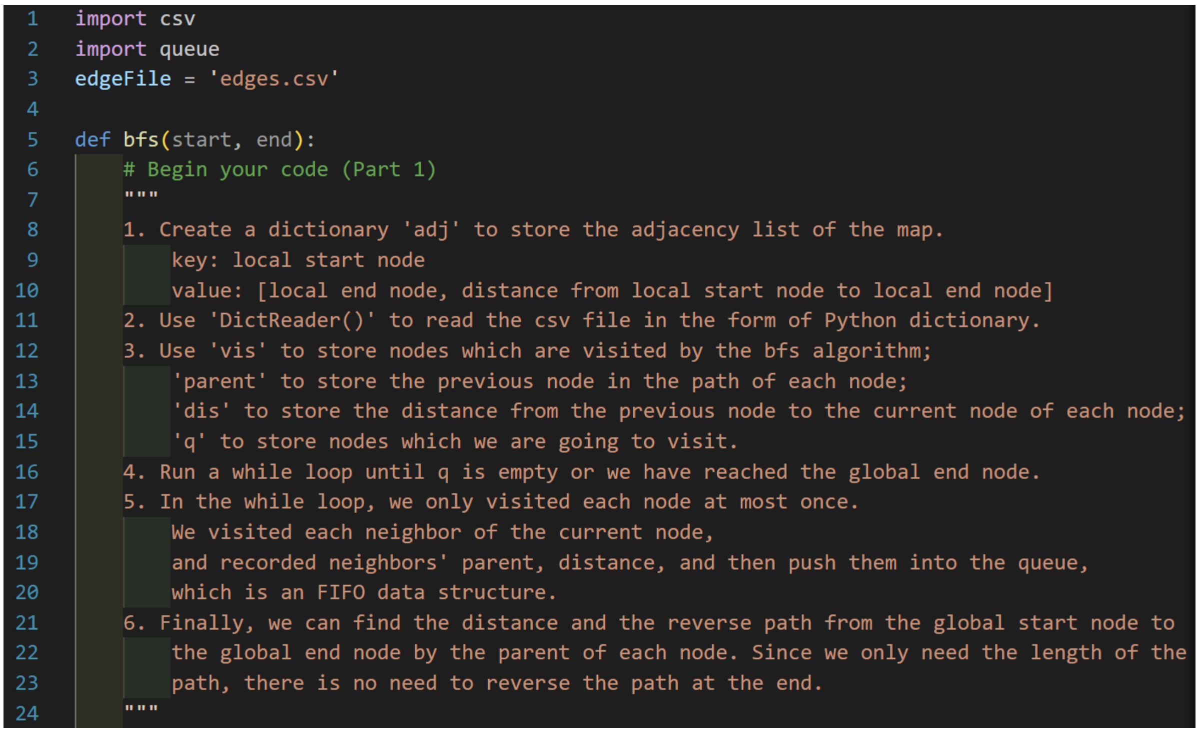
Homework 2: Route Finding

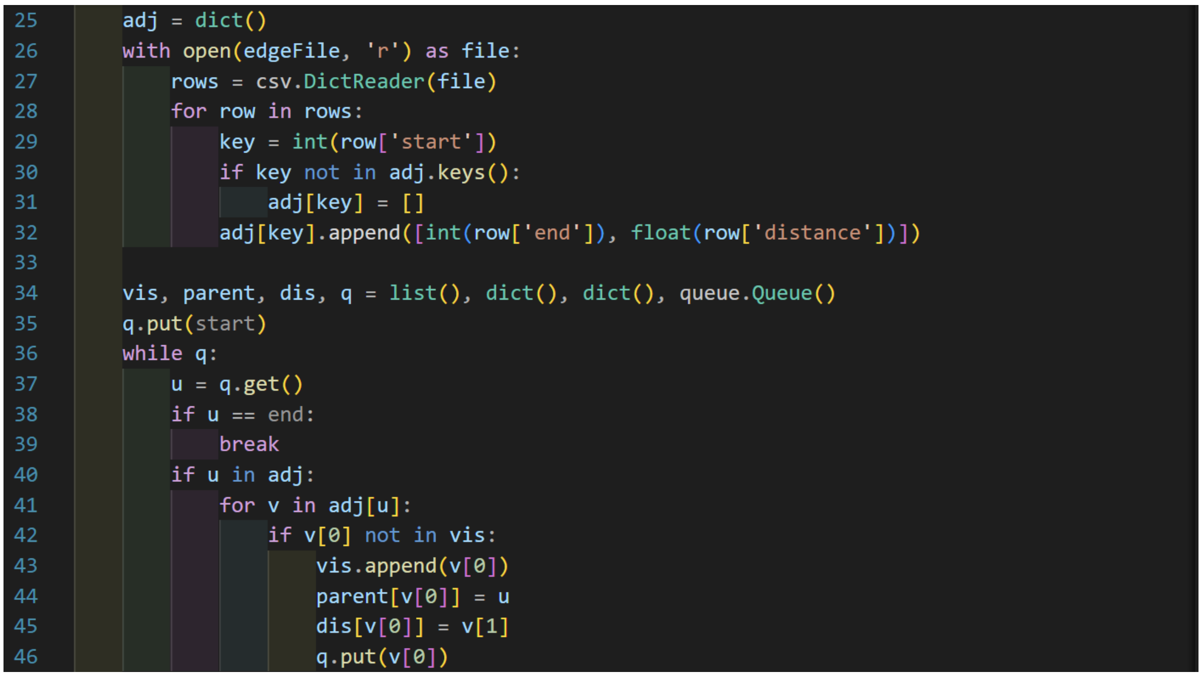
110550108 施柏江

Part I. Implementation:

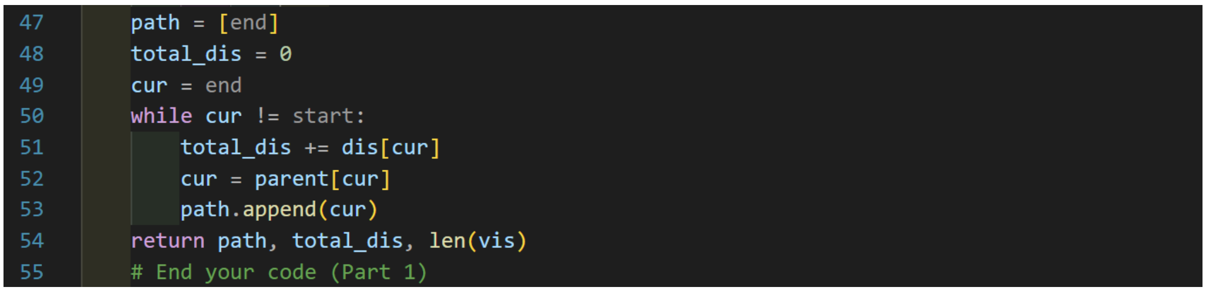
Part 1



```python
import csv
import queue
edgeFile = 'edges.csv'

def bfs(start, end):
    # Begin your code (Part 1)
    """
    1. Create a dictionary 'adj' to store the adjacency list of the map.
        key: local start node
        value: [local end node, distance from local start node to local end node]
    2. Use 'DictReader()' to read the csv file in the form of Python dictionary.
    3. Use 'vis' to store nodes which are visited by the bfs algorithm;
        'parent' to store the previous node in the path of each node;
        'dis' to store the distance from the previous node to the current node of each node;
        'q' to store nodes which we are going to visit.
    4. Run a while loop until q is empty or we have reached the global end node.
    5. In the while loop, we only visited each node at most once.
        We visited each neighbor of the current node,
        and recorded neighbors' parent, distance, and then push them into the queue,
        which is an FIFO data structure.
    6. Finally, we can find the distance and the reverse path from the global start node to
        the global end node by the parent of each node. Since we only need the length of the
        path, there is no need to reverse the path at the end.
    """
```

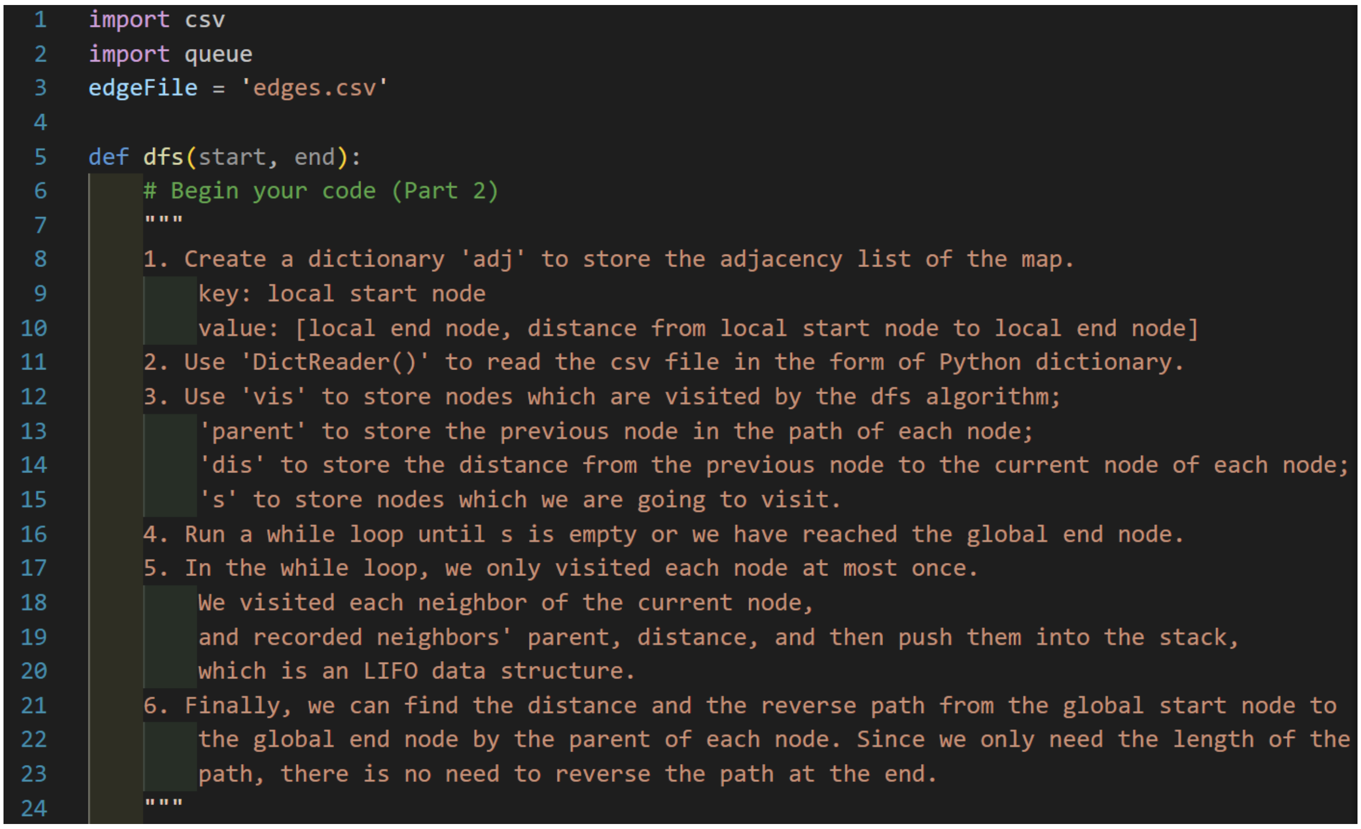


```python
    adj = dict()
    with open(edgeFile, 'r') as file:
        rows = csv.DictReader(file)
        for row in rows:
            key = int(row['start'])
            if key not in adj.keys():
                adj[key] = []
            adj[key].append([int(row['end']), float(row['distance'])])

    vis, parent, dis, q = list(), dict(), dict(), queue.Queue()
    q.put(start)
    while q:
        u = q.get()
        if u == end:
            break
        if u in adj:
            for v in adj[u]:
                if v[0] not in vis:
                    vis.append(v[0])
                    parent[v[0]] = u
                    dis[v[0]] = v[1]
                    q.put(v[0])
```

