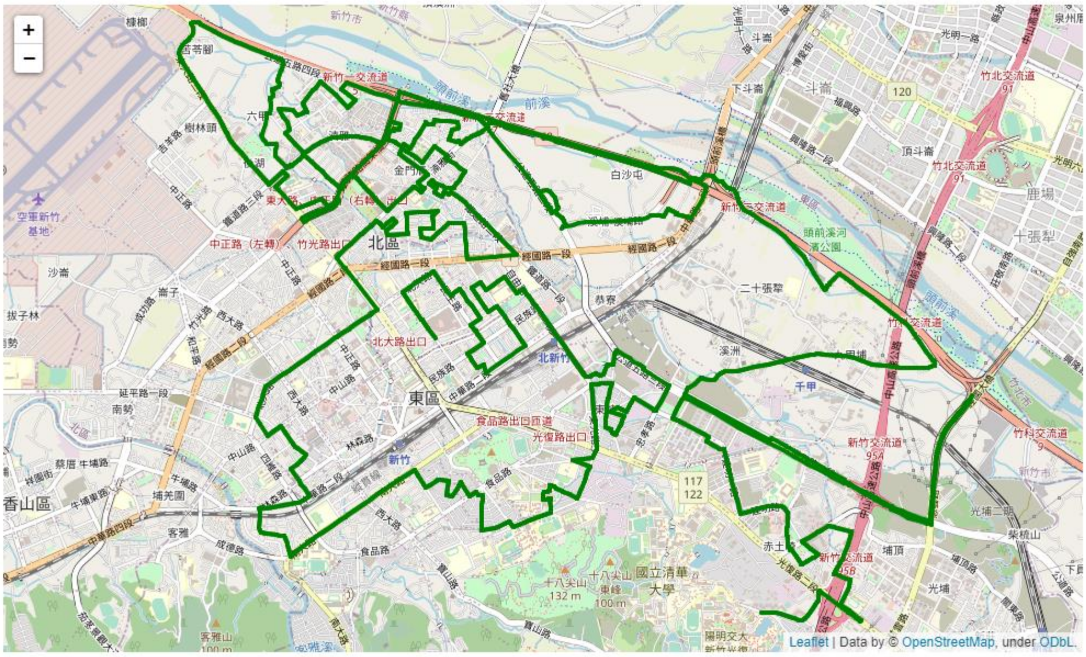


Leaflet | Data by © OpenStreetMap, under ODbL.

- DFS (stack):

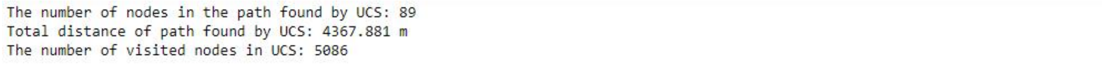


```
The number of nodes in the path found by DFS: 1232
Total distance of path found by DFS: 57208.987000000045 m
The number of visited nodes in DFS: 4211
```

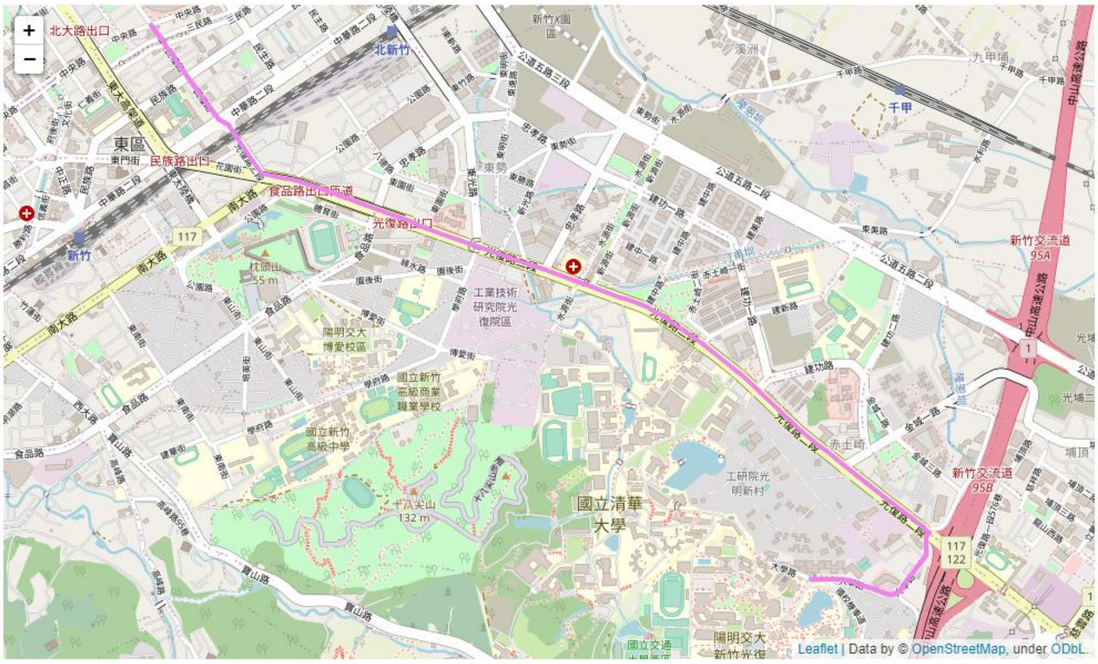


Leaflet | Data by © OpenStreetMap, under ODbL.

