交大資工 110550062
朱致伶

Homework 2: Route Finding

Part I. Implementation (6%):

- **Please screenshot your code snippets of Part 1 ~ Part 4 and explain your implementation.**
- **<u>BFS with queue</u>:**

```python
def bfs(start, end):
# Begin your code (Part 1)
    """
    Build a graph with edge.csv, and implement BFS with queue(FIFO)
    """
    with open("edges.csv", 'r') as edgeFile:
        data = csv.reader(edgeFile)
        # skip the first line(header) of the data
        next(data)
        # declare graph as defaultdict(list) to store nodes
        graph = defaultdict(list)
        for row in data:
            # store the node and corresponding information
            # transform data type to int or float, since the original data type is string
            graph[int(row[0])].append((int(row[1]), float(row[2]), float(row[3])))
            # row[0]:start, row[1]:end, row[2]:distance, row[3]:speed limit

    # initialize visited set and queue
    visited = {start}
    queue = deque([(start, [start], 0)])
    while queue:
        # tuple in queue stores (variable) queue: deque[tuple[Any, list, Literal[0]]] _current)
        (cur, path, distance) = queue.popleft()
        # traverse through neighbors of current_node
        for neighbor, neighbor_distance, neighbor_speed_limit in graph[cur]:
            if neighbor == end:
                return path+[neighbor], distance+neighbor_distance, len(visited)
            if neighbor not in visited:
                # store neighbor_node and its information in queue, traverse its neighbors later
                queue.append((neighbor, path+[neighbor], distance+neighbor_distance))
                visited.add(neighbor)
    return [], 0, 0
# End your code (Part 1)
```

交大資工 110550062
朱致伶

- **<u>DFS with stack</u>:**

```python
def dfs(start, end):
# Begin your code (Part 2)
    """
    Build a graph with edge.csv, and implement DFS with stack(FILO)
    """
    with open("edges.csv", 'r') as edgeFile:
        data = csv.reader(edgeFile)
        # skip the first line(header) of the data
        next(data)
        # declare graph as defaultdict(list) to store nodes
        graph = defaultdict(list)
        for row in data:
            # store the node and corresponding information
            # transform data type to int or float, since the original data type is string
            graph[int(row[0])].append((int(row[1]), float(row[2]), float(row[3])))
            # row[0]:start, row[1]:end, row[2]:distance, row[3]:speed limit

    # initialize visited set and stack
    visited = set()
    stack = [(start, [start], 0)]
    while stack:
        # tuple in stack stores (current_node, current_path_list, distance_from_start_to_current)
        (cur, path, distance) = stack.pop()
        if cur not in visited:
            # if reach end_node, return data
            if cur == end:
                return path, distance, len(visited)
            # mark current node as visited
            visited.add(cur)
            # traverse through neighbors of current_node
            for neighbor, neighbor_distance, neighbor_speed_limit in graph[cur]:
                if neighbor not in visited:
                    # store neighbor_node and its information in stack, traverse its neighbors later
                    stack.append((neighbor, path+[neighbor], distance+neighbor_distance))
    return [], 0, 0
# End your code (Part 2)
```

交大資工 110550062
朱致伶

- **<u>Uniform Cost Search(USC) with priority queue</u>:**

```python
import csv
from queue import PriorityQueue;
from collections import defaultdict;
edgeFile = 'edges.csv'

def ucs(start, end):
# Begin your code (Part 3)
    """
    Build a graph with edge.csv, and implement Uniform Cost Search(USC) with priority_queue
    Just to remind that priority_queue sorts by the first elements of tuple, distance is set as the first element
    """
    with open("edges.csv", 'r') as edgeFile:
        data = csv.reader(edgeFile)
        # skip the first line(header) of the data
        next(data)
        # declare graph as defaultdict(list) to store nodes
        graph = defaultdict(list)
        for row in data:
            # store the node and corresponding information
            # transform data type to int or float, since the original data type is string
            graph[int(row[0])].append((int(row[1]), float(row[2]), float(row[3])))
            # row[0]:start, row[1]:end, row[2]:distance, row[3]:speed limit

    # initialize visited set and priority queue
    pqueue = PriorityQueue()
    pqueue.put((0, start, [start]))
    visited = set()
    while not pqueue.empty():
        # tuple in priority_queue stores (distance_from_start_to_current, current_node, current_path_list)
        # Since priority_queue sorts by the first elements of the tuple, distance is set as the first element
        (distance, cur, path) = pqueue.get()
        if cur not in visited:
            # mark current node as visited
            visited.add(cur)
            # if reach end_node, return data
            if  cur == end:
                return path, distance, len(visited)
            # traverse through neighbors of current_node
            for neighbor, neighbor_distance, neighbor_speed_limit in graph[cur]:
                if neighbor not in visited:
                    # store neighbor_node and its information in priority_queue, traverse its neighbors later
                    pqueue.put((distance+neighbor_distance, neighbor, path+[neighbor]))
    return [], 0, 0
```