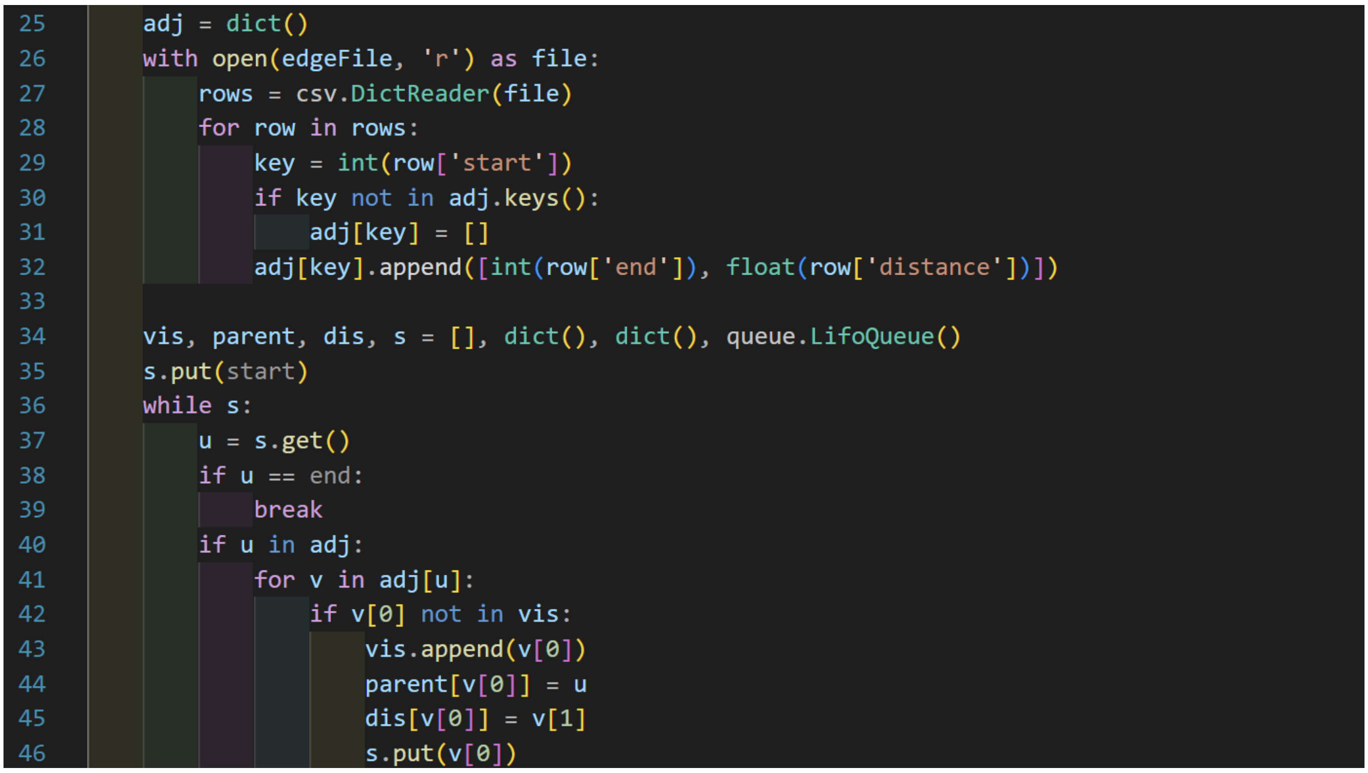


```python
    path = [end]
    total_dis = 0
    cur = end
    while cur != start:
        total_dis += dis[cur]
        cur = parent[cur]
        path.append(cur)
    return path, total_dis, len(vis)
    # End your code (Part 1)
```

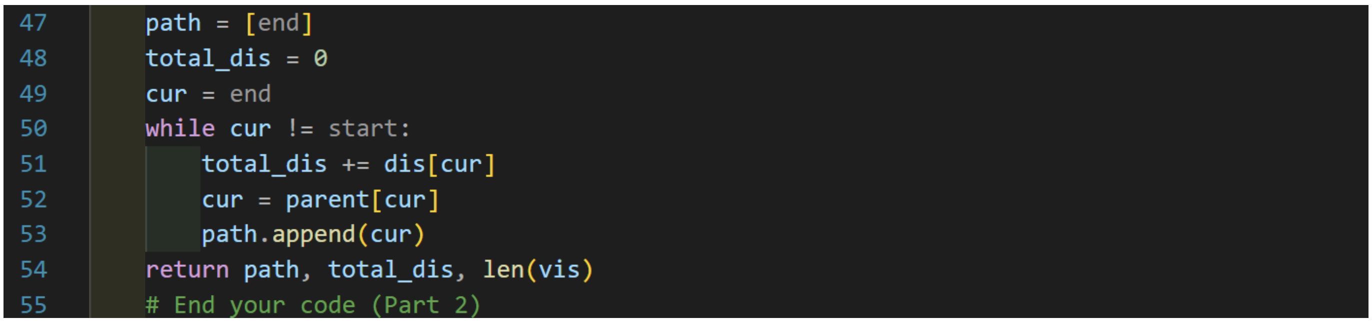
Part 2



```python
import csv
import queue
edgeFile = 'edges.csv'

def dfs(start, end):
    # Begin your code (Part 2)
    """
    1. Create a dictionary 'adj' to store the adjacency list of the map.
        key: local start node
        value: [local end node, distance from local start node to local end node]
    2. Use 'DictReader()' to read the csv file in the form of Python dictionary.
    3. Use 'vis' to store nodes which are visited by the dfs algorithm;
        'parent' to store the previous node in the path of each node;
        'dis' to store the distance from the previous node to the current node of each node;
        's' to store nodes which we are going to visit.
    4. Run a while loop until s is empty or we have reached the global end node.
    5. In the while loop, we only visited each node at most once.
        We visited each neighbor of the current node,
        and recorded neighbors' parent, distance, and then push them into the stack,
        which is an LIFO data structure.
    6. Finally, we can find the distance and the reverse path from the global start node to
        the global end node by the parent of each node. Since we only need the length of the
        path, there is no need to reverse the path at the end.
    """
```

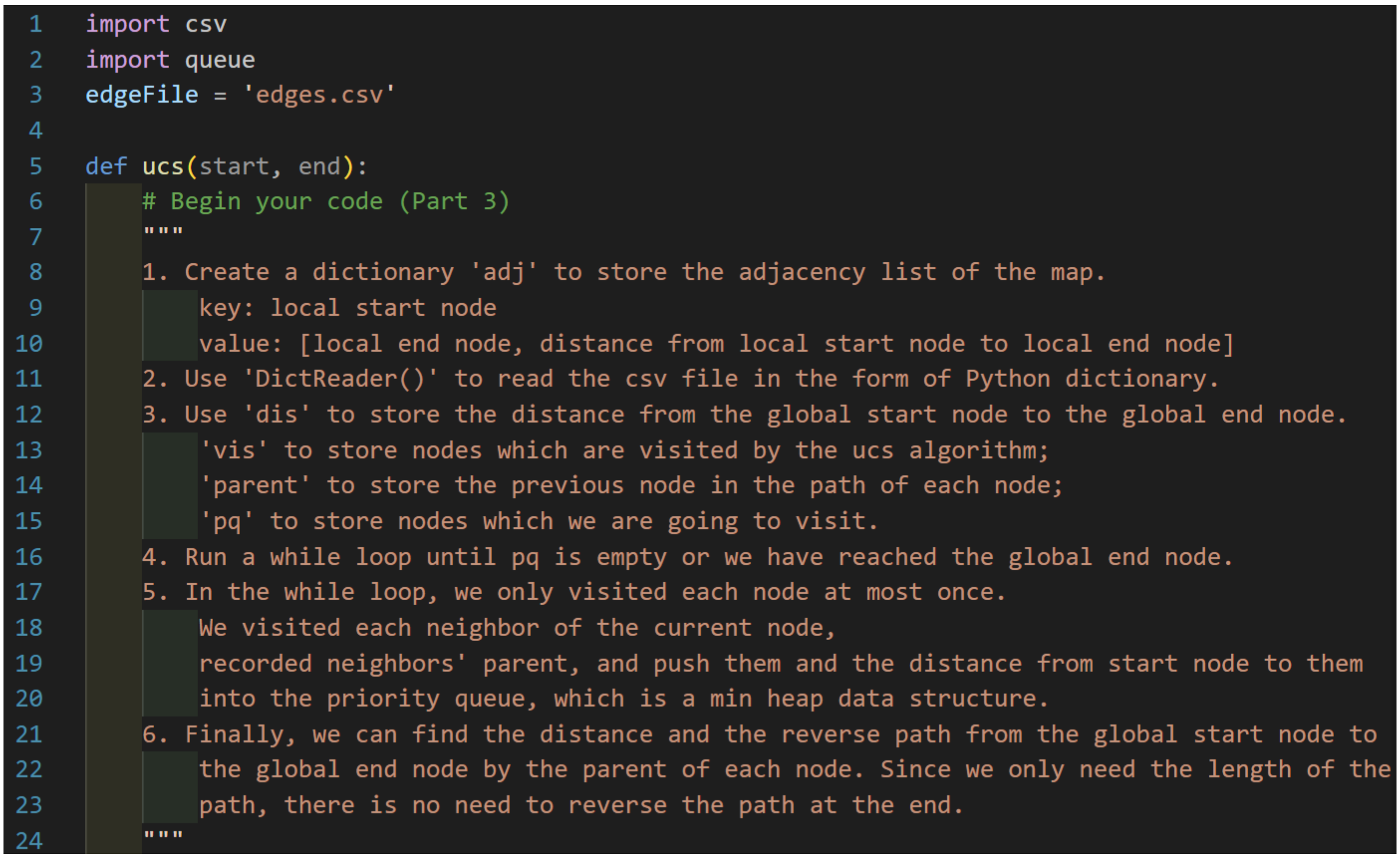


```python
    adj = dict()
    with open(edgeFile, 'r') as file:
        rows = csv.DictReader(file)
        for row in rows:
            key = int(row['start'])
            if key not in adj.keys():
                adj[key] = []
            adj[key].append([int(row['end']), float(row['distance'])])

    vis, parent, dis, s = [], dict(), dict(), queue.LifoQueue()
    s.put(start)
    while s:
        u = s.get()
        if u == end:
            break
        if u in adj:
            for v in adj[u]:
                if v[0] not in vis:
                    vis.append(v[0])
                    parent[v[0]] = u
                    dis[v[0]] = v[1]
                    s.put(v[0])
```