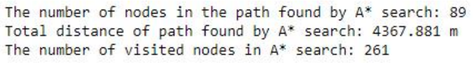
- UCS:



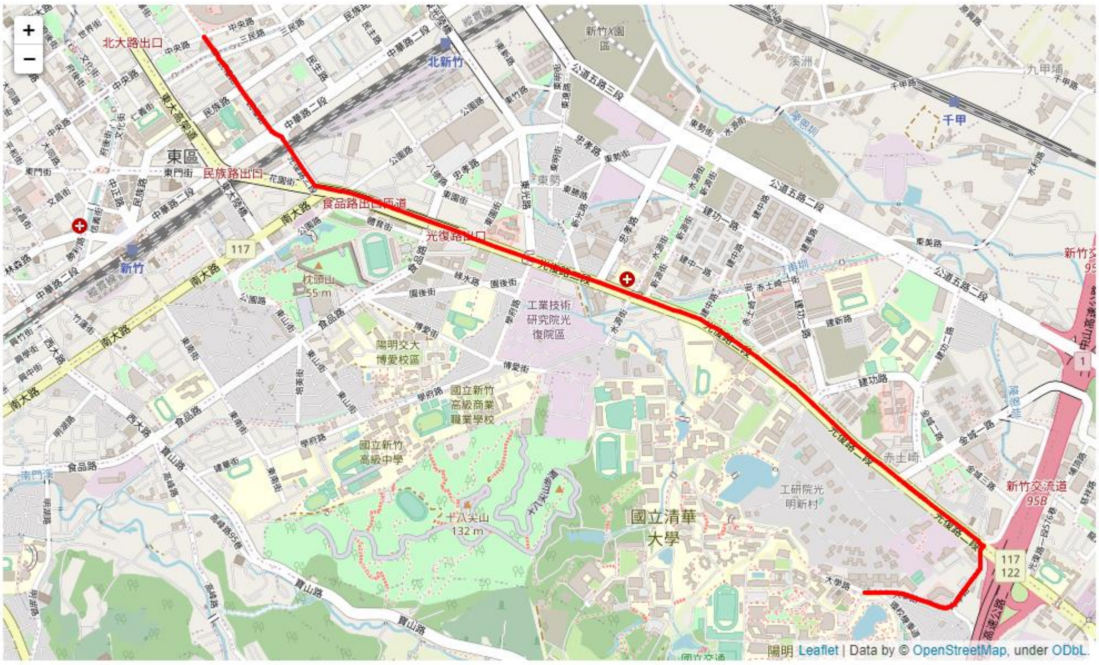
```
The number of nodes in the path found by UCS: 89
Total distance of path found by UCS: 4367.881 m
The number of visited nodes in UCS: 5086
```



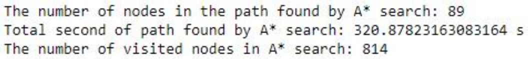
- A*:



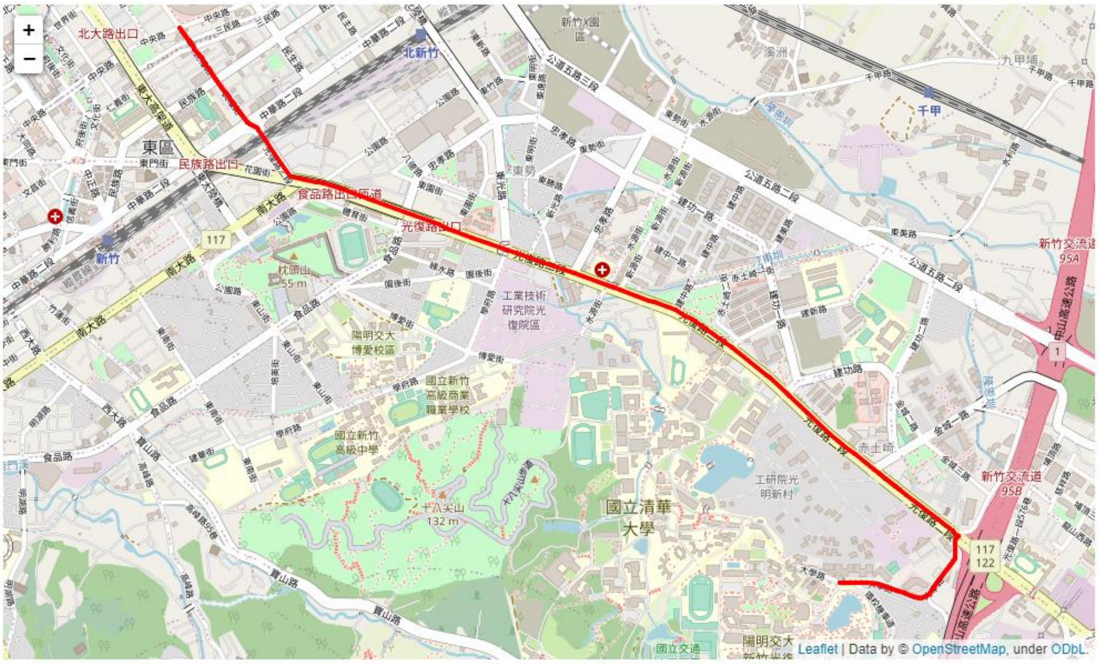
```
The number of nodes in the path found by A* search: 89
Total distance of path found by A* search: 4367.881 m
The number of visited nodes in A* search: 261
```