交大資工 110550062
朱致伶

- **A* search with priority queue:**

```python
import csv
from queue import PriorityQueue;
from collections import defaultdict;
edgeFile = 'edges.csv'
heuristicFile = 'heuristic.csv'

def astar(start, end):
# Begin your code (Part 4)
    """
    Build a graph with edge.csv, and implement A* search with priority_queue
    Just to remind that priority_queue sorts by the first elements of tuple, sum_of_pathcost_and_goalproximity is set as the first element
    """
    with open("edges.csv", 'r') as edgeFile, open('heuristic.csv', 'r') as heuristicFile:
        edge_data = csv.reader(edgeFile)
        heuristic_data = csv.reader(heuristicFile)
        # skip the first line(header) of the data
        next(edge_data)
        next(heuristic_data)
        # declare graph and h_func as defaultdict(list) to store nodes and heuristic function of each nodes
        graph = defaultdict(list)
        h_func = defaultdict(list)
        for e_row in edge_data:
            # store the node and corresponding information
            # transform data type to int or float, since the original data type is string
            # e_row[0]:start, e_row[1]:end, e_row[2]:distance, e_row[3]:speed limit
            graph[int(e_row[0])].append((int(e_row[1]), float(e_row[2]), float(e_row[3])))
        for h_row in heuristic_data:
            # transform data type to int or float, since the original data type is string
            # h_row[0]:node_ID
            # h_row[1]:straight-line distance from node to Big City Shopping Mall(ID: 1079387396)
            # h_row[2]:straight-line distance from node to COSTCO Hsinchu Store(ID: 1737223506)
            # h_row[3]:straight-line distance from node to Nanliao Fighing Port(ID: 8513026827)
            h_func[int(h_row[0])].append((float(h_row[1]), float(h_row[2]), float(h_row[3])))

    # determine which column of h_func should be used with the end_node
    if end==1079387396:
        dest = 0
    elif end==1737223506:
        dest = 1
    elif end==8513026827:
        dest = 2

    # initialize visited set and priority queue
    pqueue = PriorityQueue()
    pqueue.put((float(h_func[start][0][dest]), 0, start, [start]))
    visited = set()
    while not pqueue.empty():
        # tuple in priority_queue stores (sum_of_pathcost_and_goalproximity, distance_from_start_to_current, current_node, current_path_list)
        # Since priority_queue sorts by the first elements of the tuple, heuristic_of_distance is set as the first element
        (hd, distance, cur, path) = pqueue.get()
        if cur not in visited:
            # mark current node as visited
            visited.add(cur)
            # if reach end_node, return data
            if  cur == end:
                return path, distance, len(visited)
            # traverse through neighbors of current_node
            for neighbor, neighbor_distance, neighbor_speed_limit in graph[cur]:
                if neighbor not in visited:
                    # store neighbor_node and its information in priority_queue, traverse its neighbors later
                    # hd is the neighbor's new distance + h_func of neighbor
                    straight_line = float(h_func[neighbor][0][dest])
                    pqueue.put((distance+neighbor_distance+straight_line, distance+neighbor_distance, neighbor, path+[neighbor]))
    return [], 0, 0
# End your code (Part 4)
```

交大資工 110550062
朱致伶

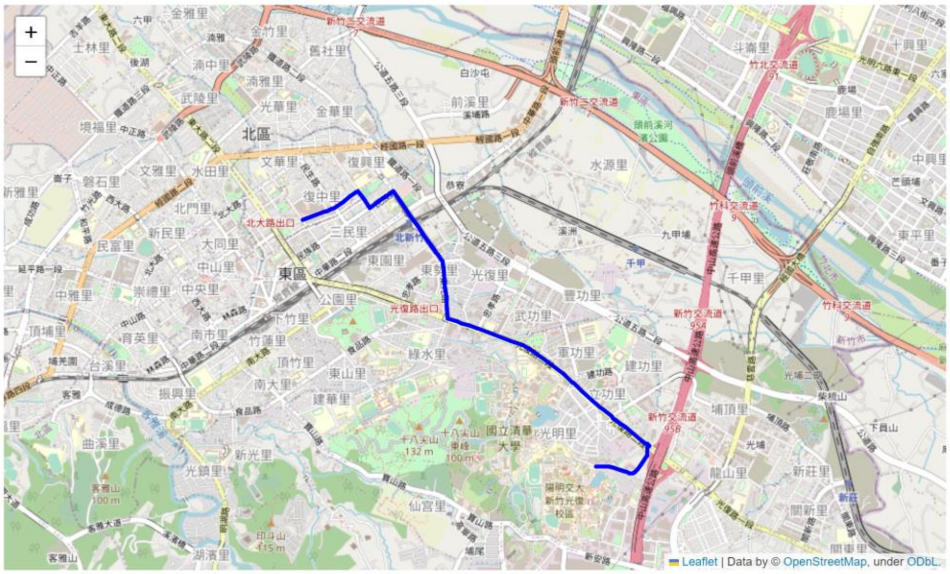
Part II. Results & Analysis (12%): (Please screenshot the results.)