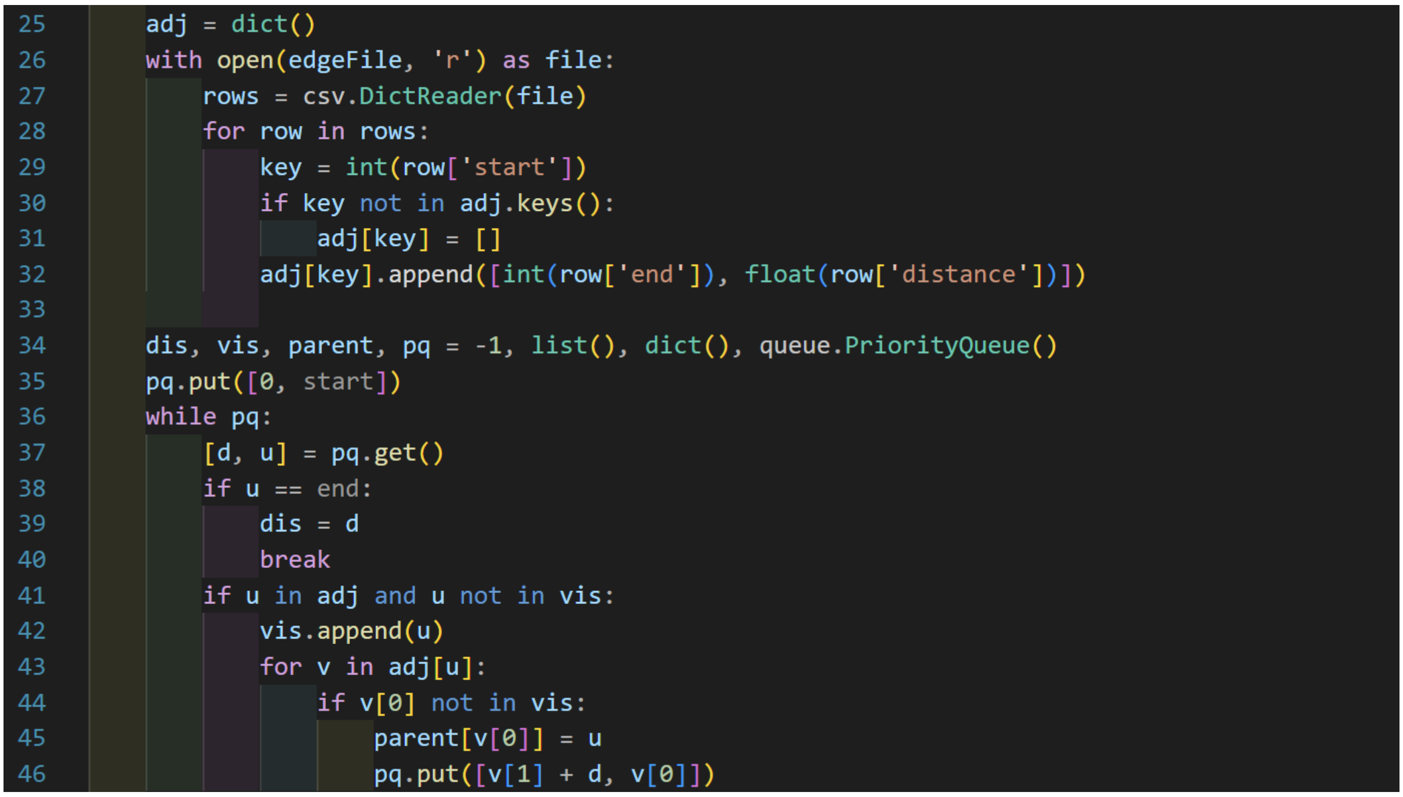


```python
    path = [end]
    total_dis = 0
    cur = end
    while cur != start:
        total_dis += dis[cur]
        cur = parent[cur]
        path.append(cur)
    return path, total_dis, len(vis)
    # End your code (Part 2)
```

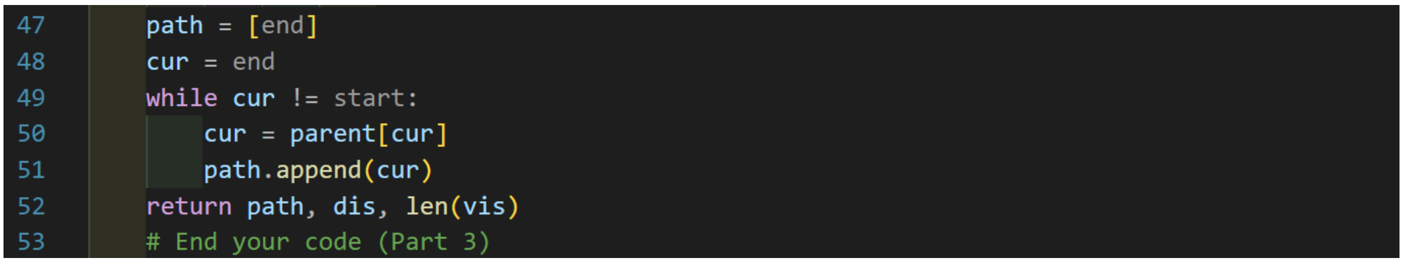
Part 3



```python
import csv
import queue
edgeFile = 'edges.csv'

def ucs(start, end):
    # Begin your code (Part 3)
    """
    1. Create a dictionary 'adj' to store the adjacency list of the map.
        key: local start node
        value: [local end node, distance from local start node to local end node]
    2. Use 'DictReader()' to read the csv file in the form of Python dictionary.
    3. Use 'dis' to store the distance from the global start node to the global end node.
        'vis' to store nodes which are visited by the ucs algorithm;
        'parent' to store the previous node in the path of each node;
        'pq' to store nodes which we are going to visit.
    4. Run a while loop until pq is empty or we have reached the global end node.
    5. In the while loop, we only visited each node at most once.
        We visited each neighbor of the current node,
        recorded neighbors' parent, and push them and the distance from start node to them
        into the priority queue, which is a min heap data structure.
    6. Finally, we can find the distance and the reverse path from the global start node to
        the global end node by the parent of each node. Since we only need the length of the
        path, there is no need to reverse the path at the end.
    """
```

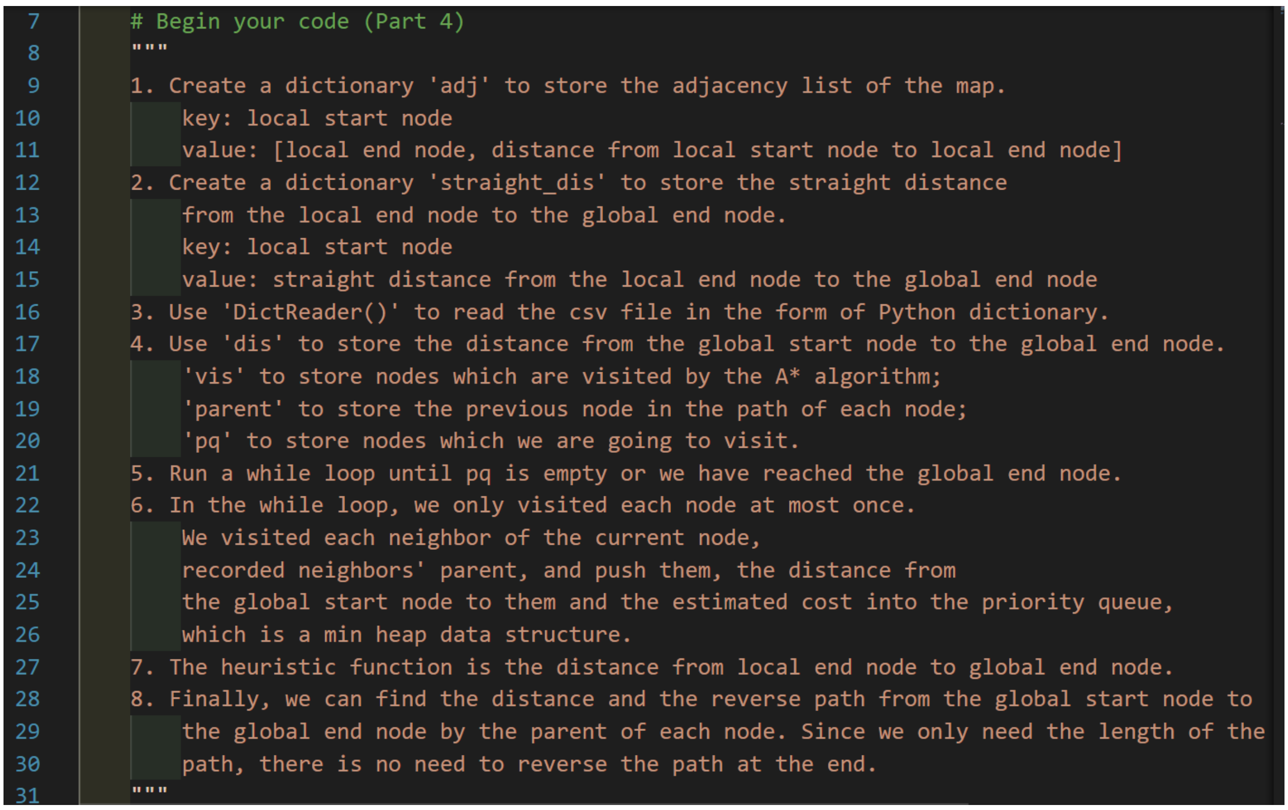


```python
    adj = dict()
    with open(edgeFile, 'r') as file:
        rows = csv.DictReader(file)
        for row in rows:
            key = int(row['start'])
            if key not in adj.keys():
                adj[key] = []
            adj[key].append([int(row['end']), float(row['distance'])])

    dis, vis, parent, pq = -1, list(), dict(), queue.PriorityQueue()
    pq.put([0, start])
    while pq:
        [d, u] = pq.get()
        if u == end:
            dis = d
            break
        if u in adj and u not in vis:
            vis.append(u)
            for v in adj[u]:
                if v[0] not in vis:
                    parent[v[0]] = u
                    pq.put([v[1] + d, v[0]])
```

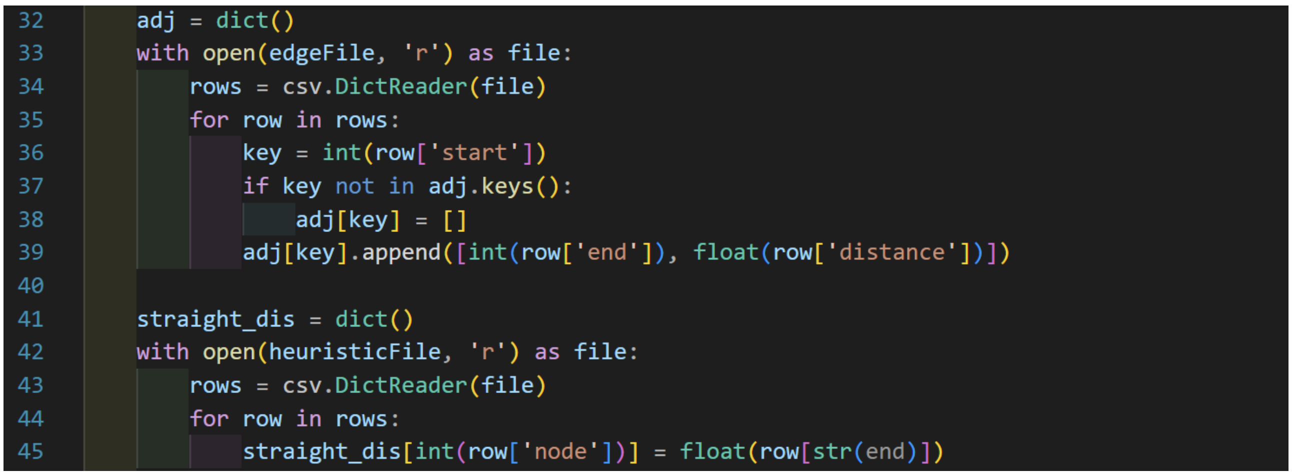


```python
    path = [end]
    cur = end
    while cur != start:
        cur = parent[cur]
        path.append(cur)
    return path, dis, len(vis)
    # End your code (Part 3)
```

