Homework 2: Route Finding

Report Template

110550012 黃鵬軒

Part I. Implementation (6%):

Part1

```
    # Begin your code (Part 1)
    """
    First, read edges.csv and construct an adjacency list representation of a graph,
    where each edge has a distance. It then performs BFS using queue and keeps track of the visited nodes.
    If the end node is found, returns path, distance, and number of nodes visited.
    """
    adjList = {}
    with open(edgeFile, newline='') as csvFile:
        rows = csv.reader(csvFile)
        next(rows)  # skip first row(header)
        for row in rows:
            startNode = int(row[0])
            endNode = int(row[1])
            distance = float(row[2])
            if startNode not in adjList:
                adjList[startNode] = []
            adjList[startNode].append((endNode, distance))

    q = Queue()
    visited = set()
    q.put((start, [start], 0))

    while not q.empty():
        curNode, path, curDistance = q.get()
        visited.add(curNode)
        if curNode == end:
            return path, curDistance, len(visited)
        for adj, dist in adjList.get(curNode, []):
            if adj not in visited:
                q.put((adj, path+[adj], curDistance+dist))

    return path, dist, len(visited)
    # path: list of node IDs. first start, last end.
    # End your code (Part 1)
```

Part2

```
    # Begin your code (Part 2)
    """
    First, read edges.csv and construct an adjacency list representation of a graph,
    where each edge has a distance.Then, perform DFS using stack to keep track of nodes to be
    visited next. Initialize the stack with a tuple containing startNode, a list with the startNode, and a distance of 0.
    As long as the stack is not empty, it pops element from the stack and adds its node to the set of visited nodes.
    If the node is endNode, returns the path, distance, and num_visited.
    Otherwise, it loops through the adjacent nodes of the current node in the adjacency list,
    adding them to the path and pushing them onto the stack if they have not been visited yet.
    """
    adjList = {}
    with open(edgeFile, newline='') as csvFile:
        rows = csv.reader(csvFile)
        next(rows)
        for row in rows:
            startNode = int(row[0])
            endNode = int(row[1])
            distance = float(row[2])
            if startNode not in adjList:
                adjList[startNode] = []
            adjList[startNode].append((endNode, distance))

    stack = [(start, [start], 0)]
    visited = set()
    while stack:
        curNode, path, curDistance = stack.pop()
        visited.add(curNode)
        if curNode == end:
            return path, curDistance, len(visited)
        for adj, dist in adjList.get(curNode, []):
            if adj not in visited:
                stack.append((adj, path+[adj], curDistance+dist))

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

Part3

```python
    # Begin your code (Part 3)
    """
    First, read edges.csv and construct an adjacency list representation of a graph,
    where each edge has a distance. It then performs UCS using priority queue and keeps track of the visited nodes.
    initializes pq with a tuple current distance from the start node (0), starNode, and a list containing startNode.
    explores the graph by repeatedly selecting connected edge with the lowest cost that have not been visited.
    If the end node is found, returns path, distance, and number of nodes visited.
    """
    adjList = {}
    with open(edgeFile, newline='') as csvFile:
        rows = csv.reader(csvFile)
        next(rows)  # skip first row(header)
        for row in rows:
            startNode = int(row[0])
            endNode = int(row[1])
            distance = float(row[2])
            if startNode not in adjList:
                adjList[startNode] = []
            adjList[startNode].append((endNode, distance))

    pq = PriorityQueue()
    visited = set()
    pq.put((0, start, [start])) # Maintaining order based on distance

    while not pq.empty():
        curDistance, curNode, path = pq.get()
        visited.add(curNode)
        if curNode == end:
            return path, curDistance, len(visited)
        for adj, dist in adjList.get(curNode, []):
            if adj not in visited:
                pq.put((curDistance+dist, adj, path+[adj]))