- **Test 1:**

From *National Yang Ming Chiao Tung University (ID: 2270143902)*
To *Big City Shopping Mall (ID: 1079387396)*

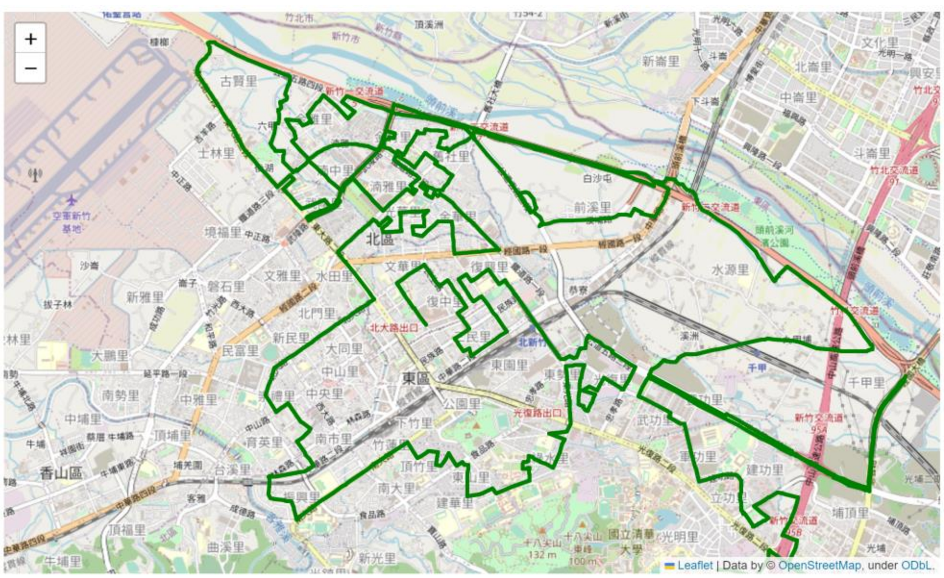
BFS:

```
The number of nodes in the path found by BFS: 88
Total distance of path found by BFS: 4978.8820000000005
The number of visited nodes in BFS: 4273
```



DFS with stack:

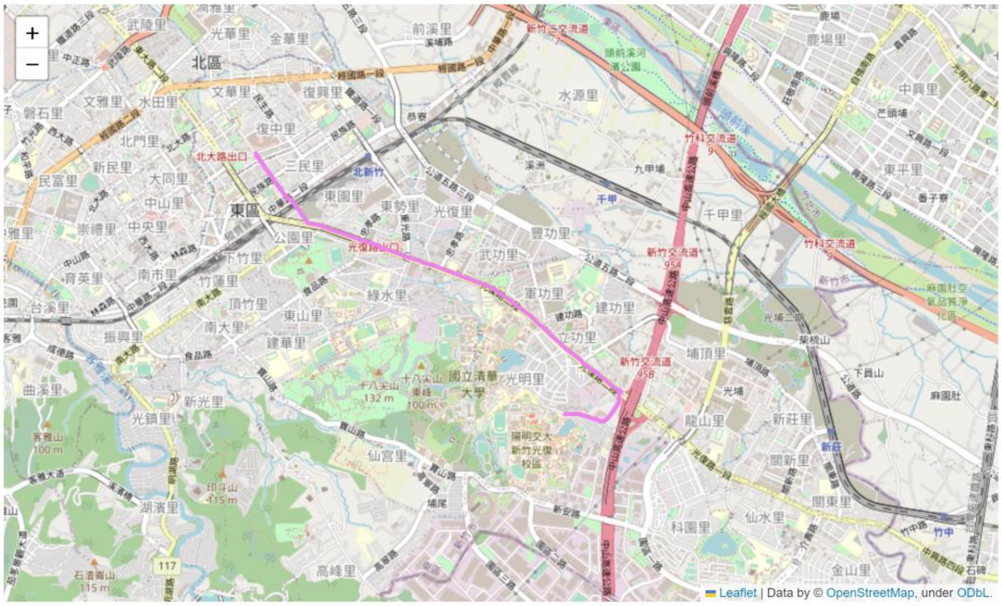
```
The number of nodes in the path found by DFS: 1232
Total distance of path found by DFS: 57208.987000000045 m
The number of visited nodes in DFS: 4210
```



交大資工 110550062
朱致伶

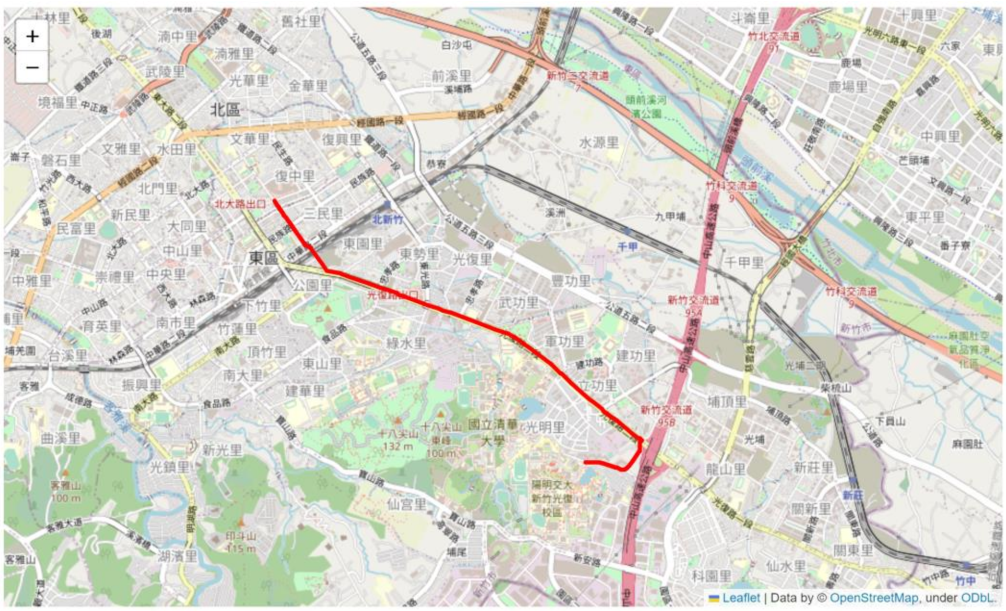
Uniform Cost Search:

```
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* Search:

```
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
```



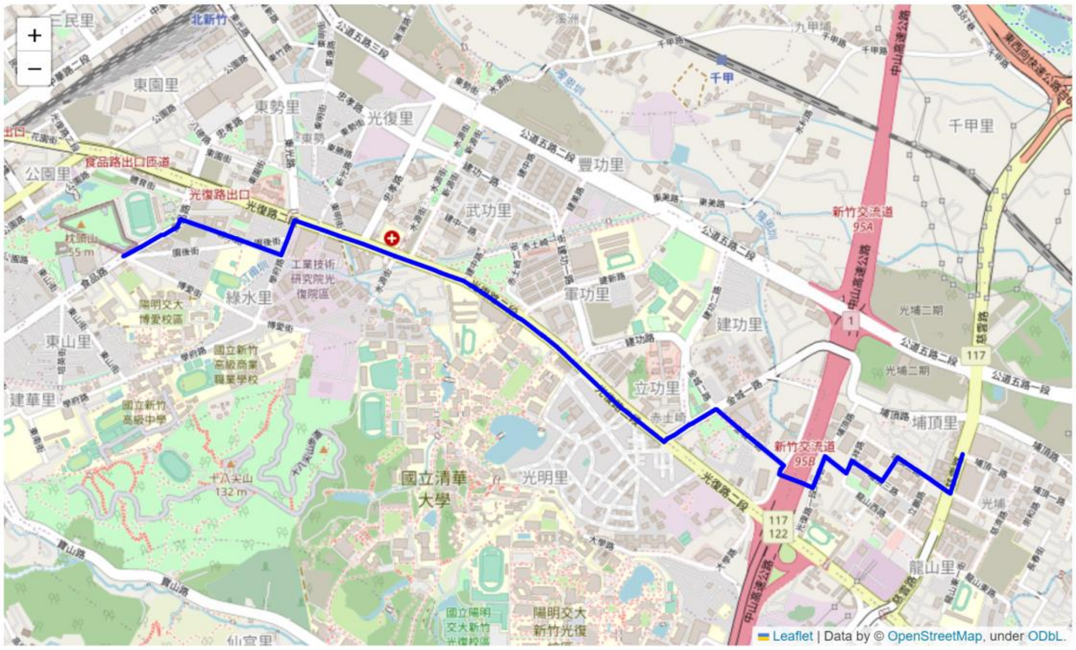
交大資工 110550062
朱致伶

- **Test 2:**

From *Hsinchu Zoo (ID: 426882161)*
To *COSTCO Hsinchu Store (ID: 1737223506)*

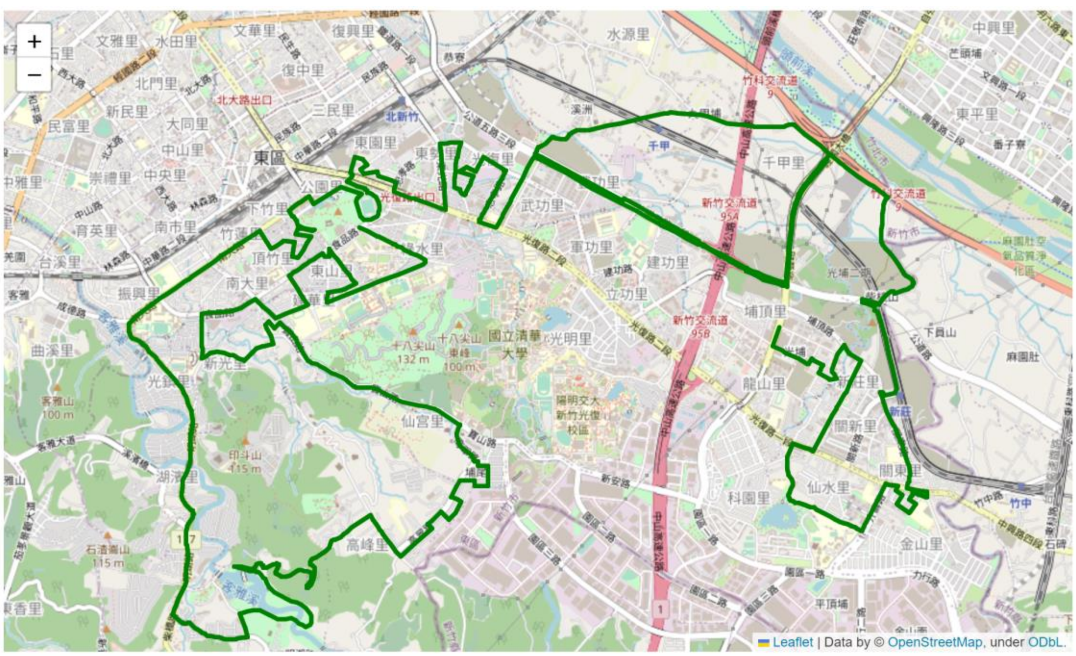
BFS:

```
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: 4606
```



DFS with stack:

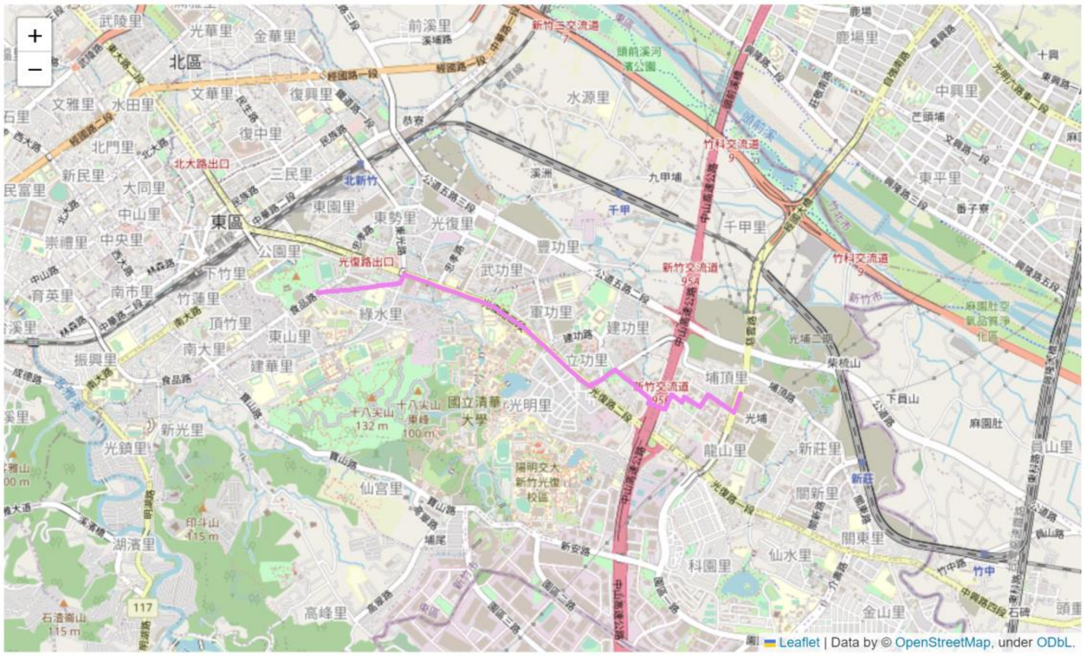
```
The number of nodes in the path found by DFS: 998
Total distance of path found by DFS: 41094.65799999992 m
The number of visited nodes in DFS: 8030
```



交大資工 110550062
朱致伶

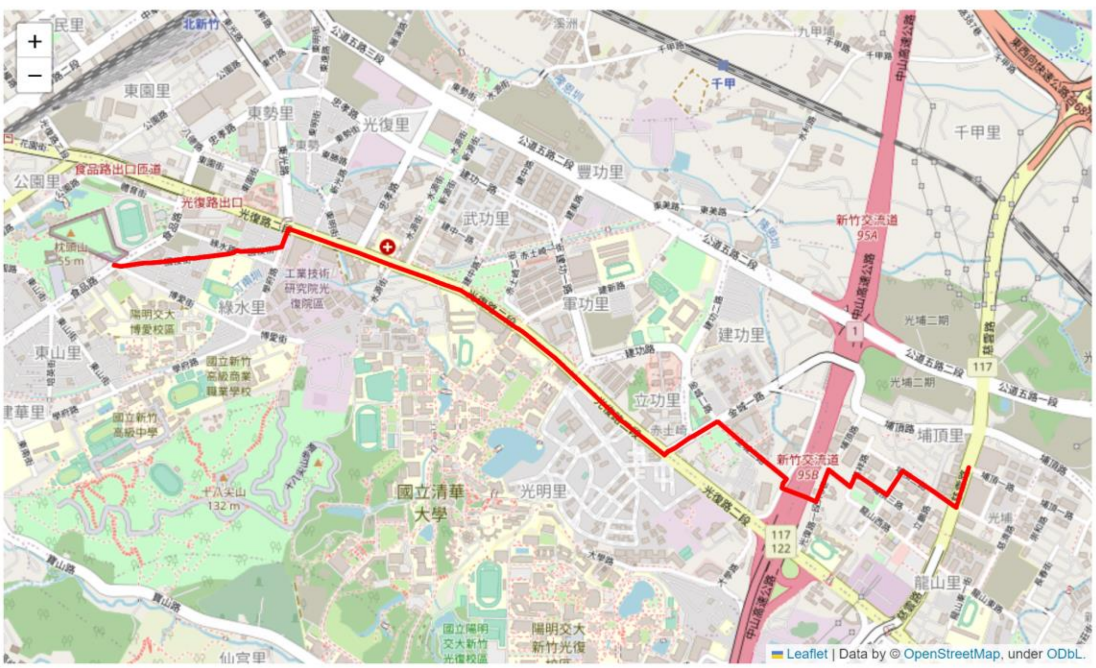
Uniform Cost Search:

```
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* Search:

```
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
```