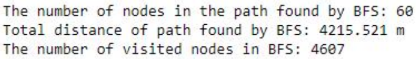
- A* (time):



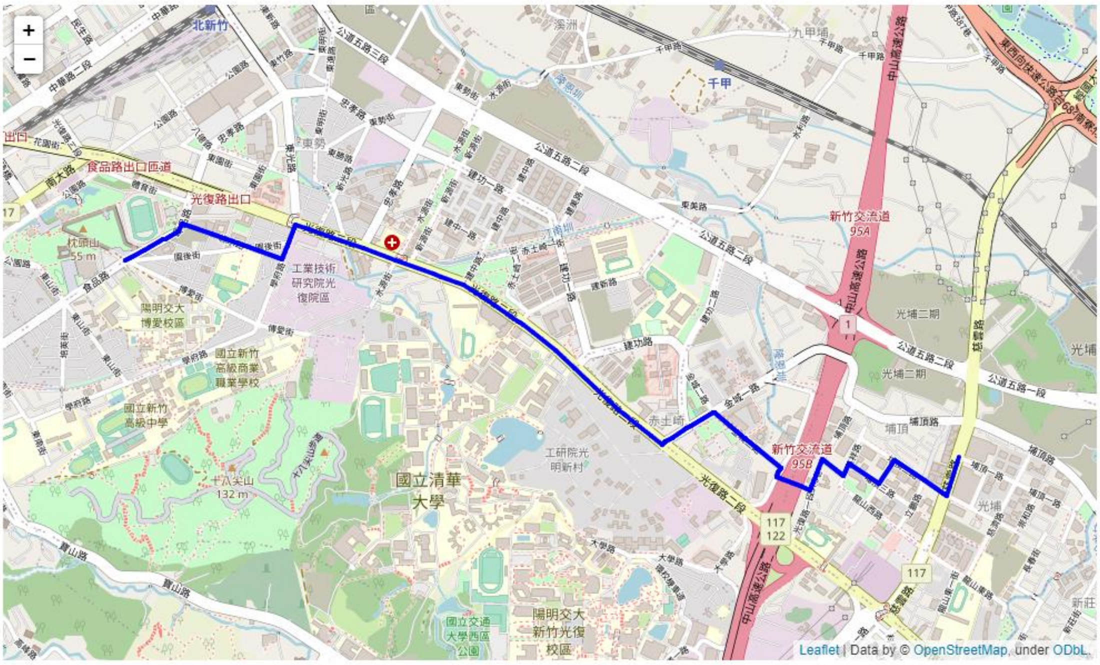
```
The number of nodes in the path found by A* search: 89
Total second of path found by A* search: 320.87823163083164 s
The number of visited nodes in A* search: 814
```



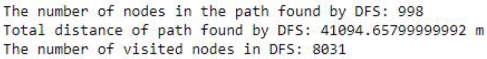
- Test 2: from Hsinchu Zoo (ID: 426882161) to COSTCO Hsinchu Store (ID: 1737223506)
  - BFS:



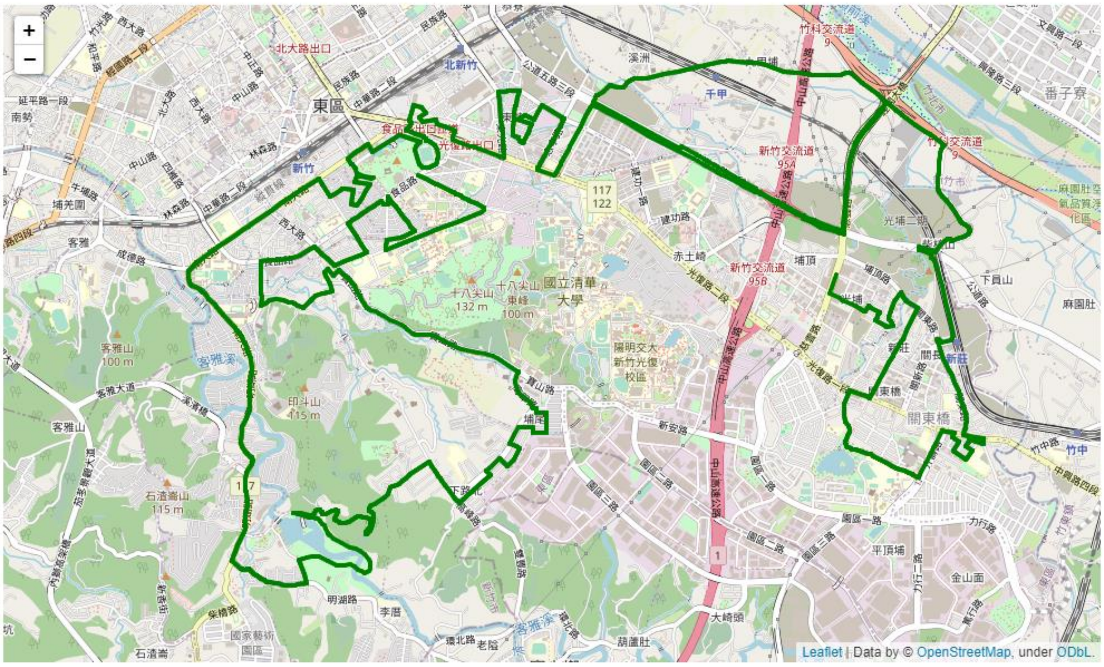
```
The number of nodes in the path found by BFS: 60
Total distance of path found by BFS: 4215.521 m
The number of visited nodes in BFS: 4607
```



