HW 2: Route Finding

Student ID: 111550100 Name: 邱振源

Part I. Implementation

Part 1:

```
"""
Part I: Read the 'edges.csv'
1. Create a dictionary named 'edges' to store the graph.
2. Use csv.reader() to read the whole file.
3. Since the first row is the column, use next() to skip the first one.
4. Since every node should be a key in the dictionary, for every node in the file,
   create a key in the 'edges' for it.
5. Store each path which (key = start ID) and (value = (end ID, distince)) in 'edges'.
"""

edges = {}
with open(edgeFile) as file:
    csvfile = csv.reader(file)
    next(csvfile)
    for row in csvfile:
        first_node = int(row[0])
        second_node = int(row[1])
        distance = float(row[2])
        if (first_node not in edges):
            edges[first_node] = []
        if (second_node not in edges):
            edges[second_node] = []
        edges[first_node].append((second_node, distance))
```

```
"""
Part II: BFS
1. Create some variables:
    (a) path_visited: a dictionary for storing the distance between visited nodes.
    (b) queue: a list storing pair for the neighbor and its distance to current nodes.
    (c) visited: a set storing the nodes which have been visited.
    (d) num_visited: record the number of visited node for return.
2. Pop the first element of 'queue'.
3. If the node pop from the queue is the endpoint of whole function, compute the path.
   If not, plus one to the 'num_visited', and traverse its neighbors. If its neighbor isn't in
   the 'visited', i.e, hasn't been visited, add this node and its distance to 'queue', add this
   node to 'visited', and store this node and its distance to 'path_visited'.
4. Compute the path:
    (a) Create some variables:
        (i)  path: a list storing the nodes on the path.
        (ii) dist: count the total distance of the path.
    (b) Add the 'end' to the 'path', since I want to compute path from the end point by 'path_visited'.
    (c) Repeat the below step until the fist element of 'path' is the start point:
        (i)  Use 'dist' to plus the distance between current node and its neighbor on the path.
        (ii) Insert the neighbor of the current to the front of 'path'.
    (d) Return 'path', 'dist', and 'num_visited'.
5. If bfs doesn't find the path successfully, return empty path, infinite, 'num_visited'.
"""
path_visited = {}
queue = [(start, 0)]
visited = set()
visited.add(start)
num_visited = 0
while queue:
    node = queue.pop(0)
    if node[0] == end:
        path = []
        dist = 0
        path.append(end)
        while path[0] != start:
            dist = dist + path_visited[path[0]][1]
            path.insert(0, path_visited[path[0]][0])
        return path, dist, num_visited
    else:
        num_visited += 1
        for end_node, dis in edges[node[0]]:
            if end_node not in visited:
                queue.append((end_node, dis))
                visited.add(end_node)
                path_visited[end_node] = (node[0], dis)
return [], float('inf'), num_visited
```

Part 2:

```
"""
Part I: Read the 'edges.csv'
1. Create a dictionary named 'edges' to store the graph.
2. Use csv.reader() to read the whole file.
3. Since the first row is the column, use next() to skip the first one.
4. Since every node should be a key in the dictionary, for every node in the file,
   create a key in the 'edges' for it.
5. Store each path which (key = start ID) and (value = (end ID, distince)) in 'edges'.
"""

edges = {}
with open(edgeFile) as file:
    csvfile = csv.reader(file)
    next(csvfile)
    for row in csvfile:
        first_node = int(row[0])
        second_node = int(row[1])
        distance = float(row[2])
        if (first_node not in edges):
            edges[first_node] = []
        if (second_node not in edges):
            edges[second_node] = []
        edges[first_node].append((second_node, distance))
```

```
"""
Part II: DFS
1. Create some variables:
    (a) path_visited: a dictionary for storing the distance between visited nodes.
    (b) stack: a list storing pair for the neighbor and its distance to current nodes.
    (c) visited: a set storing the nodes which have been visited.
    (d) num_visited: record the number of visited node for return.
2. Pop the last element of 'stack'.
3. If the node pop from the stack is the endpoint of whole function, compute the path.
   If not, plus one to the 'num_visited', and traverse its neighbors. If its neighbor isn't in
   the 'visited', i.e, hasn't been visited, add this node and its distance to 'stack', add this
   node to 'visited', and store this node and its distance to 'path_visited'.
4. Compute the path:
    (a) Create some variables:
        (i)  path: a list storing the nodes on the path.
        (ii) dist: count the total distance of the path.
    (b) Add the 'end' to the 'path', since I want to compute path from the end point by 'path_visited'.
    (c) Repeat the below step until the fist element of 'path' is the start point:
        (i)  Use 'dist' to plus the distance between current node and its neighbor on the path.
        (ii) Insert the neighbor of the current to the front of 'path'.
    (d) Return 'path', 'dist', and 'num_visited'.
5. If dfs doesn't find the path successfully, return empty path, infinite, 'num_visited'.
"""
path_visited = {}
stack = [(start, 0)]
visited = set()
visited.add(start)
num_visited = 0
while stack:
    node = stack.pop()
    if node[0] == end:
        path = []
        dist = 0
        path.append(end)
        while path[0] != start:
            dist = dist + path_visited[path[0]][1]
            path.insert(0, path_visited[path[0]][0])
        return path, dist, num_visited
    else:
        num_visited += 1
        for end_node, dis in edges[node[0]]:
            if end_node not in visited:
                stack.append((end_node, dis))
                visited.add(end_node)
                path_visited[end_node] = (node[0], dis)
return [], float('inf'), num_visited
```