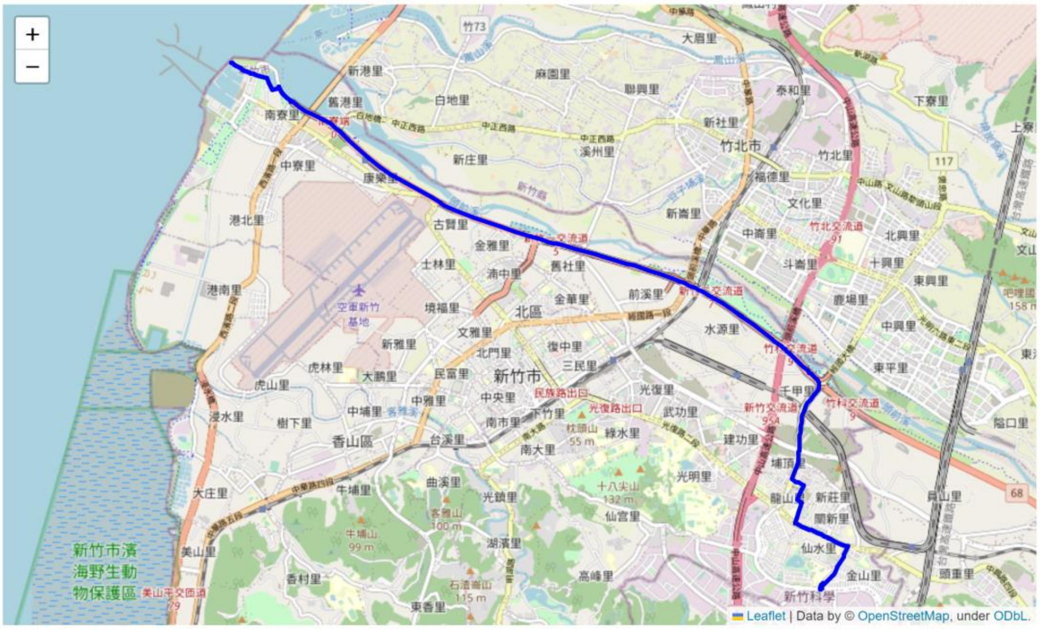
交大資工 110550062
朱致伶

- **Test 3:**

From *National Experimental High School At Hsinchu Science Park (ID: 1718165260)*
To *Nanliao Fighing 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: 11241
```



DFS with 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: 3291
```

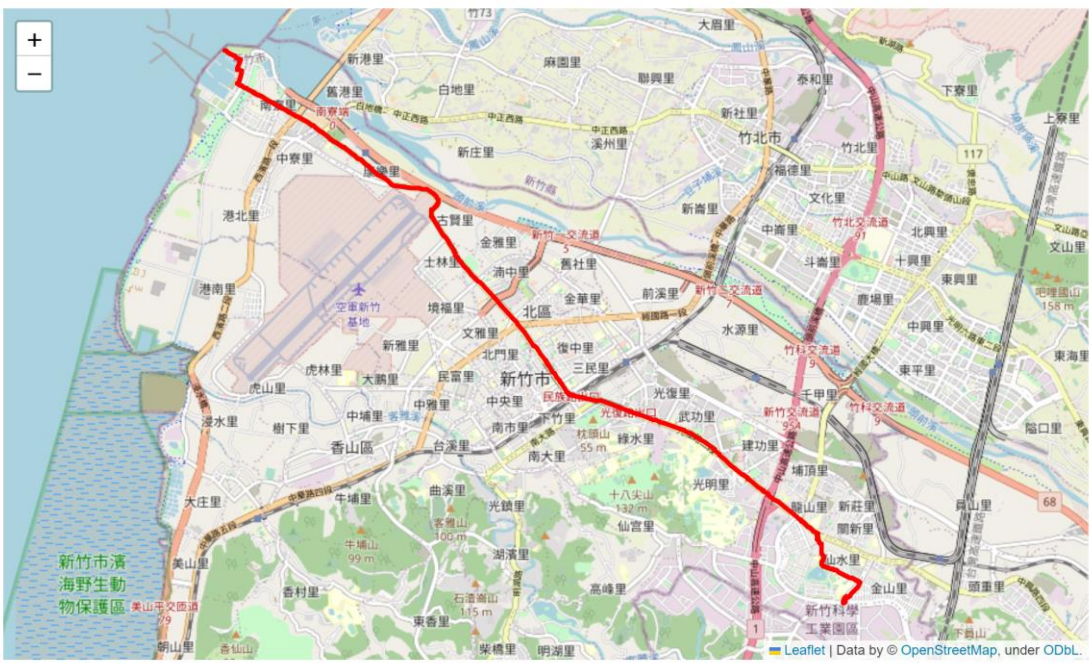
交大資工 110550062
朱致伶

Uniform Cost Search:

```
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* Search:

```
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
```



交大資工 110550062
朱致伶

Bonus(Part 6):

- **<u>A* Time Search:</u>**

The search orders by sum of the time cost $t(x)$ and the goal proximity $h(x)$. The time cost for each neighbor is the current node time add the neighbor distance divided by the speed limit of the road. Heuristic function is set as the straight-line distance divided by the highest speed limit of any road. The proposed heuristic function is admissible, since it never overestimates the true cost to goal.