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

Part4

```python
    # Begin your code (Part 4)
    """
    store edge information in adjList and store heuristic in dictionary,
    where each node is a key and its value is heuristic value for reaching endNode.
    Initialize priority queue with startNode, distance from the start (0),
    its heuristic estimate, and path (startNode).
    Iteratively retrieve the node with the lowest estimated cost from the priority queue and
    expand it by adding its adjacent nodes to pq with distance to the adjacent node + the heuristic value for reaching endNode.
    Continue until the endNode is reached, and return.
    """
    adjList = {}
    with open(edgeFile, newline='') as csvFile:
        rows = csv.reader(csvFile)
        next(rows)  # skip first row(header)
        for row in rows:
            startNode = int(row[0])
            endNode = int(row[1])
            distance = float(row[2])
            if startNode not in adjList:
                adjList[startNode] = []
            adjList[startNode].append((endNode, distance))

    heuristic = {}
    with open(heuristicFile, newline='') as csvFile:
        rows = csv.reader(csvFile)
        header = next(rows)  # skip first row(header)
        endIndex = header.index(str(end))
        for row in rows:
            node = int(row[0])
            heuristic[node] = float(row[endIndex])
    pq = PriorityQueue()
    visited = set()
    h = heuristic[start]
    pq.put((h, 0, start, [start]))

    while not pq.empty():
        _, curDistance, curNode, path = pq.get()
        visited.add(curNode)
        if curNode == end:
            return path, curDistance, len(visited)
        for adj, dist in adjList.get(curNode, []):
            if adj not in visited:
                h = heuristic[adj]
                pq.put((curDistance+dist+h, curDistance+dist, adj, path+[adj]))

    return path, curDistance, len(visited)
    # End your code (Part 4)
```

Part6(bonus)

```python
    # Begin your code (Part 6)
    """
    Basically same as astar.py, the main difference is the fastest time to travel an edge is calculated
    and stored instead of distance, import pandas to get the maximum speed limit,
    and the heuristic value is evaluted by dividing it by the maximum speed limit, that is estimated time.
    """
    adjList = {}
    with open(edgeFile, newline='') as csvFile:
        rows = csv.reader(csvFile)
        next(rows) # skip first row(header)
        for row in rows:
            startNode = int(row[0])
            endNode = int(row[1])
            distance = float(row[2])
            speedLim = float(row[3])
            time = distance / (speedLim * 1000 / 3600)  # calculate the fastest time to travel the edge
            if startNode not in adjList:
                adjList[startNode] = []
            adjList[startNode].append((endNode, time))  # store the time instead of the distance

    df = pd.read_csv(edgeFile)
    MAX = float(df['speed limit'].max())
    MAX = MAX * 1000 / 3600
    heuristic = {}
    with open(heuristicFile, newline='') as csvFile:
        rows = csv.reader(csvFile)
        header = next(rows)  # skip first row(header)
        endIndex = header.index(str(end))
        for row in rows:
            node = int(row[0])
            heuristic[node] = float(row[endIndex]) / MAX

    pq = PriorityQueue()
    visited = set()
    h = heuristic[start]
    pq.put((h, 0, start, [start]))
    while not pq.empty():
        h, t, curNode, path = pq.get()
        visited.add(curNode)
        if curNode == end:
            return path, t, len(visited)
        for adj, edgeTime in adjList.get(curNode, []):
            if adj not in visited:
                h = heuristic[adj]
                pq.put((t + edgeTime + h, t + edgeTime, adj, path + [adj]))
    # time :float, the time of path
    # End your code (Part 6)
```