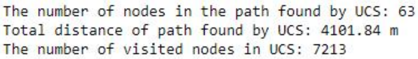
  - DFS (stack):



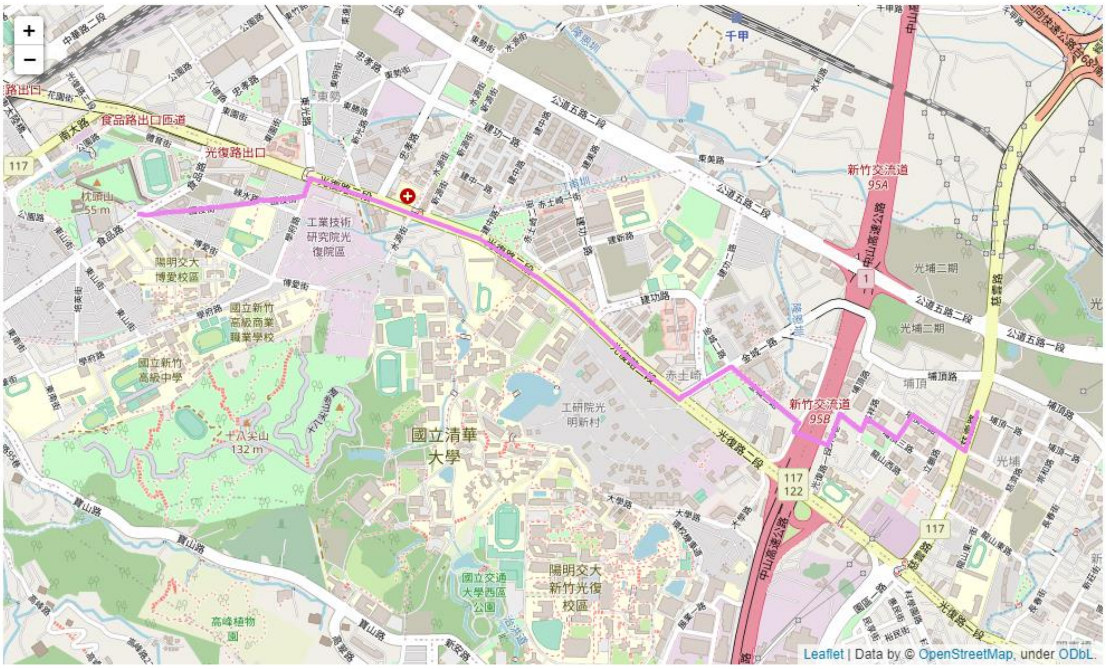
```
The number of nodes in the path found by DFS: 998
Total distance of path found by DFS: 41094.65799999992 m
The number of visited nodes in DFS: 8031
```



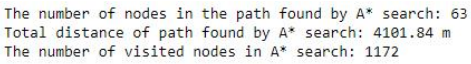
- UCS:



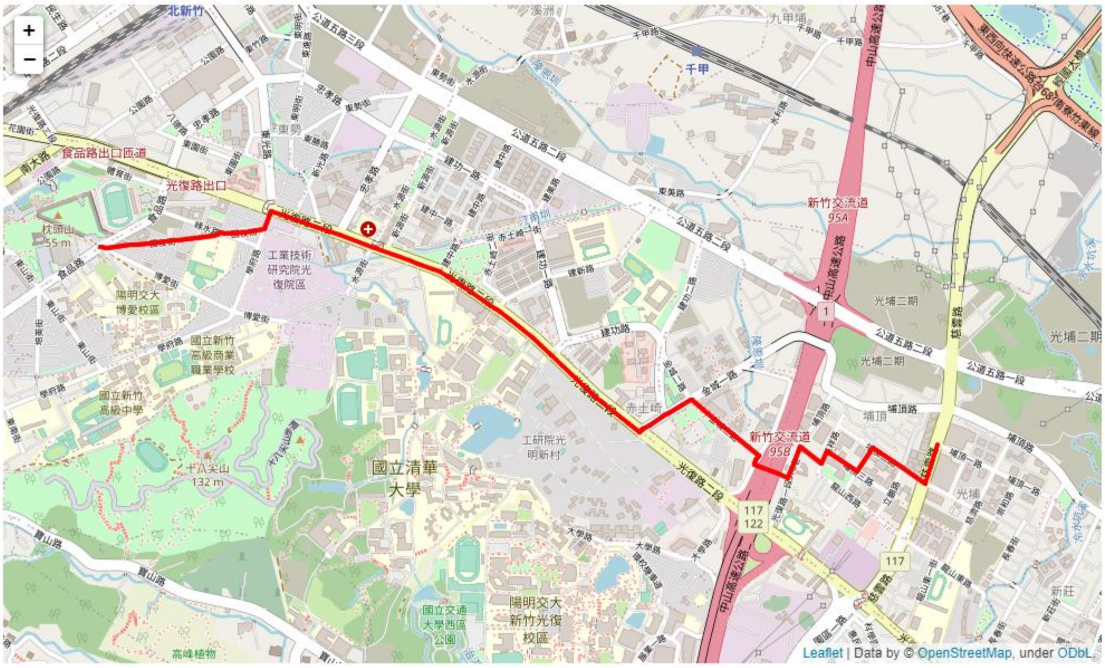
```
The number of nodes in the path found by UCS: 63
Total distance of path found by UCS: 4101.84 m
The number of visited nodes in UCS: 7213
```



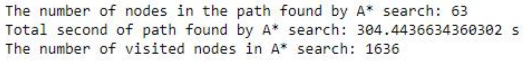
- A*:



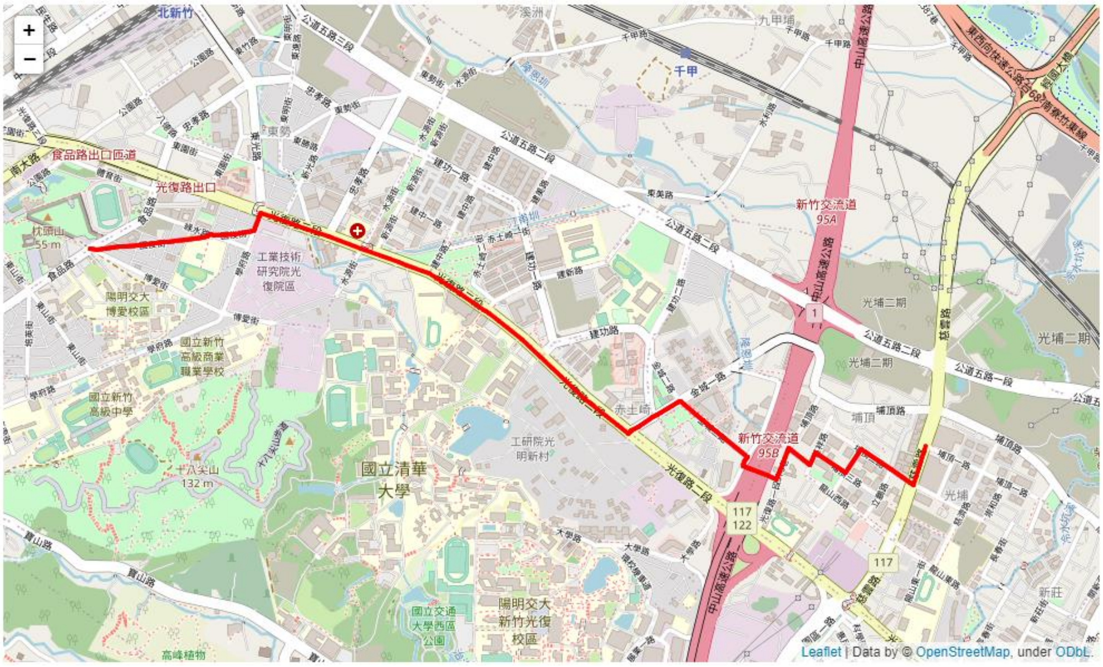
```
The number of nodes in the path found by A* search: 63
Total distance of path found by A* search: 4101.84 m
The number of visited nodes in A* search: 1172
```



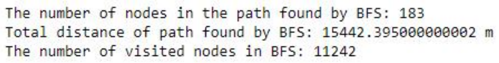
- A* (time):