```python
import csv
from queue import PriorityQueue;
from collections import defaultdict;
edgeFile = 'edges.csv'
heuristicFile = 'heuristic.csv'

def astar_time(start, end):
    # Begin your code (Part 6)
    max_speed = 0
    with open("edges.csv", 'r') as edgeFile, open('heuristic.csv', 'r') as heuristicFile:
        edge_data = csv.reader(edgeFile)
        heuristic_data = csv.reader(heuristicFile)
        # skip the first line(header) of the data
        next(edge_data)
        next(heuristic_data)
        # declare graph and h_func as defaultdict(list) to store nodes and heuristic function of each nodes
        graph = defaultdict(list)
        h_func = defaultdict(list)
        for e_row in edge_data:
            # store the node and corresponding information
            # transform data type to int or float, since the original data type is string
            # e_row[0]:start, e_row[1]:end, e_row[2]:distance, e_row[3]:speed limit
            graph[int(e_row[0])].append((int(e_row[1]), float(e_row[2]), float(e_row[3])))
            # record the max_speed_limit of all roads
            max_speed = float(e_row[3]) if (float(e_row[3])>max_speed) else max_speed
        for h_row in heuristic_data:
            # transform data type to int or float, since the original data type is string
            # h_row[0]:node_ID
            # h_row[1]:straight-line distance from node to Big City Shopping Mall(ID: 1079387396)
            # h_row[2]:straight-line distance from node to COSTCO Hsinchu Store(ID: 1737223506)
            # h_row[3]:straight-line distance from node to Nanliao Fighing Port(ID: 8513026827)
            h_func[int(h_row[0])].append((float(h_row[1]), float(h_row[2]), float(h_row[3])))
    # determine which column of h_func should be used with the end_node
    if end==1079387396:
        dest = 0
    elif end==1737223506:
        dest = 1
    elif end==8513026827:
        dest = 2
    # initialize visited set and priority queue
    pqueue = PriorityQueue()
    pqueue.put((float(h_func[start][0][dest])/max_speed, 0, start, [start]))
    visited = set()
    while not pqueue.empty():
        # tuple in priority_queue stores (goalproximity_of_time, time_from_start_to_current, current_node, current_path_list)
        # Since priority_queue sorts by the first elements of the tuple, heuristic_of_time is set as the first element
        (ht, time, cur, path) = pqueue.get()
        if cur not in visited:
            # mark current node as visited
            visited.add(cur)
            # if reach end_node, return data
            if  cur == end:
                return path, time, len(visited)
            # traverse through neighbors of current_node
            for neighbor, neighbor_distance, neighbor_speed_limit in graph[cur]:
                if neighbor not in visited:
                    # store neighbor_node and its information in priority_queue, traverse its neighbors later
                    # ht is the neighbor's update time + heristic_time of neighbor
                    neighbor_time = (neighbor_distance/neighbor_speed_limit)*3600/1000
                    straight_line = float(h_func[neighbor][0][dest])
                    heuristic_time = straight_line/max_speed
                    pqueue.put((time+neighbor_time+heuristic_time, time+neighbor_time, neighbor, path+[neighbor]))
    return [], 0, 0
    # End your code (Part 6)
```

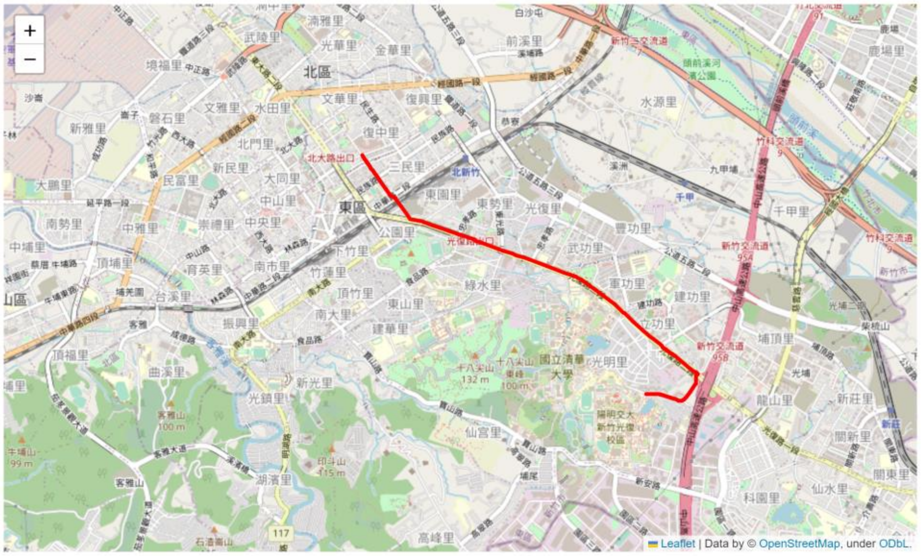
交大資工 110550062
朱致伶

- **Test 1:**

From *National Yang Ming Chiao Tung University (ID: 2270143902)*
To *Big City Shopping Mall (ID: 1079387396)*

<u>A* Time Search:</u>