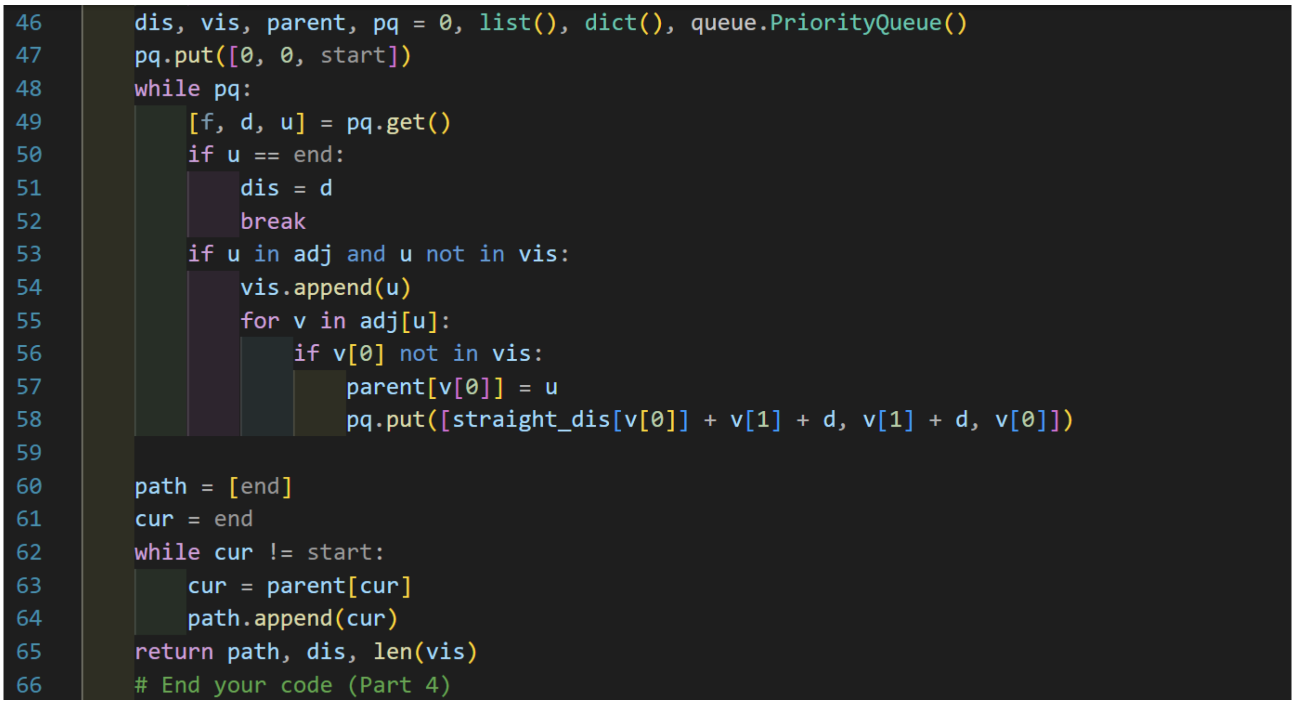
Part 4



```python
    # Begin your code (Part 4)
    """
    1. Create a dictionary 'adj' to store the adjacency list of the map.
        key: local start node
        value: [local end node, distance from local start node to local end node]
    2. Create a dictionary 'straight_dis' to store the straight distance
        from the local end node to the global end node.
        key: local start node
        value: straight distance from the local end node to the global end node
    3. Use 'DictReader()' to read the csv file in the form of Python dictionary.
    4. Use 'dis' to store the distance from the global start node to the global end node.
        'vis' to store nodes which are visited by the A* algorithm;
        'parent' to store the previous node in the path of each node;
        'pq' to store nodes which we are going to visit.
    5. Run a while loop until pq is empty or we have reached the global end node.
    6. In the while loop, we only visited each node at most once.
        We visited each neighbor of the current node,
        recorded neighbors' parent, and push them, the distance from
        the global start node to them and the estimated cost into the priority queue,
        which is a min heap data structure.
    7. The heuristic function is the distance from local end node to global end node.
    8. Finally, we can find the distance and the reverse path from the global start node to
        the global end node by the parent of each node. Since we only need the length of the
        path, there is no need to reverse the path at the end.
    """
```

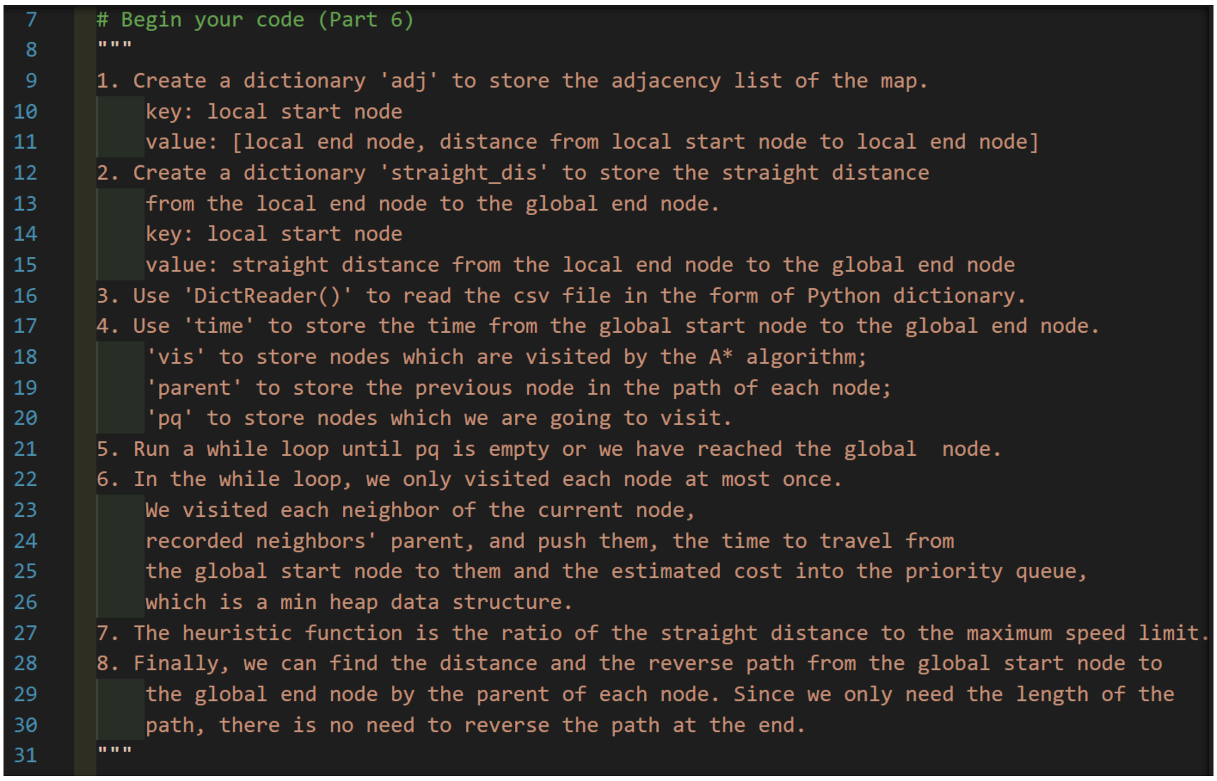


```python
    adj = dict()
    with open(edgeFile, 'r') as file:
        rows = csv.DictReader(file)
        for row in rows:
            key = int(row['start'])
            if key not in adj.keys():
                adj[key] = []
            adj[key].append([int(row['end']), float(row['distance'])])

    straight_dis = dict()
    with open(heuristicFile, 'r') as file:
        rows = csv.DictReader(file)
        for row in rows:
            straight_dis[int(row['node'])] = float(row[str(end)])
```

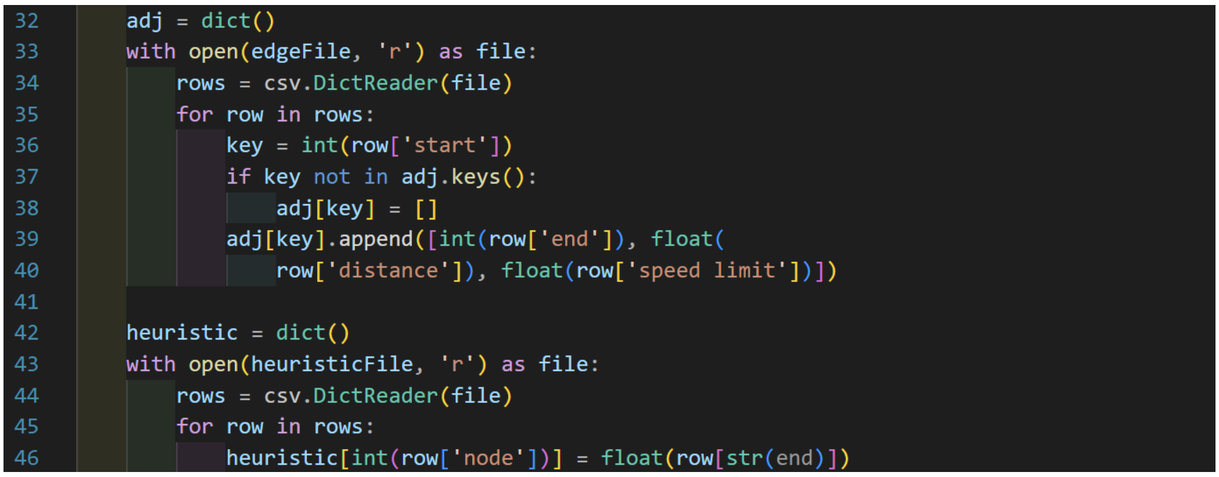


```python
    dis, vis, parent, pq = 0, list(), dict(), queue.PriorityQueue()
    pq.put([0, 0, start])
    while pq:
        [f, d, u] = pq.get()
        if u == end:
            dis = d
            break
        if u in adj and u not in vis:
            vis.append(u)
            for v in adj[u]:
                if v[0] not in vis:
                    parent[v[0]] = u
                    pq.put([straight_dis[v[0]] + v[1] + d, v[1] + d, v[0]])

    path = [end]
    cur = end
    while cur != start:
        cur = parent[cur]
        path.append(cur)
    return path, dis, len(vis)
    # End your code (Part 4)
```