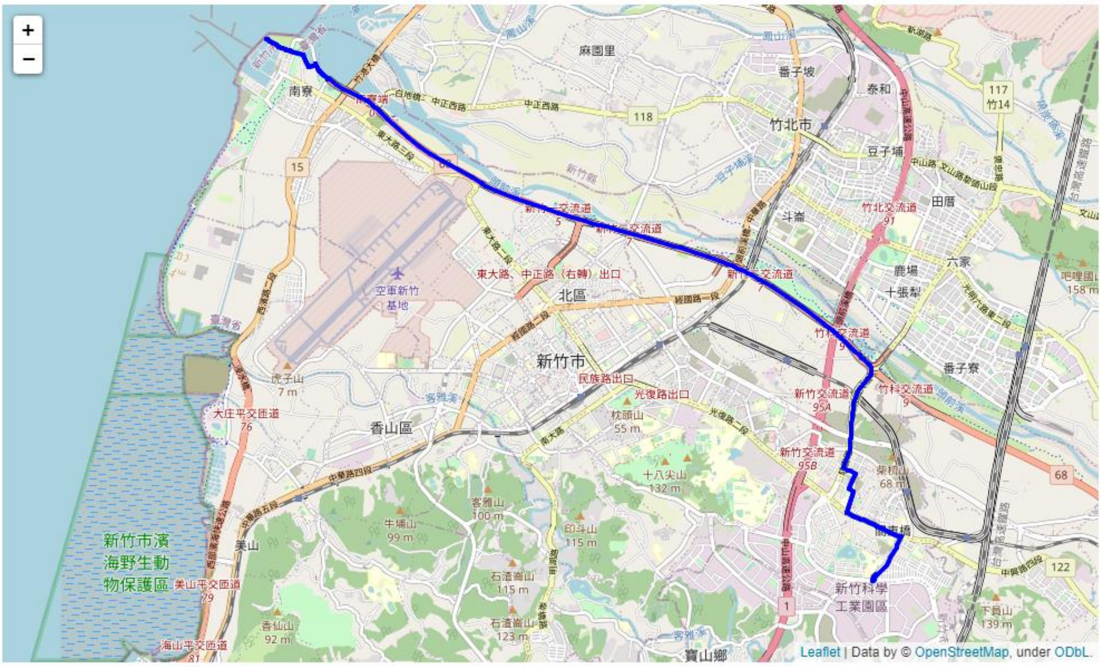
```
The number of nodes in the path found by A* search: 63
Total second of path found by A* search: 304.4436634360302 s
The number of visited nodes in A* search: 1636
```



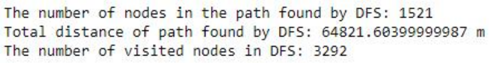
- Test 3: from National Experimental High School At Hsinchu Science Park (ID: 1718165260) to Nanliao Fishing Port (ID: 8513026827)
  - BFS:



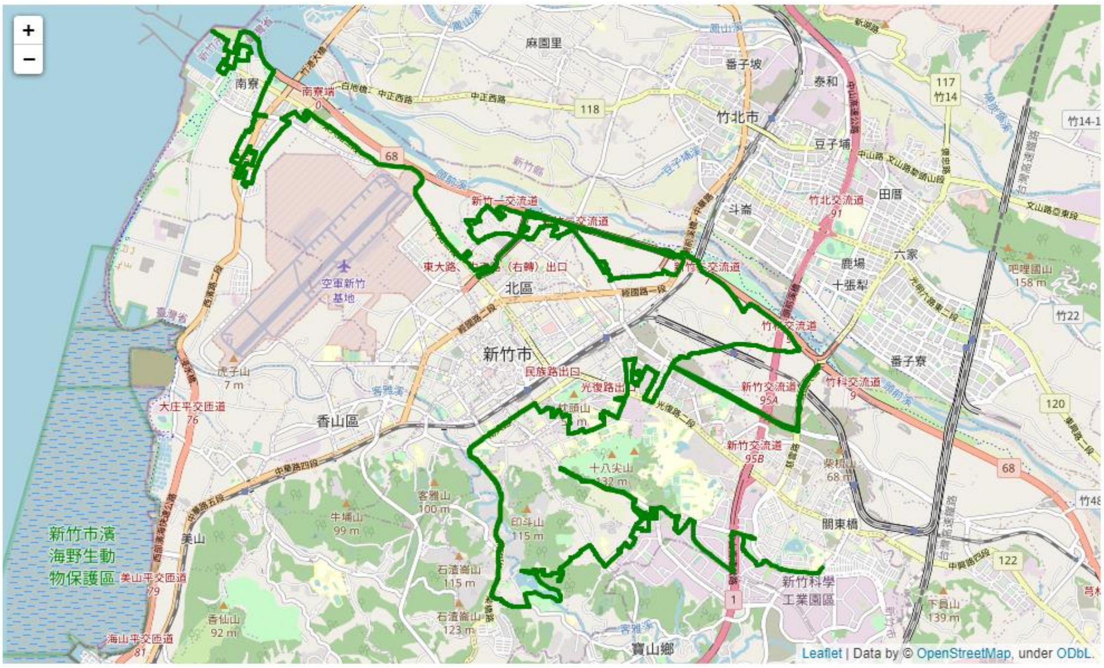
```
The number of nodes in the path found by BFS: 183
Total distance of path found by BFS: 15442.395000000002 m
The number of visited nodes in BFS: 11242
```



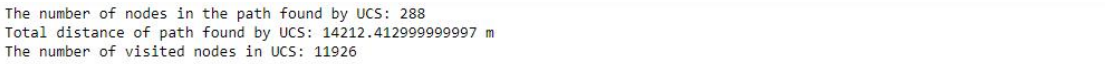
- DFS (stack):