Part 3:

```
# Begin your code (Part 3)
"""
Part I: Read the 'edges.csv'
1. Create a dictionary named 'edges' to store the graph.
2. Use csv.reader() to read the whole file.
3. Since the first row is the column, use next() to skip the first one.
4. Since every node should be a key in the dictionary, for every node in the file,
   create a key in the 'edges' for it.
5. Store each path which (key = start ID) and (value = (end ID, distance)) in 'edges'.
"""

edges = {}
with open(edgeFile) as file:
    csvfile = csv.reader(file)
    next(csvfile)
    for row in csvfile:
        first_node = int(row[0])
        second_node = int(row[1])
        distance = float(row[2])
        if first_node not in edges:
            edges[first_node] = []
        if second_node not in edges:
            edges[second_node] = []
        edges[first_node].append((second_node, distance))
```

```
"""
Part II: UCS
1. Create some variables:
    (a) path_visited: a dictionary for storing the distance between visited nodes.
    (b) priority_queue: a heap storing tuples of (cost, node) to prioritize exploring nodes.
    (c) visited: a set storing the nodes which have been visited.
    (d) num_visited: record the number of visited node for return.
2. Push the start node with cost 0 to the 'priority_queue'.
3. While the 'priority_queue' is not empty, pop the node with the least cost.
4. If the popped node is the endpoint, compute the path.
   If not, traverse its neighbors. If its neighbor isn't in the 'visited', i.e., hasn't been
   visited, add this node and its distance to 'priority_queue', add this node to 'visited',
   and store this node and its distance to 'path_visited'.
5. Compute the path:
    (a) Create some variables:
        (i)  path: a list storing the nodes on the path.
        (ii) dist: equal to the cost of the end point.
    (b) Add the 'end' to the 'path', since I want to compute path from the end point by 'path_visited'.
    (c) Repeat the below step until the first element of 'path' is the start point:
        (i)  Insert the neighbor of the current to the front of 'path'.
    (d) Return 'path', 'dist', and 'num_visited'.
6. If UCS doesn't find the path successfully, return empty path, infinite, 'num_visited'.
"""
path_visited = {}
priority_queue = [(0, start)]
visited = set()
num_visited = 0
while priority_queue:
    cost, node = heapq.heappop(priority_queue)
    if node == end:
        path = []
        dist = cost
        path.append(end)
        while path[0] != start:
            path.insert(0, path_visited[path[0]][0])
        return path, dist, num_visited
    if node not in visited:
        num_visited += 1
        visited.add(node)
        for end_node, dis in edges[node]:
            if end_node not in visited:
                heapq.heappush(priority_queue, (cost + dis, end_node))
                path_visited[end_node] = (node, dis)
return [], float('inf'), num_visited
# End your code (Part 3)
```