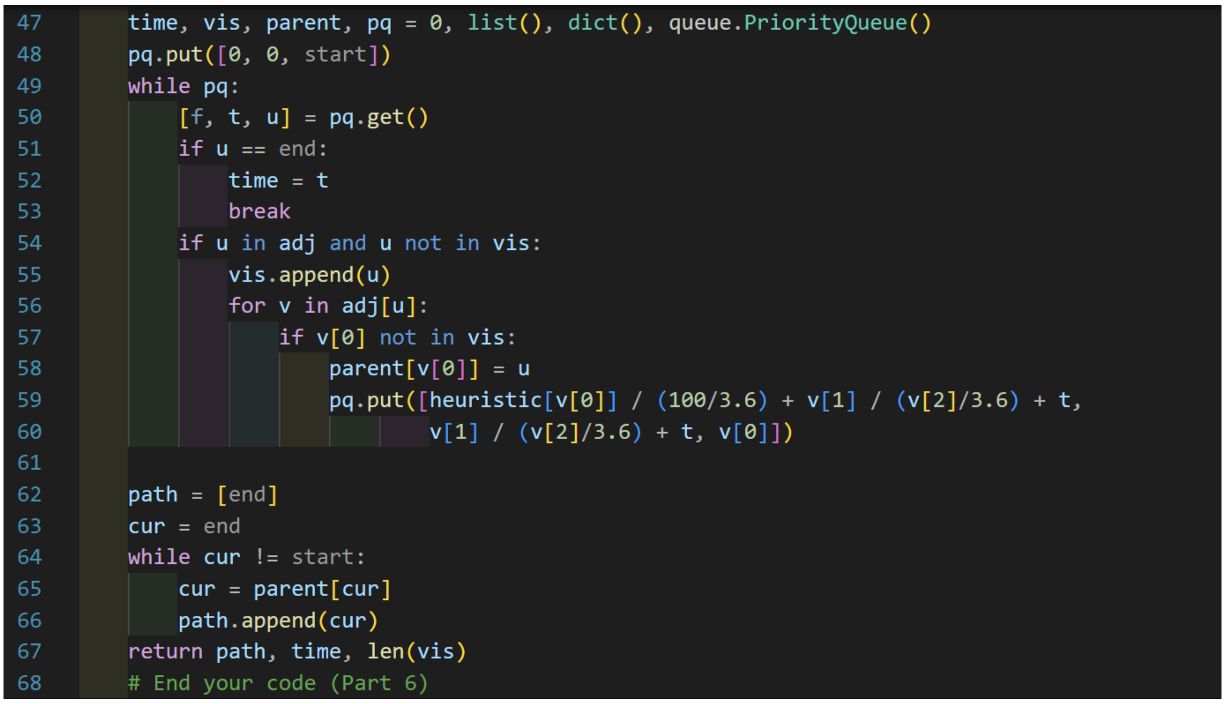
Part 6



```python
# Begin your code (Part 6)
"""
1. Create a dictionary 'adj' to store the adjacency list of the map.
    key: local start node
    value: [local end node, distance from local start node to local end node]
2. Create a dictionary 'straight_dis' to store the straight distance
    from the local end node to the global end node.
    key: local start node
    value: straight distance from the local end node to the global end node
3. Use 'DictReader()' to read the csv file in the form of Python dictionary.
4. Use 'time' to store the time from the global start node to the global end node.
    'vis' to store nodes which are visited by the A* algorithm;
    'parent' to store the previous node in the path of each node;
    'pq' to store nodes which we are going to visit.
5. Run a while loop until pq is empty or we have reached the global  node.
6. In the while loop, we only visited each node at most once.
    We visited each neighbor of the current node,
    recorded neighbors' parent, and push them, the time to travel from
    the global start node to them and the estimated cost into the priority queue,
    which is a min heap data structure.
7. The heuristic function is the ratio of the straight distance to the maximum speed limit.
8. Finally, we can find the distance and the reverse path from the global start node to
    the global end node by the parent of each node. Since we only need the length of the
    path, there is no need to reverse the path at the end.
"""
```



```python
    adj = dict()
    with open(edgeFile, 'r') as file:
        rows = csv.DictReader(file)
        for row in rows:
            key = int(row['start'])
            if key not in adj.keys():
                adj[key] = []
            adj[key].append([int(row['end']), float(
                row['distance']), float(row['speed limit'])])

    heuristic = dict()
    with open(heuristicFile, 'r') as file:
        rows = csv.DictReader(file)
        for row in rows:
            heuristic[int(row['node'])] = float(row[str(end)])
```



```python
    time, vis, parent, pq = 0, list(), dict(), queue.PriorityQueue()
    pq.put([0, 0, start])
    while pq:
        [f, t, u] = pq.get()
        if u == end:
            time = t
            break
        if u in adj and u not in vis:
            vis.append(u)
            for v in adj[u]:
                if v[0] not in vis:
                    parent[v[0]] = u
                    pq.put([heuristic[v[0]] / (100/3.6) + v[1] / (v[2]/3.6) + t,
                            v[1] / (v[2]/3.6) + t, v[0]])

    path = [end]
    cur = end
    while cur != start:
        cur = parent[cur]
        path.append(cur)
    return path, time, len(vis)
    # End your code (Part 6)
```

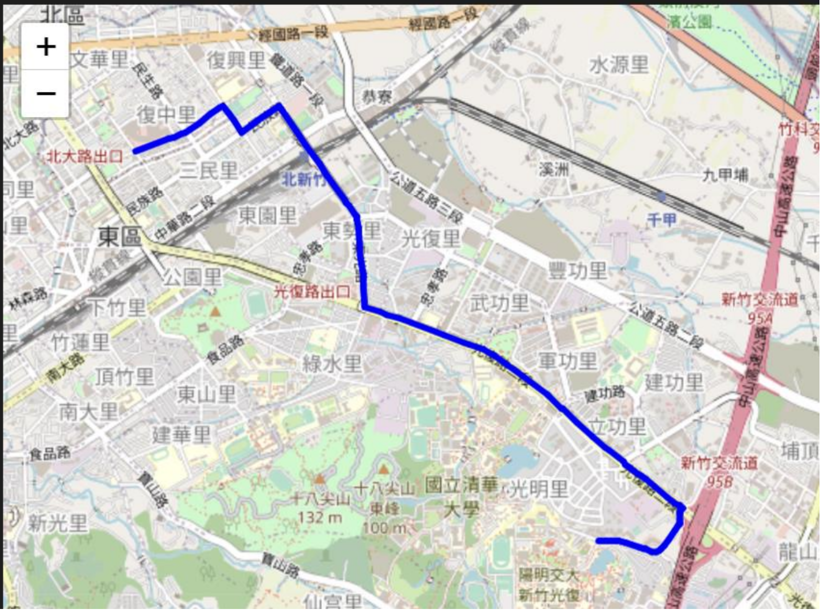
Part II. Results & Analysis:

Test 1：

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

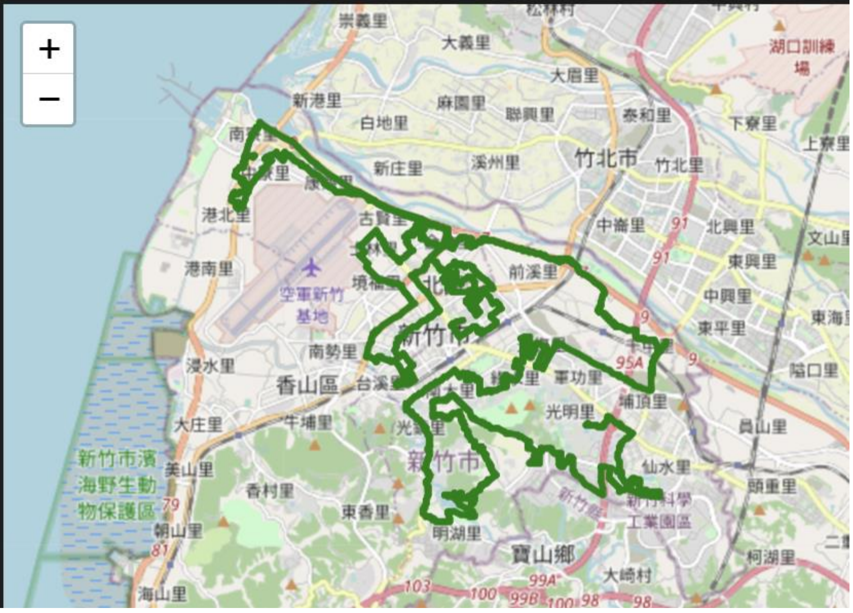
BFS:

```
The number of nodes in the path found by BFS: 88
Total distance of path found by BFS: 4978.881999999998 m
The number of visited nodes in BFS: 4403
```



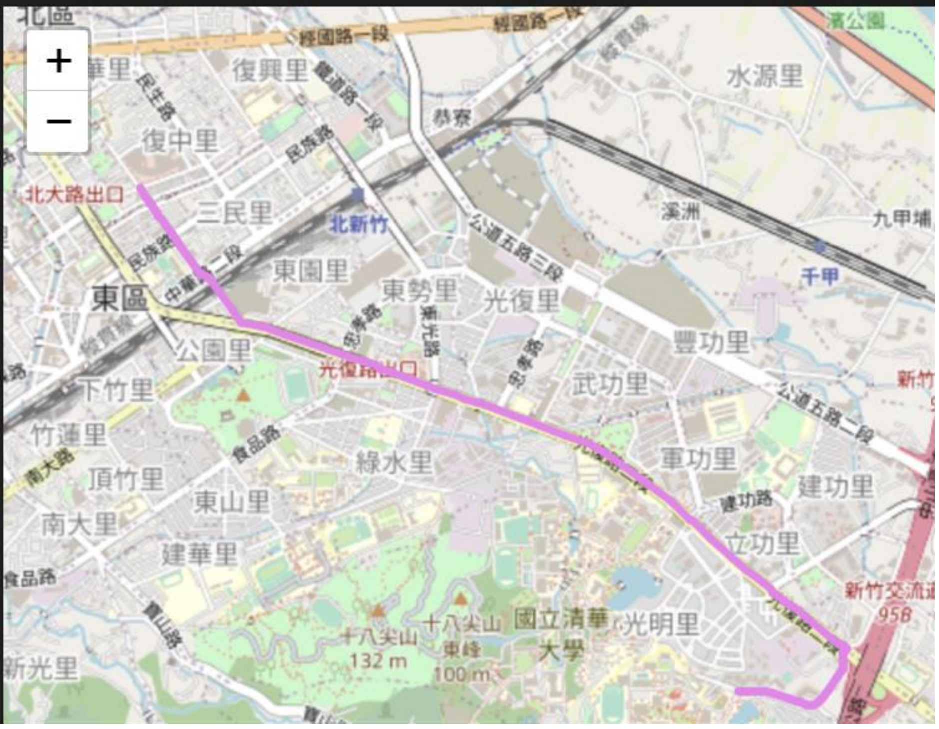
DFS(stack):

```
The number of nodes in the path found by DFS: 1718
Total distance of path found by DFS: 75504.3150000001 m
The number of visited nodes in DFS: 5236
```



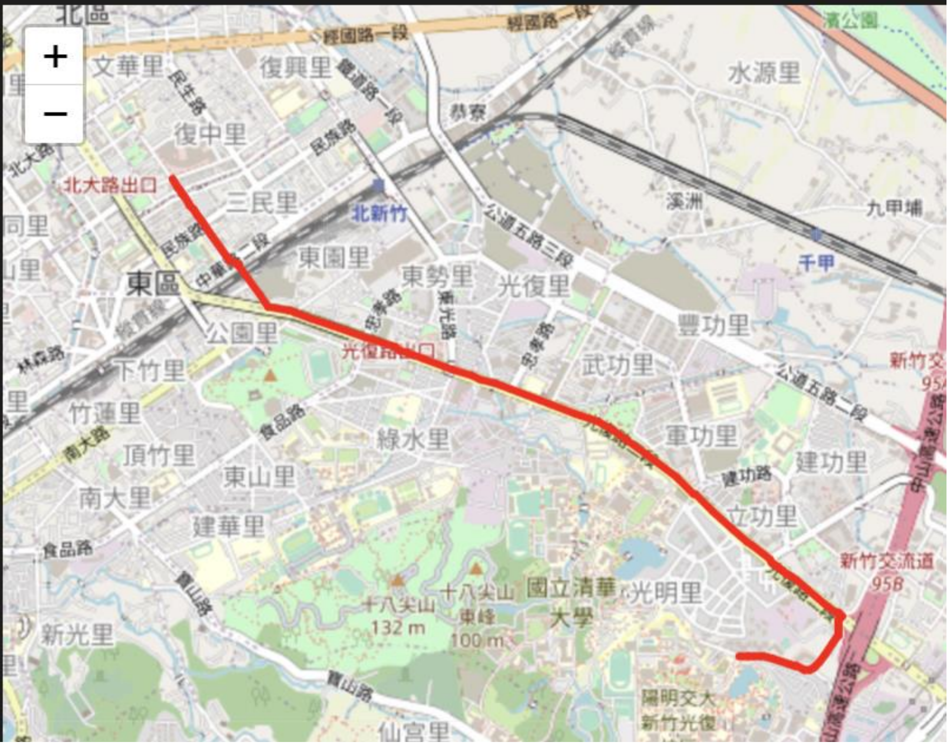
UCS:

```
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: 5076
```