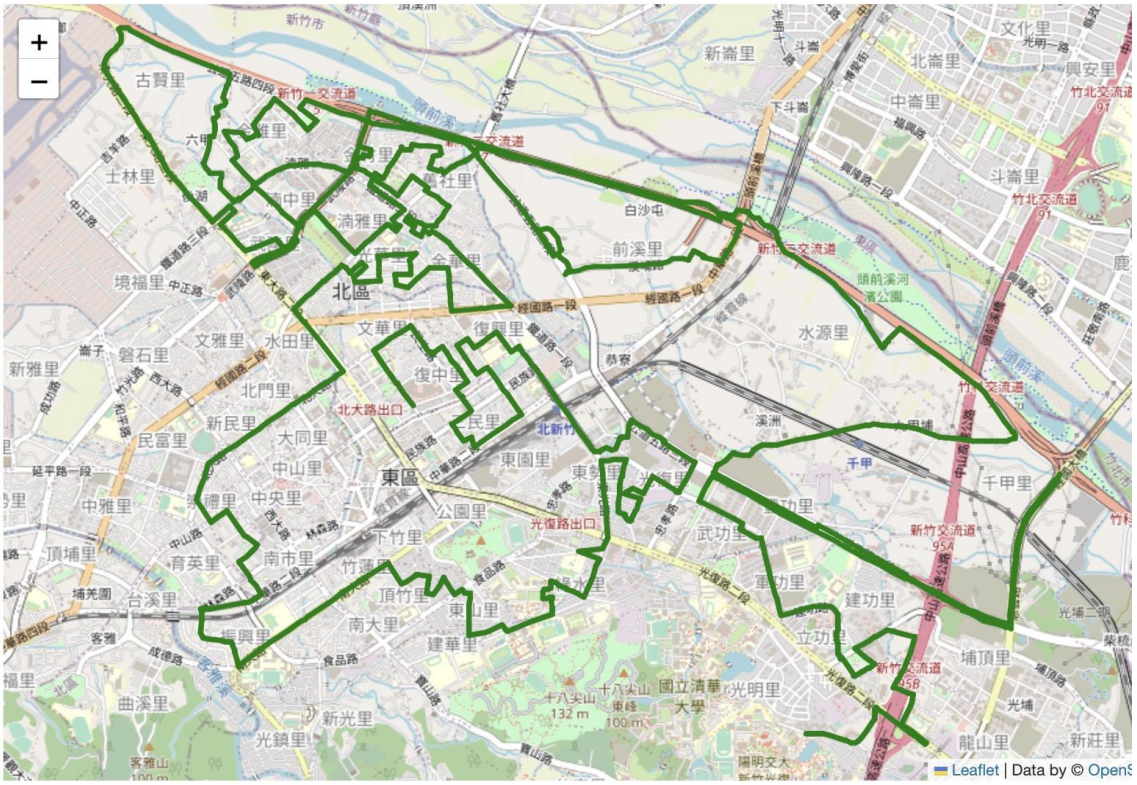
Part II. Results & Analysis (12%):

Test1: from National Yang Ming Chiao Tung University (ID: 2270143902) to Big City Shopping Mall (ID: 1079387396)
Part1 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
```

Part2 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
```



Part3 UCS:

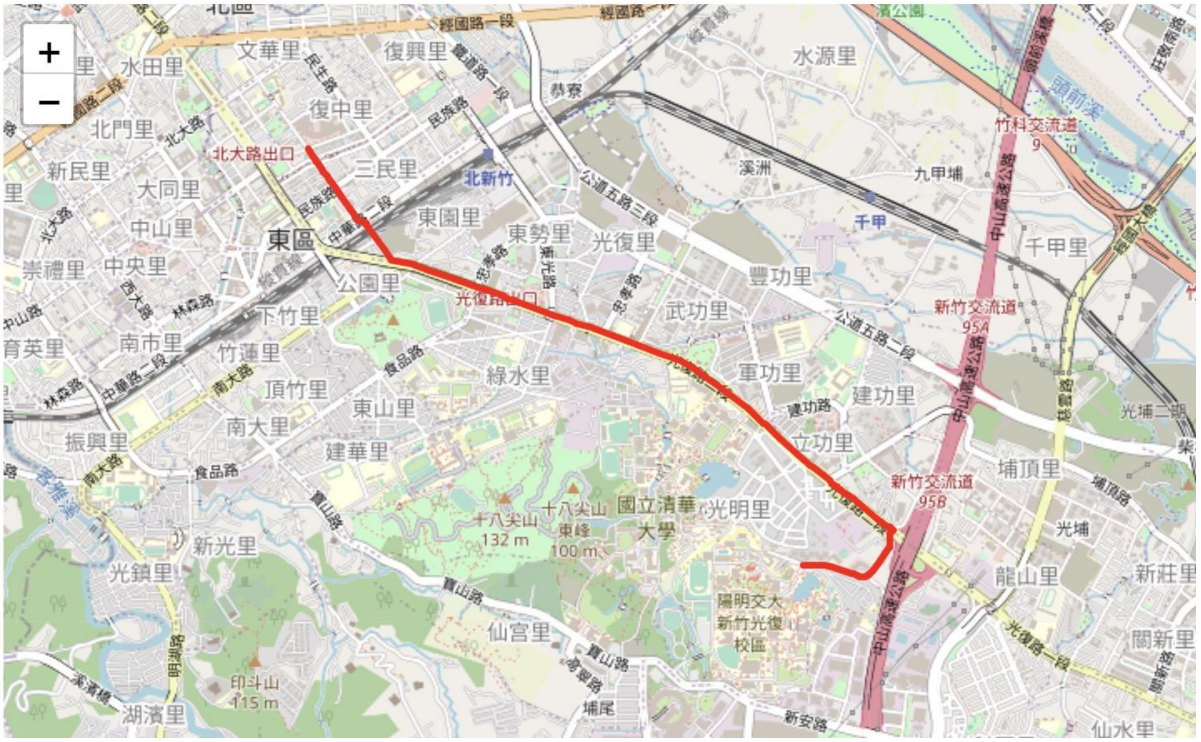
```
The number of nodes in the path found by UCS: 89
Total distance of path found by UCS: 4367.881 m
The number of visited nodes in UCS: 5086
```