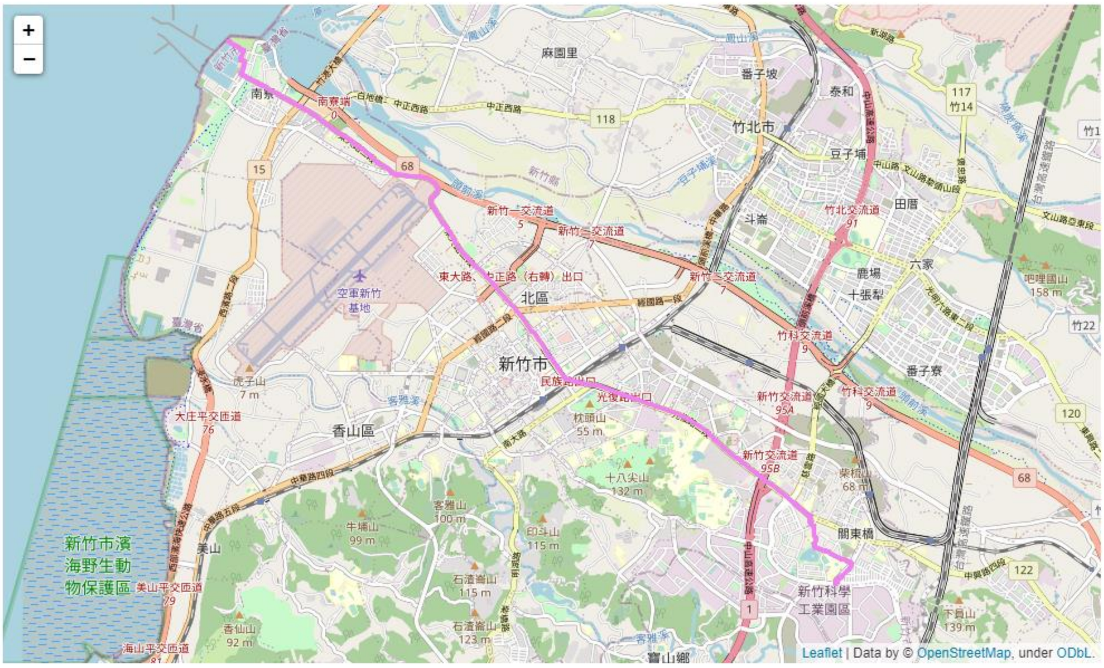
```
The number of nodes in the path found by DFS: 1521
Total distance of path found by DFS: 64821.60399999987 m
The number of visited nodes in DFS: 3292
```



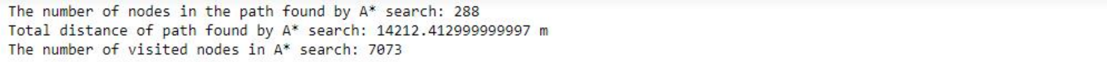
- UCS:



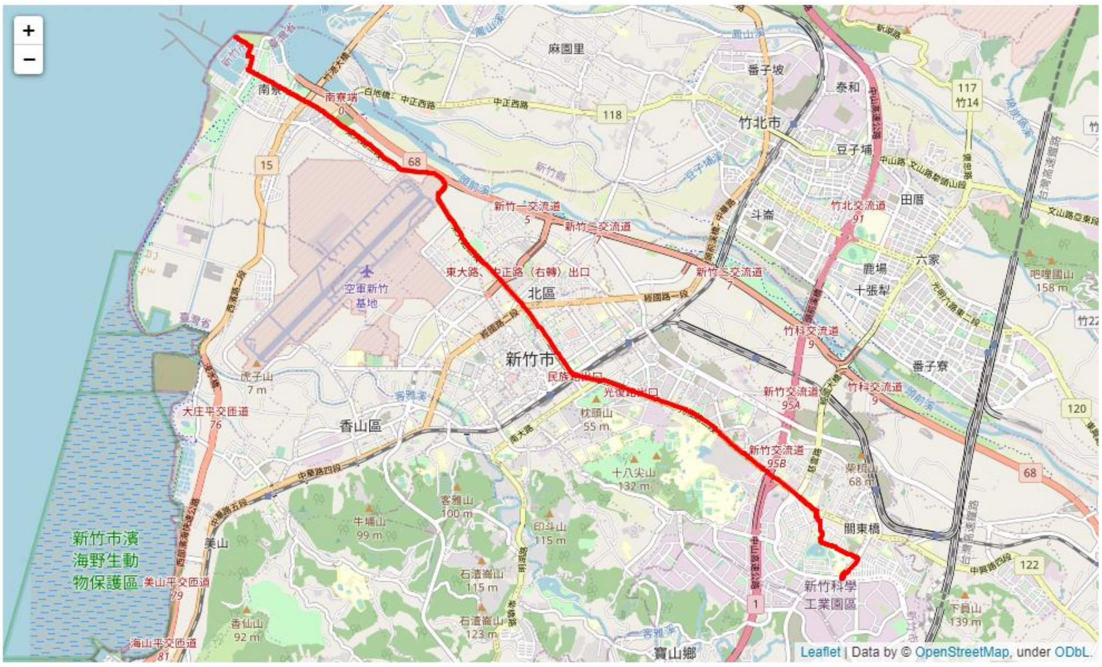
```
The number of nodes in the path found by UCS: 288
Total distance of path found by UCS: 14212.412999999997 m
The number of visited nodes in UCS: 11926
```