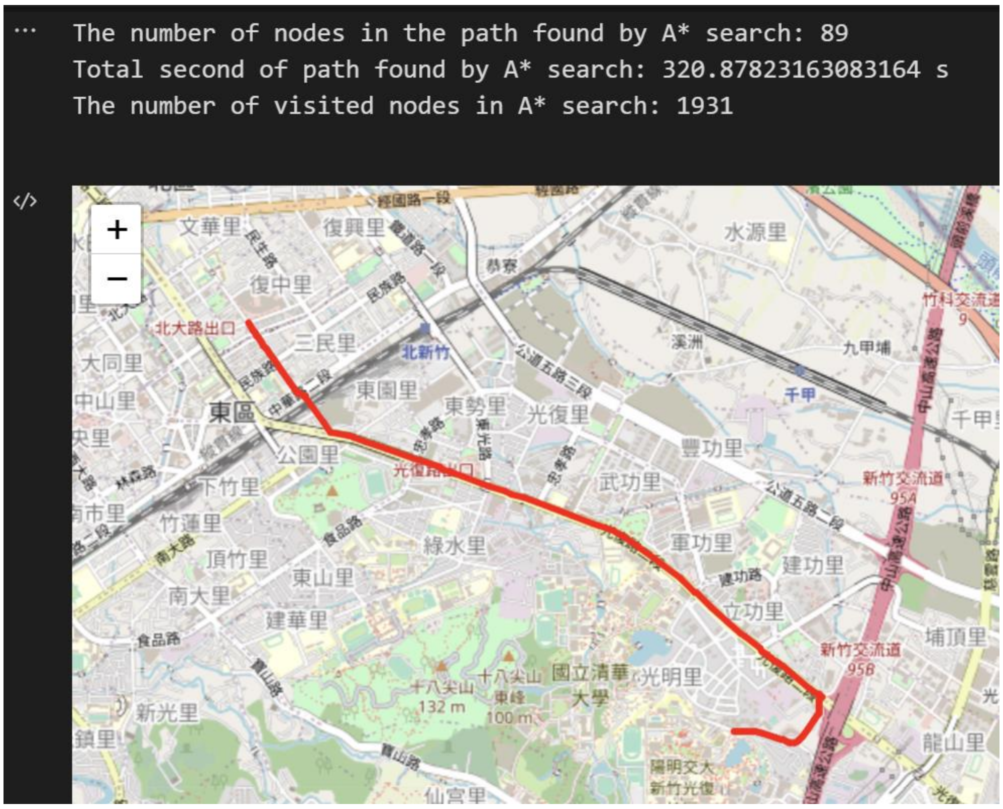


A*:

```
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: 260
```



Part 6 A*(time):

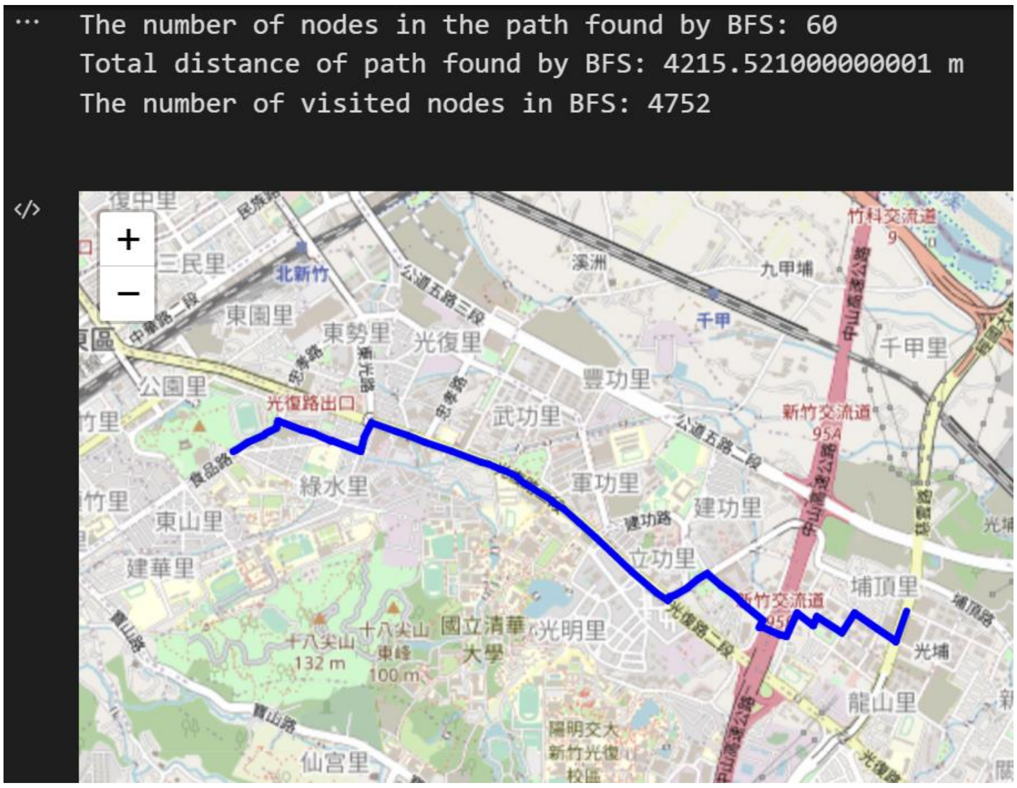


```
The number of nodes in the path found by A* search: 89
Total second of path found by A* search: 320.87823163083164 s
The number of visited nodes in A* search: 1931
```

Test 2：

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

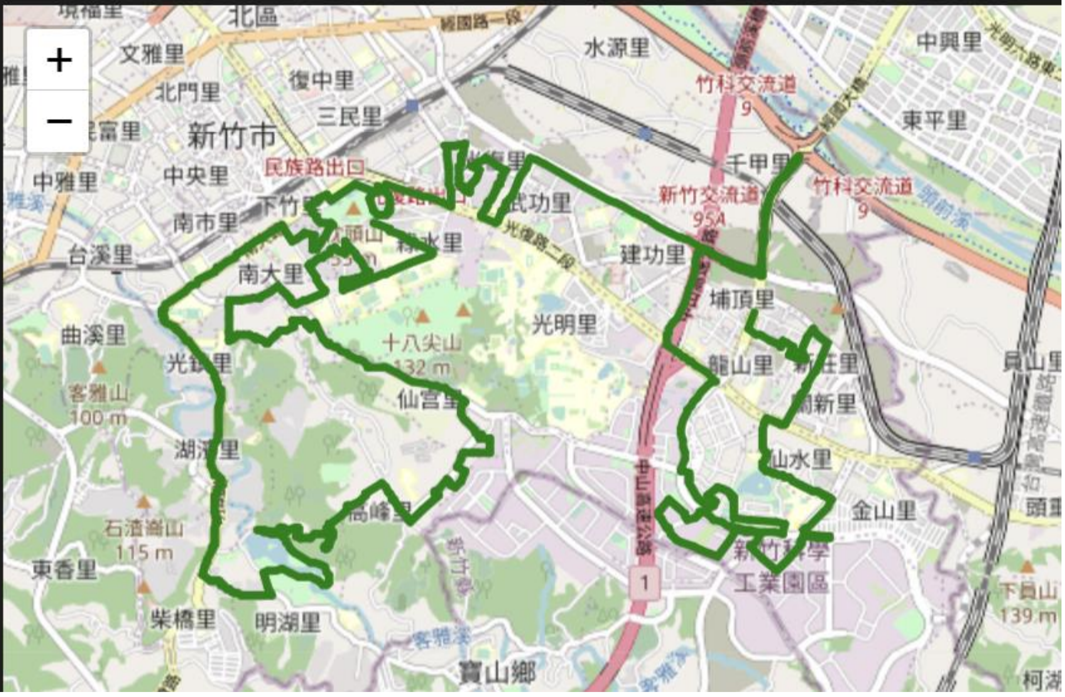
BFS:



```
The number of nodes in the path found by BFS: 60
Total distance of path found by BFS: 4215.521000000001 m
The number of visited nodes in BFS: 4752
```

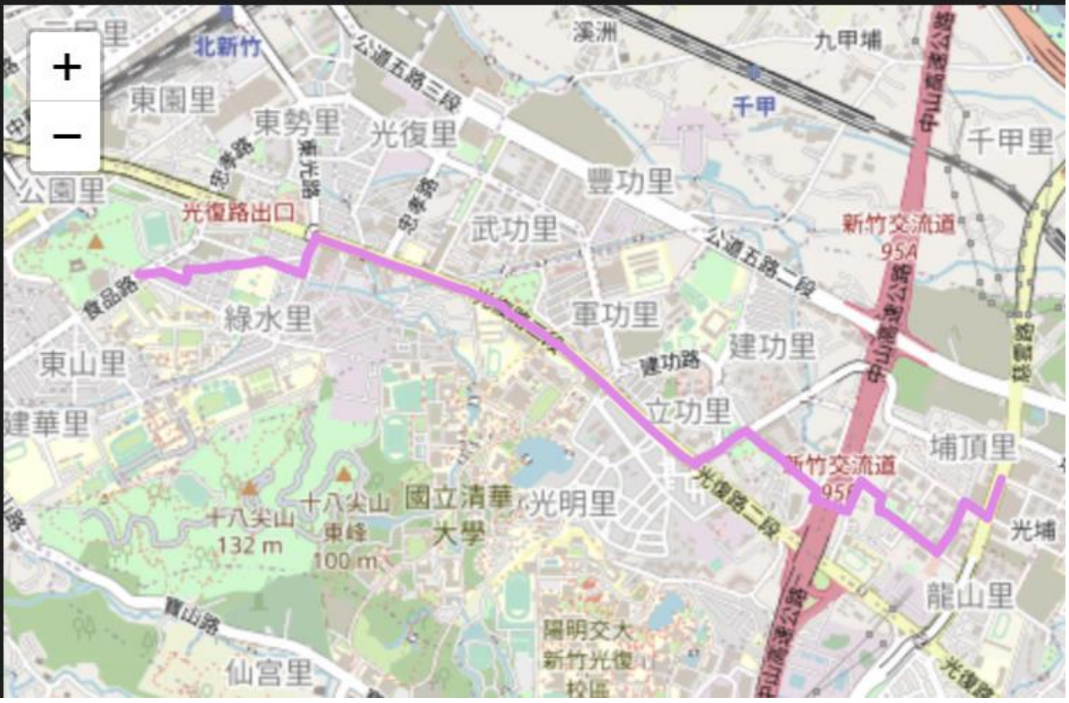
DFS(stack):

```
The number of nodes in the path found by DFS: 930
Total distance of path found by DFS: 38752.307999999895 m
The number of visited nodes in DFS: 9616
```