Part4 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
```

Part6 A*(time) (bonus):

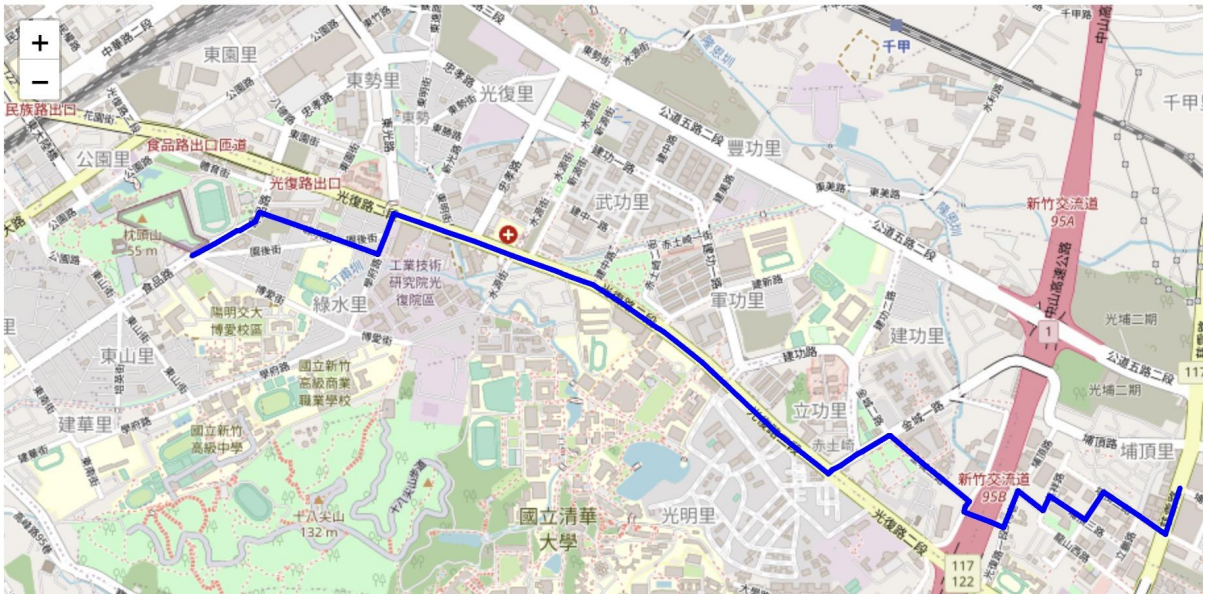
```
The number of nodes in the path found by A* search: 89
Total second of path found by A* search: 320.87823163083164 s
The number of visited nodes in A* search: 1934
```



Test2: from 426882161 to 1737223506

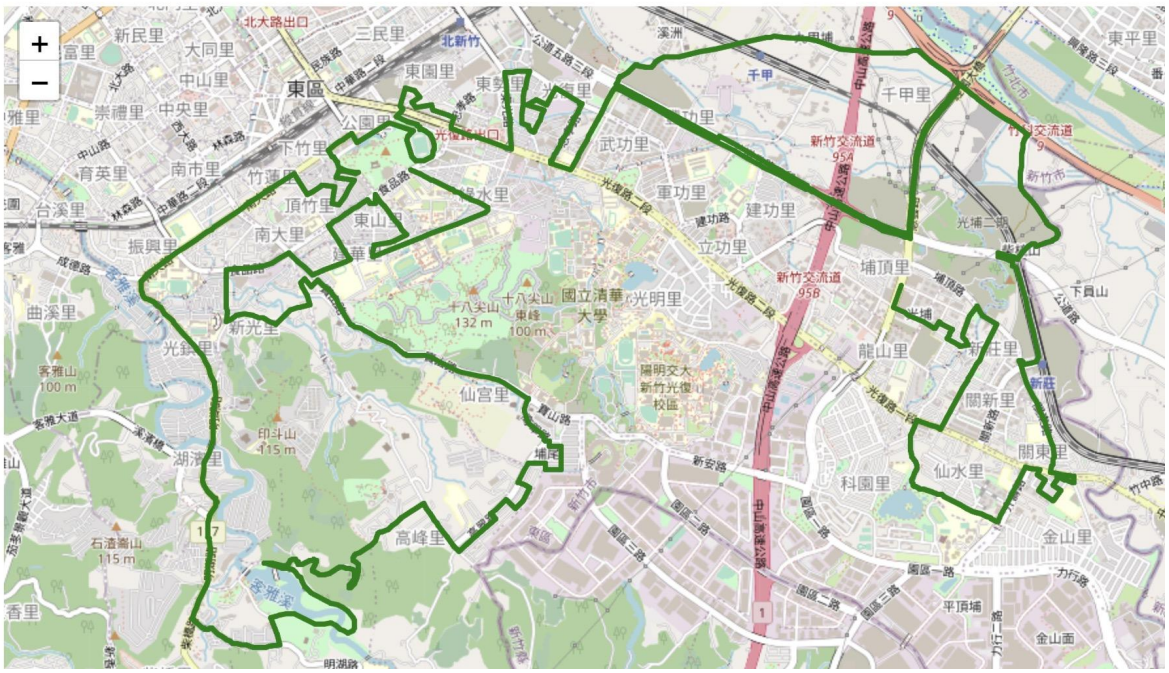
Part1 BFS:

```
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
```



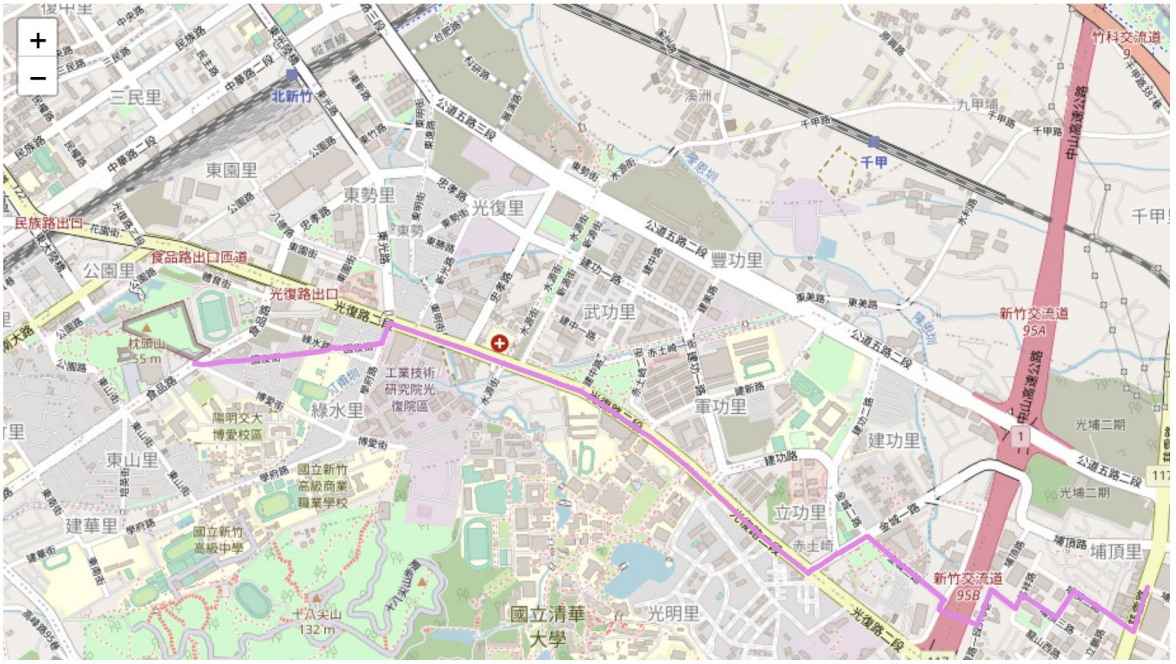
Part2 DFS:

```
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
```



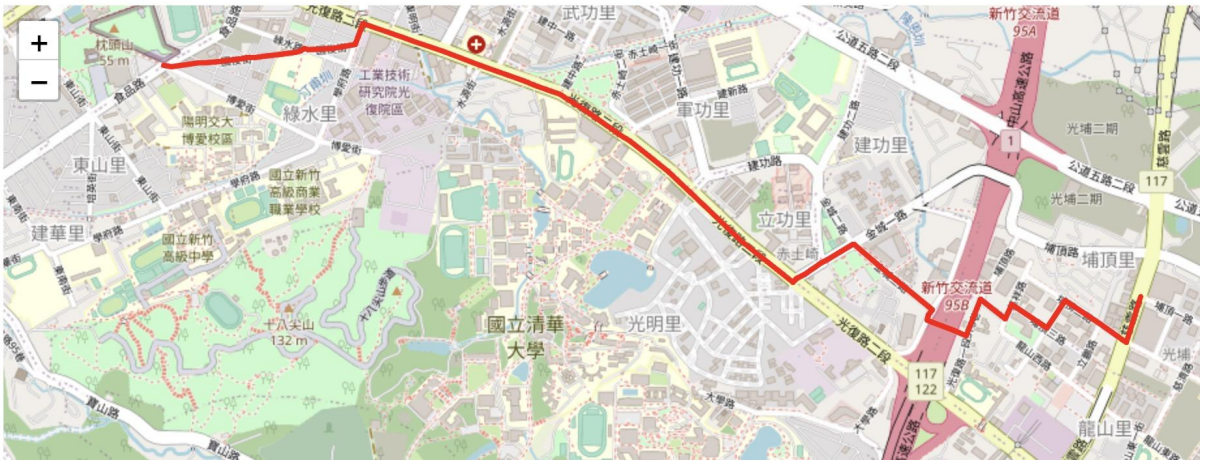
Part3 UCS:

```
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
```



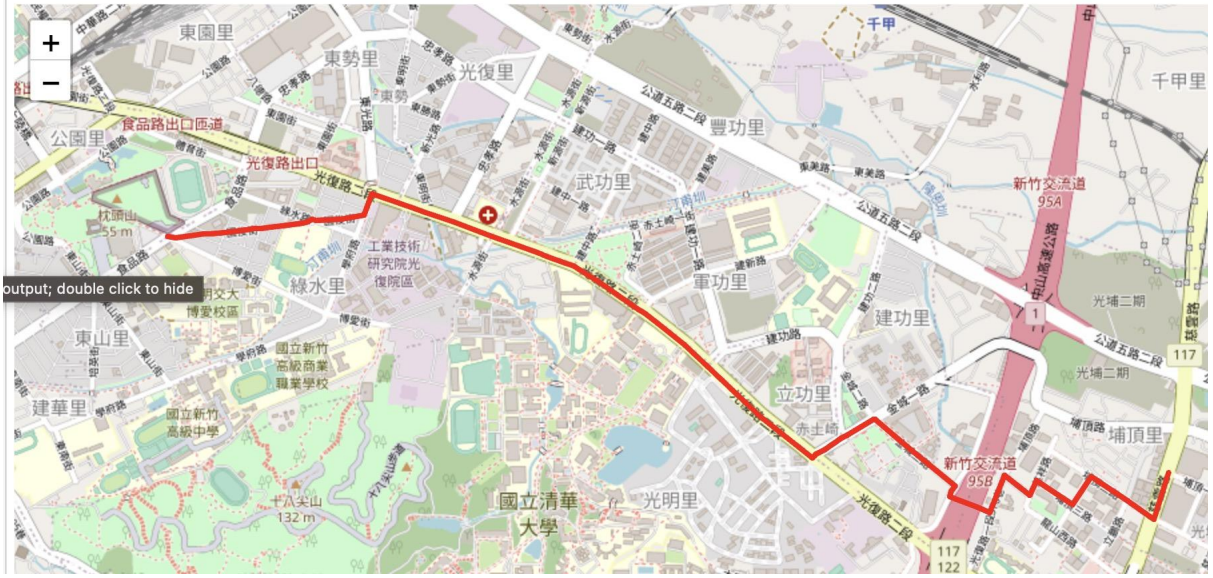
Part4 A*:

```
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
```



Part6 A*(time):

```
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: 2870
```



Test3: from 1718165260 to 8513026827

Part1 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
```

Part2 DFS:

```
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
```

Part3 UCS:

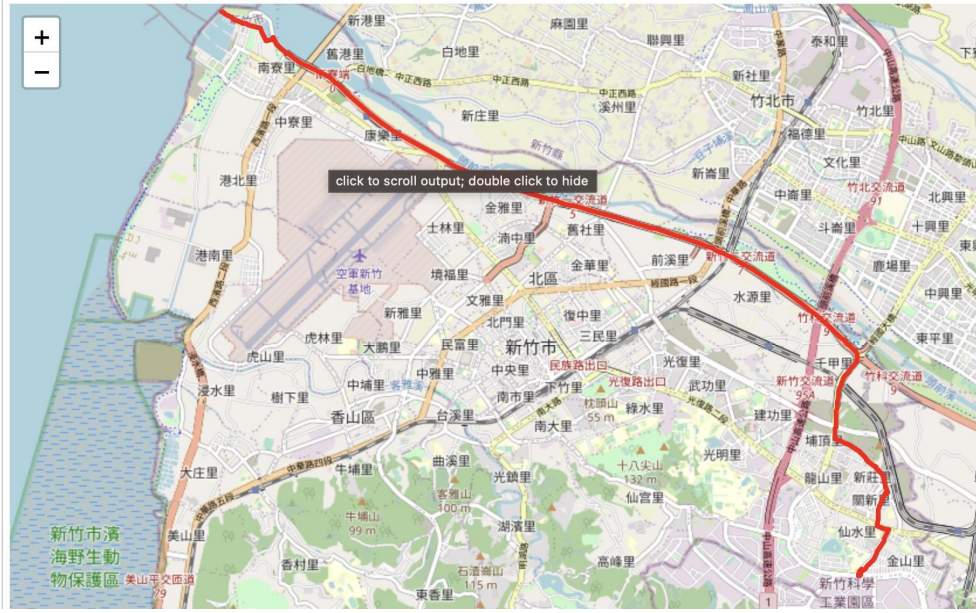
```
The number of nodes in the path found by UCS: 288
Total distance of path found by UCS: 14212.412999999997 m
The number of visited nodes in UCS: 11926
```

Part4 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
```

Part6 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: 8458
```



Analysis:

From the above results, it is apparent that DFS is the worst method for route finding as the path found by DFS is extremely far from the shortest path. The results of other methods such as BFS, UCS, and A* are not significantly different. The results of UCS and A* are the same and slightly better than BFS. A*(time) has no significant difference with A*, but I think the way of representing travel time is more practical.

Part III. Question Answering (12%):

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

   In the beginning, I had difficulty understanding the concept of a heuristic function, so I spent a lot of time searching on Google to implement astar.py. And in Part 6, I had no idea how to design a heuristic function, but eventually I simply converted the heuristic value to travel time by dividing the maximum speed limit, and it seemed to work. This part was interesting to me.

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.**

   I think the presence of traffic signals is an important attribute. Traffic signals can significantly impact travel time, especially during rush hour. Maybe we can consider the amount of signals of the path in the algorithm to optimize travel 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?**

   mapping: Use lidar detection. Lidar sensors can scan the surrounding environment and create a 3D map, which can be used for localization and route planning.

   localization: We can use GPS. GPS receivers can estimate the location of a vehicle by using signals from GPS satellites.

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.**

   We can split the ETA into two cases: ETA when the order is placed and ETA after the delivery driver picks up the food. For the first case, the function would be meal prep time + estimated arrival time of the delivery driver to the restaurant + possible additional time (such as delivery priority or multiple orders that can be estimated using past data) + time from pickup to delivery (initial value for the second case). The time from pickup to delivery can be estimated using the algorithm used in Part 6, with the heuristic function modified to (estimate distance to the target) / (median of speed limit), which is more likely to provide accurate estimates. The time can then be dynamically updated based on the driver's location and adjusted for traffic congestion.