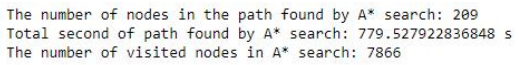
- A*:



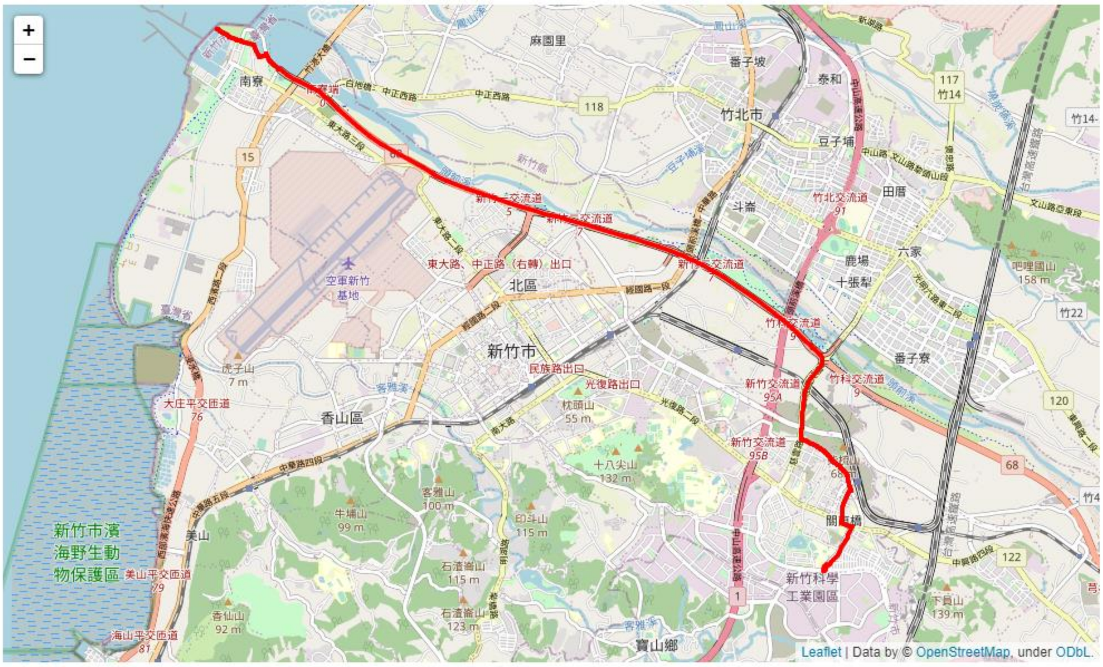
```
The number of nodes in the path found by A* search: 288
Total distance of path found by A* search: 14212.412999999997 m
The number of visited nodes in A* search: 7073
```



- A* (time):



```
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: 7866
```



- Analysis
  - BFS can always find the path that has lowest path node count, but can't ensure that one is the shortest path.
  - DFS is the dumbest search algorithm in shortest path, because it always goes through all possible path, don't care about path cost or direction. Thus, the result path often makes a big loop to reach the end node, and the path cost is extremely high.
  - UCS can ensure the result is shortest, but it tends to visit too much nodes to generate a shortest path, causes long running time.
  - A* is like UCS, always choose the most potential path, but it added the concept of heuristic value to make sure the path is getting closer to end node. Thus, heuristic value makes A* visit less nodes than UCS, makes it faster.
  - In A* (time), I changed the distance heuristic value into time heuristic value to make sure the heuristic value is admissible, and other details are same as the original A*.

III. Question Answering

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

   At first when I was working on A*(time), I directly used the original heuristic value, but I found the result time is too high (1109 seconds for the third case). After thinking some methods to improve the algorithm, I changed the heuristic value into time unit, the calculation seems more reasonable, and the result time also decreased to 779 seconds.

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.

   Beside speed limit and distance, I think time is also an important attribute. In the morning or at the afternoon, there may be lots of people going to work or going home, causing high traffic load and affects the overall speed of some roads. In this case, lanes might have more potential while they have more distance than main streets.

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

   A. Mapping: maybe can use an aerial camera to record coordinate and recognize the buildings, streets, and lanes. Then key in the street names or building names manually after that.

   B. Localization: because the mapping section has already recorded the coordinate of every streets and buildings at the same time, I think using GPS to locate a user's position might be feasible.

4. The estimated time of arrival 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.

   - ETA = (remaining path length / average speed of the deliver) + (estimated traffic light waiting time).

   A. Because driving speed varies a lot for different time or different driver, average speed can reflect the attributes mentioned above.

   B. Traffic lights are annoying when we are in a hurry, especially the long-waiting-time traffic lights, so we should consider the waste of time in ETA. I think a reasonable way to calculate waiting is to sum up all traffic lights waiting time on the remaining path and multiply the percentage of encountering a red light (say 50%). That is, estimated traffic light waiting time = (∑ waiting time of traffic lights on remaining path) * 50%