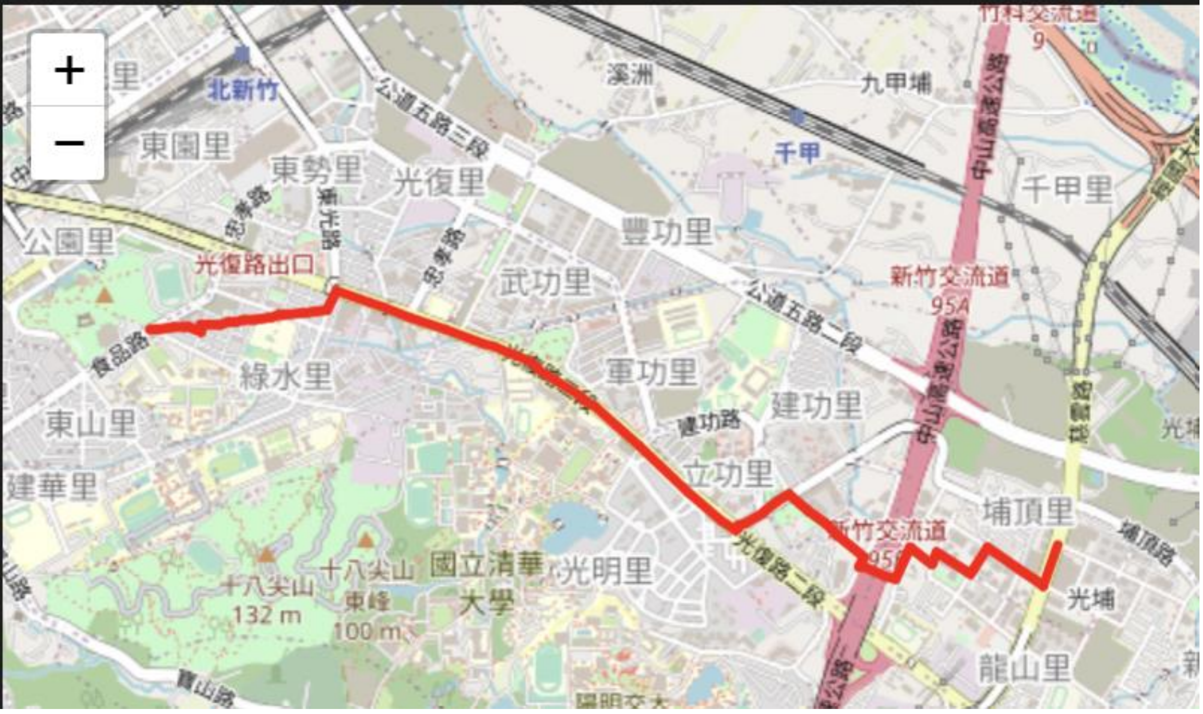
UCS:

```
The number of nodes in the path found by UCS: 72
Total distance of path found by UCS: 4101.84 m
The number of visited nodes in UCS: 7206
```



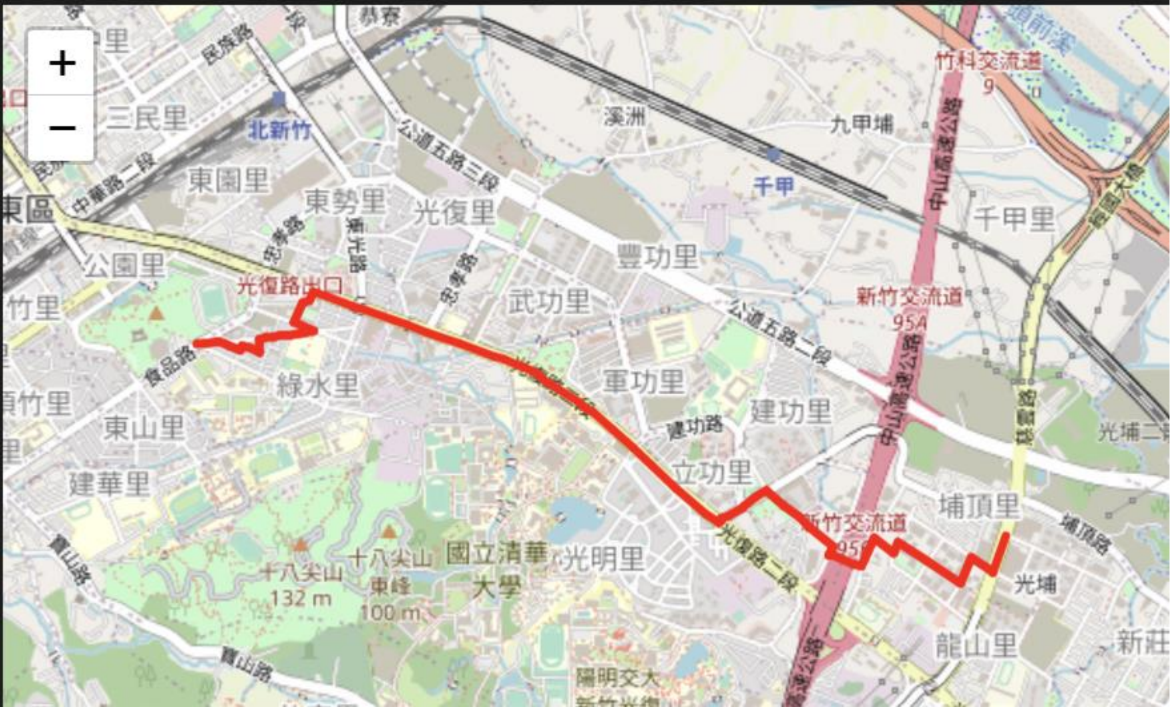
A*:

```
The number of nodes in the path found by A* search: 64
Total distance of path found by A* search: 4101.84 m
The number of visited nodes in A* search: 1170
```



Part 6 A*(time):

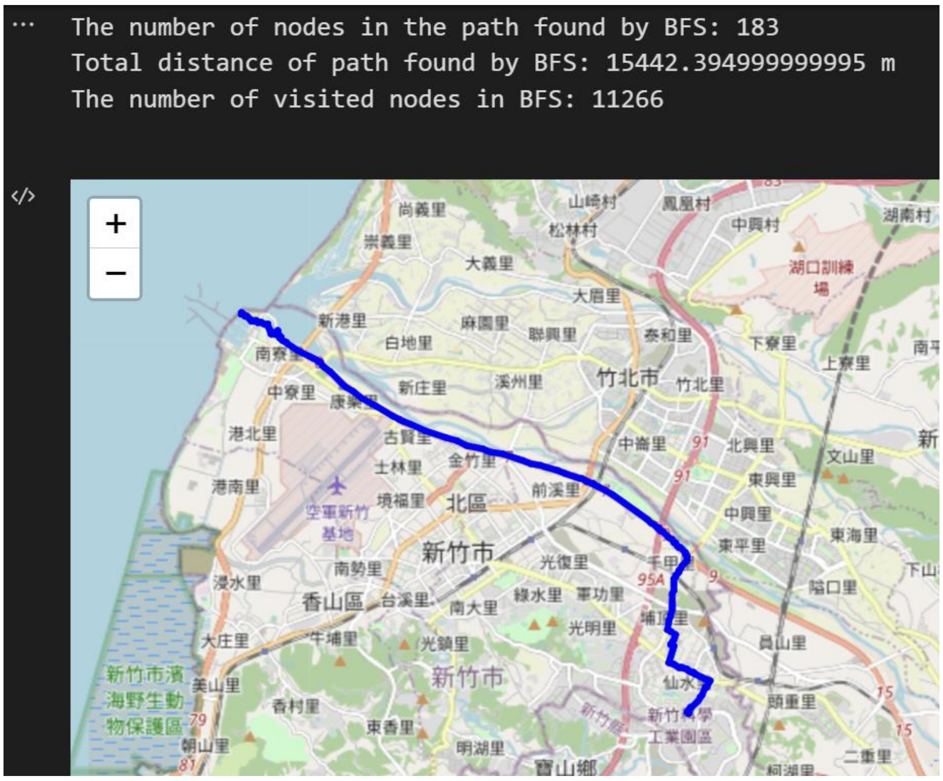
```
The number of nodes in the path found by A* search: 70
Total second of path found by A* search: 304.4436634360302 s
The number of visited nodes in A* search: 2868
```



Test 3：

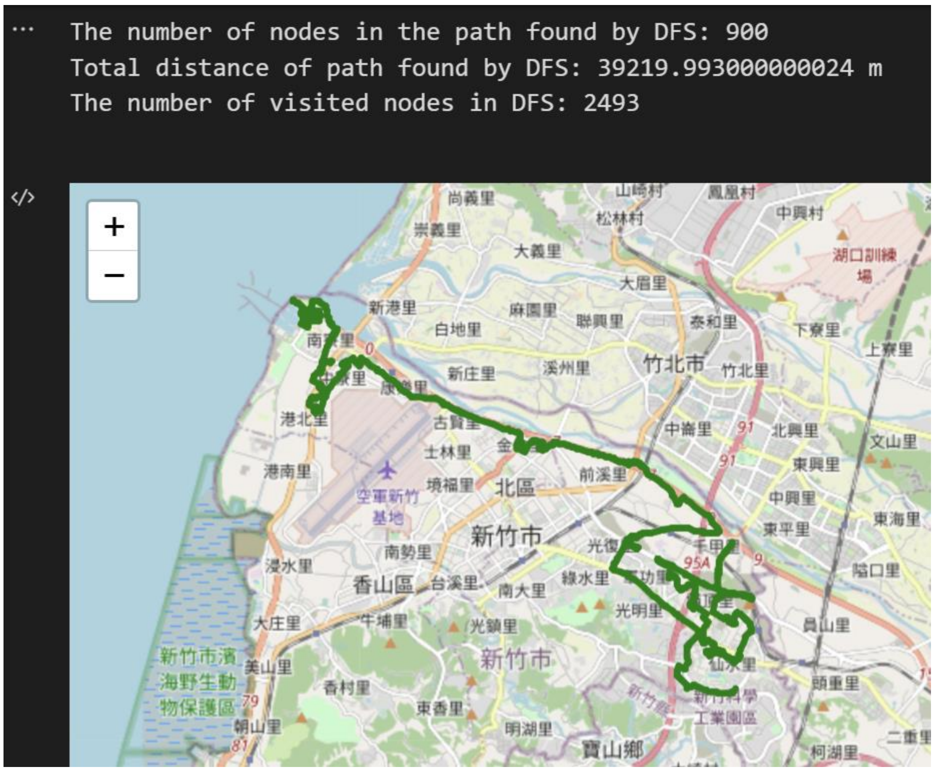
from National Experimental High School At Hsinchu Science Park (ID: 1718165260) to Nanliao Fishing Port (ID: 8513026827)

BFS:



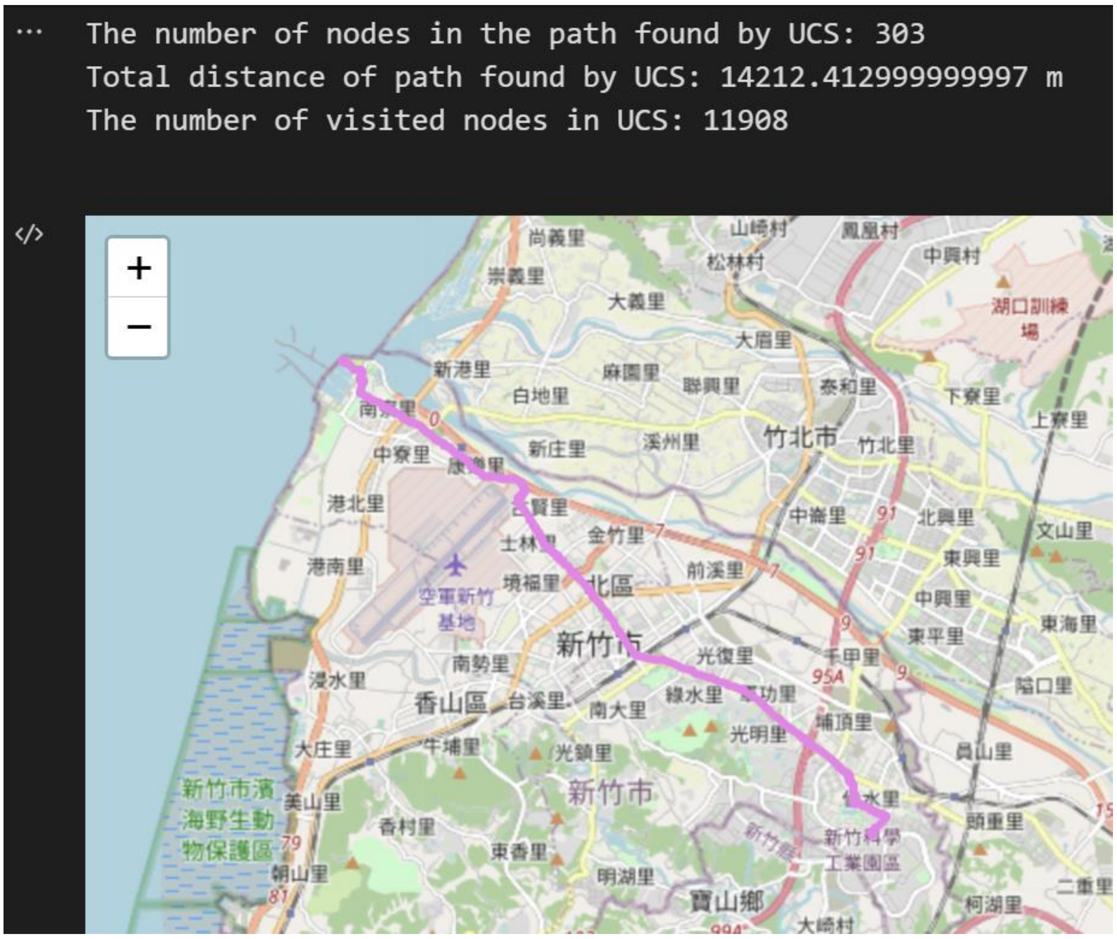
```
The number of nodes in the path found by BFS: 183
Total distance of path found by BFS: 15442.394999999995 m
The number of visited nodes in BFS: 11266
```

DFS(stack):



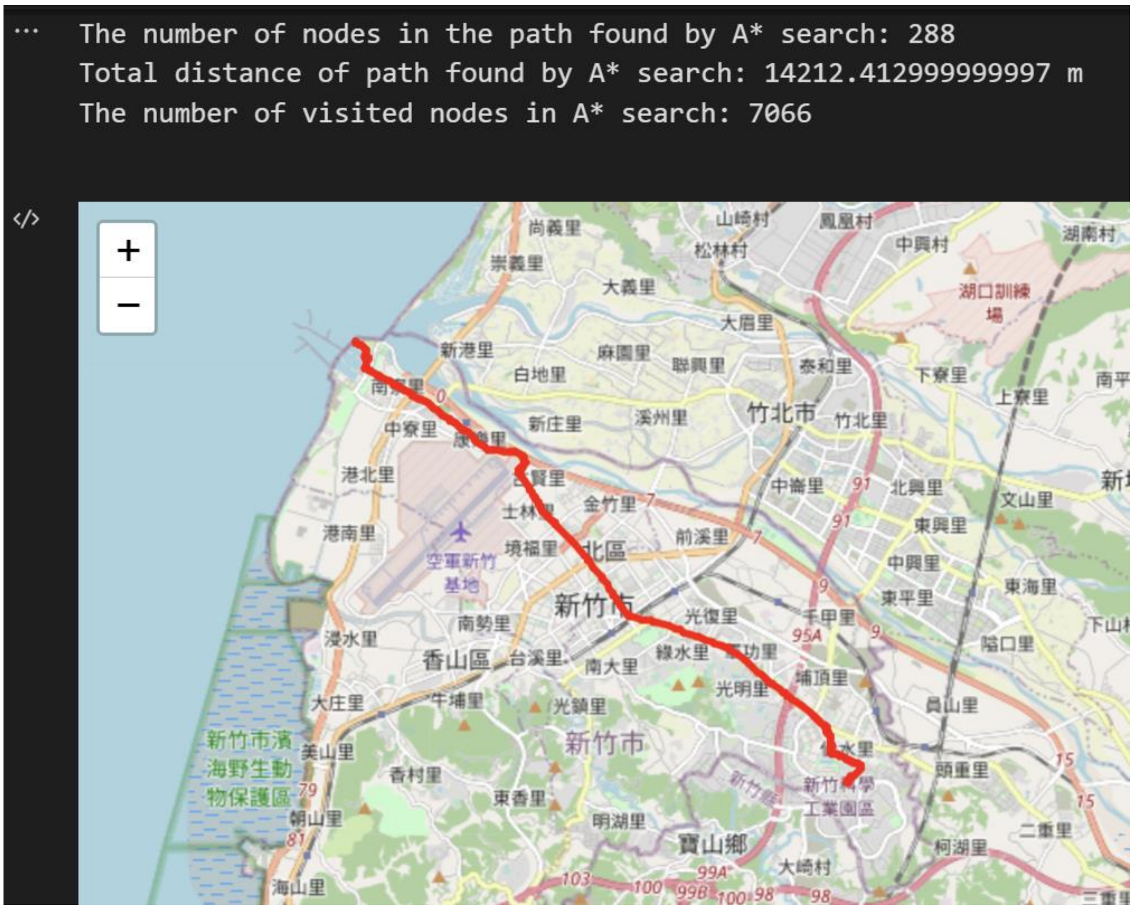
```
The number of nodes in the path found by DFS: 900
Total distance of path found by DFS: 39219.993000000024 m
The number of visited nodes in DFS: 2493
```

UCS:



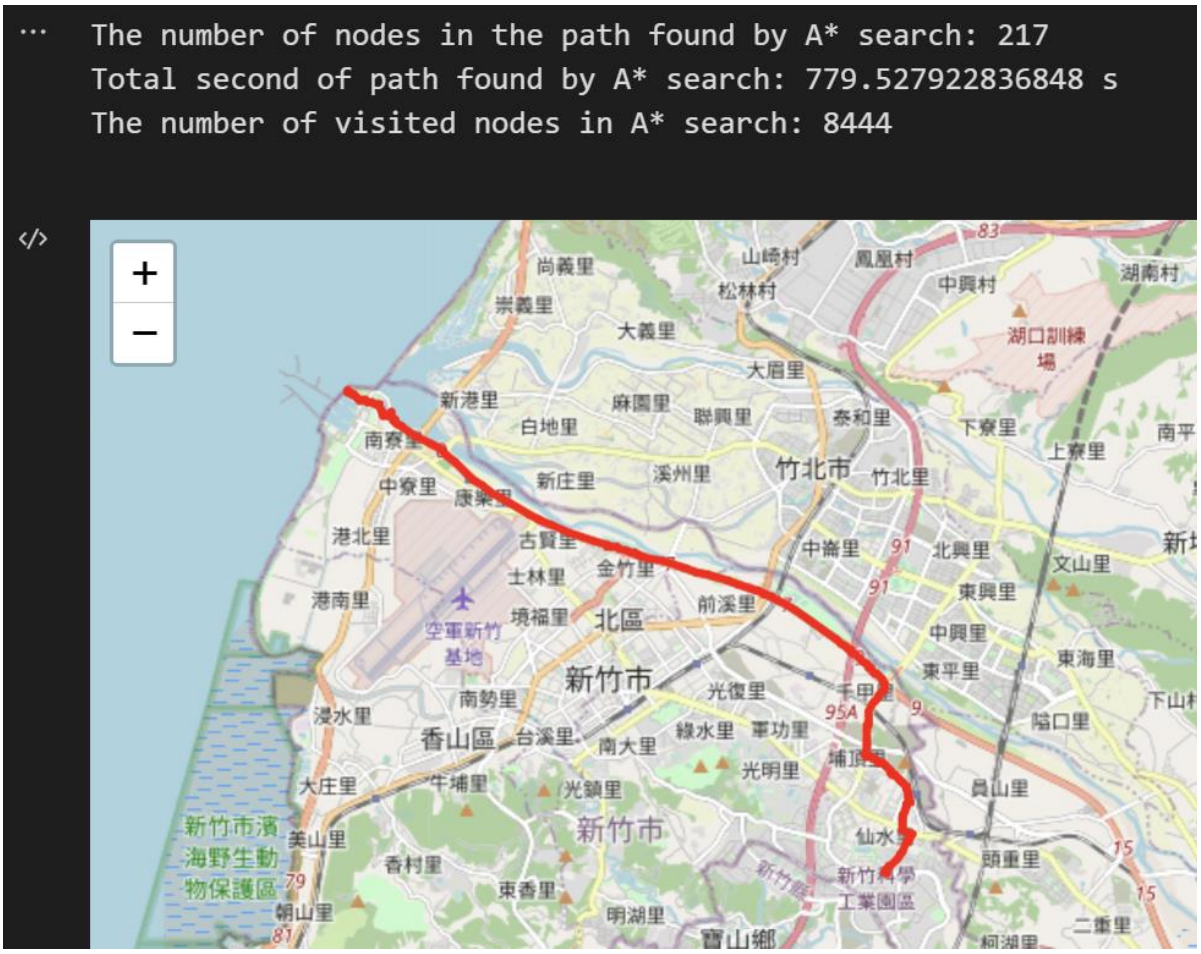
```
The number of nodes in the path found by UCS: 303
Total distance of path found by UCS: 14212.412999999997 m
The number of visited nodes in UCS: 11908
```

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: 7066
```

Part 6 A*(time):



```
The number of nodes in the path found by A* search: 217
Total second of path found by A* search: 779.527922836848 s
The number of visited nodes in A* search: 8444
```

BFS: It can find a path with the minimum number of nodes. Because the distance between each node is not a fixed value, it cannot guarantee the shortest path.

DFS: It wastes a lot of time because it will exhaustively search the current path, so the distance obtained is much longer than that of BFS.

UCS: It considers the distance of each node and can obtain the shortest path.

A*: It not only considers the distance between each node, but also uses the straight-line distance between the current node and the destination as a reference, so it can obtain the shortest path and is more efficient than UCS.

A*(time): It considers the time spent between each node and also takes into account the shortest time to reach the destination, so it can obtain the fastest path.
The shortest time here means the straight-line distance divided by the maximum speed limit in the map. Because there is no way has faster time than this, we would not overestimate the cost, and the heuristic function is admissible.

Part III. Question Answering:

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

Ans:

I had difficulty coming up with an admissible heuristic for part 6 in the beginning. To solve it, I review the criteria for admissible heuristics, which requires that the cost of reaching the goal should never be overestimated. Finally, after careful consideration, I was able to generate an idea that fulfilled the criteria.

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.

Ans:

In addition to speed limits and distance, we also need to consider the number of traffic lights and traffic congestion in the real world. Both factors can cause the speed of vehicles to slow down or even stop, resulting in an increase in travel time, so we must take them into account.

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

Ans:

Mapping: We can use lidar sensors to create the map of the places we want.

Localization: We can use GPS technology to determine the device's position on the map.

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.

Ans:

We can define the heuristic function h(x) = prep_time + delivery_time + traffic_time

where prep_time is the time required to prepare the meal, delivery_time is the time required to delivery the meal, and traffic_time is the time needed additionally based on current traffic flow.

The time to prepare the meal depending on the complexity of the order and the restaurant's workload, so when the restaurant is busy, we can add some time to the ETA. Similarly, if the traffic is heavy, we can add additional time to the ETA.