```
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: 4444
```

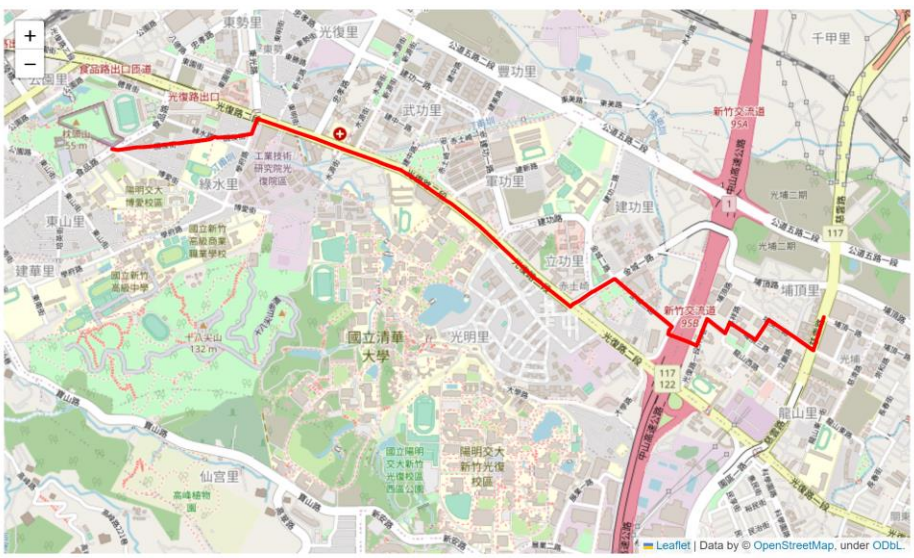


- **Test 2:**

From *Hsinchu Zoo (ID: 426882161)*
To COSTCO Hsinchu Store (ID: 1737223506)

<u>A* Time Search:</u>

```
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: 5895
```



交大資工 110550062
朱致伶

- **Test 3:**

From *National Experimental High School At Hsinchu Science Park (ID: 1718165260)*
To Nanliao Fighing Port (ID: 8513026827)

<u>A* Time Search:</u>

```
The number of nodes in the path found by A* search: 209
Total second of path found by A* search: 779.5279228368481 s
The number of visited nodes in A* search: 11372
```

交大資工 110550062
朱致伶

Part III. Question Answering (12%):

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

   While defining the heuristic function of A* time search, I first left it unchanged, which is the straight line distance to the end point. The result was not ideal since it has nothing to do with the time cost.

   Therefore, I changed the heuristic function to the neighbor distance divided by the speed limit of road. However, the heuristic function should present a global state, not current local state.

   Finally, the heuristic function was set as the straight-line distance to end point divided by the highest speed limit of any road. It presents a global state and is admissible, which underestimate the cost.

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 rationally.**

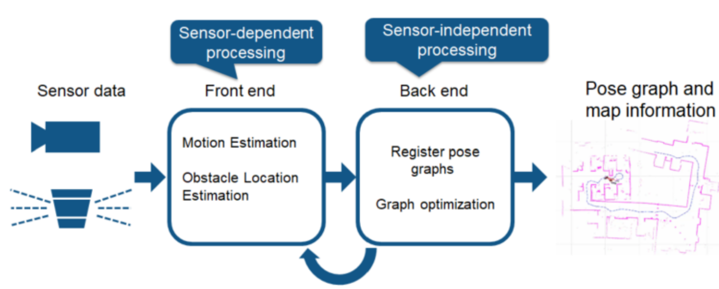
   Real-time road conditioning is also essential for route finding in the real world. Take going to work and getting off work as example, most people go to work at the range of 7 to 9 am and get off work at the range of 6 to 8 pm. During the interval time, traffic jams take place at the only way or must-passed intersection, as a result drivers spend more time to pass the jammed road than the original time, which is distance divided by speed limit of the road.

   To avoid the above problem, drivers are recommended to drive through the path may not be the shortest, but the fastest path considering the real-time road condition. The optimal path under this circumstance may exclude the road often jammed during commute time, hence include the path with longer distance but less traffic.

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 the mapping and localization components, a possible solution to obtain data is to use SLAM (Simultaneous Localization and Mapping) algorithms to create a map of the environment as the robot navigates through the area. This involves using various sensors such as LIDAR, cameras, and IMU data to construct a map that is continually updated as the robot moves.

   With the mapping data, define a set of vertices which are landmarks in real-world. With the distance between vertices and the relative position of vertices, we can build a graph storing nodes and corresponding information of the node to implement route finding.



How SLAM work

交大資工 110550062
朱致伶

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.

The dynamic heuristic function can be defined in two types:

<u>Before the meal is made:</u>

h(x) = meal prepare time + multiple order + delivery priority * (straight line distance / max speed limit)

<u>After the meal made, while delivering:</u>

h(x) = delivery priority * (straight line distance / max speed limit)

The parameters:

- **Meal Prepare Time**: the prepare time of different types of food varies a lot, set by the restaurant, only needed to be considered before start delivery.
- **Multiple Order**: a waiting interval from the first order ready to be delivered to the last order, set by the Uber Eats platform.
  Only needed to be considered before start delivery and having multiple order with the same destination.
  If the restaurant of Uber Eats platform has multiple order with the same destination, there will be a waiting interval time. From the time of the first order made to five minutes before the first meal was ready to be delivered, any order created in the interval time should be delivered by the same driver.
- **Delivery Priority:** a value between 1 to 2, the meal with higher priority should have value near 1. It is determined by the type of meal, and the user's additional requirement.