Part 4:

```
# Begin your code (Part 4)
"""
Part I: Read the 'edges.csv'
1. Create a dictionary named 'edges' to store the graph.
2. Use csv.reader() to read the whole file.
3. Since the first row is the column, use next() to skip the first one.
4. Since every node should be a key in the dictionary, for every node in the file,
   create a key in the 'edges' for it.
5. Store each path which (key = start ID) and (value = (end ID, distance)) in 'edges'.
"""
edges = {}
with open(edgeFile) as file:
    csvfile = csv.reader(file)
    next(csvfile)
    for row in csvfile:
        first_node = int(row[0])
        second_node = int(row[1])
        distance = float(row[2])
        if first_node not in edges:
            edges[first_node] = []
        if second_node not in edges:
            edges[second_node] = []
        edges[first_node].append((second_node, distance))
"""
Part II: Read the 'heuristic.csv'
1. Create a dictionary named 'heuristic' to store the file.
2. Use csv.reader() to read the whole file.
3. Use next() to read the header and skip the first row.
4. Use .index() to know the column which correspond to current 'end'.
5. Store each path which (key = node) and (value = h(node) to each 'end') in 'heuristic'.
"""
heuristic = {}
with open(heuristicFile) as file:
    csvfile = csv.reader(file)
    headers = next(csvfile)
    which_column = headers.index(str(end))
    for row in csvfile:
        heuristic[int(row[0])] = float(row[which_column])
```

```
"""
Part III: A*
1. Create some variables:
    (a) path_visited: a dictionary for storing the distance between visited nodes.
    (b) priority_queue: a heap storing tuples of (h(node), node) to prioritize exploring nodes.
    (c) visited: a set storing the nodes which have been visited.
    (d) num_visited: record the number of visited node for return.
2. Push the start node with its heuristic estimate of distance value to the 'priority_queue'.
3. While the 'priority_queue' is not empty, pop the node with the least cost.
4. If the popped node is the endpoint, compute the path.
   If not, traverse its neighbors. If its neighbor isn't in the 'visited', i.e., hasn't been
   visited, add this node and its new cost from heuristic function to 'priority_queue',
   add this node to 'visited', and store this node and its distance to 'path_visited'.
5. Compute the path:
    (a) Create some variables:
        (i)  path: a list storing the nodes on the path.
        (ii) dist: equal to the cost - h(node) of the end point.
    (b) Add the 'end' to the 'path', since I want to compute path from the end point by 'path_visited'.
    (c) Repeat the below step until the first element of 'path' is the start point:
        (i)  Insert the neighbor of the current to the front of 'path'.
    (d) Return 'path', 'dist', and 'num_visited'.
6. If A* search doesn't find the path successfully, return empty path, infinite, 'num_visited'.
"""
path_visited = {}
priority_queue = [(heuristic[start], start)]
visited = set()
num_visited = 0
while priority_queue:
    cost, node = heapq.heappop(priority_queue)
    if node == end:
        path = []
        dist = cost - heuristic[node]
        path.append(end)
        while path[0] != start:
            path.insert(0, path_visited[path[0]][0])
        return path, dist, num_visited
    if node not in visited:
        num_visited += 1
        visited.add(node)
        for end_node, dis in edges[node]:
            if end_node not in visited:
                new_cost = cost - heuristic[node] + dis + heuristic[end_node]
                heapq.heappush(priority_queue, (new_cost, end_node))
                path_visited[end_node] = (node, dis)
return [], float('inf'), num_visited
# End your code (Part 4)
```

Part 6:

```
# Begin your code (Part 6)
"""
Part I: Read the 'edges.csv'
1. Create a dictionary named 'edges' to store the graph.
2. Use csv.reader() to read the whole file.
3. Since the first row is the column, use next() to skip the first one.
4*.Since I need to return in the unit second, so change the speed limit to m/s. (line 29)
5*.Record the max speed limit in order to compute the heuristic function. (line 30)
6. Since every node should be a key in the dictionary, for every node in the file,
   create a key in the 'edges' for it.
7. Store each path which (key = start ID) and (value = (end ID, distance / speed_limit)) in 'edges'.
"""
edges = {}
max_speed = 0
with open(edgeFile) as file:
    csvfile = csv.reader(file)
    next(csvfile)
    for row in csvfile:
        first_node = int(row[0])
        second_node = int(row[1])
        distance = float(row[2])
        speed_limit = float(row[3]) * 1000 / (60 * 60)
        max_speed = max(max_speed, speed_limit)
        if first_node not in edges:
            edges[first_node] = []
        if second_node not in edges:
            edges[second_node] = []
        edges[first_node].append((second_node, float(distance / speed_limit)))
"""
Part II: Read the 'heuristic.csv'
1. Create a dictionary named 'heuristic' to store the file.
2. Use csv.reader() to read the whole file.
3. Use next() to read the header and skip the first row.
4. Use .index() to know the column which correspond to current 'end'.
5*.Store each path which (key = node) and (value = h(node) to each 'end') in 'heuristic'.
   (here the value of heuristic function is the (original value) / (max_speed))
"""
heuristic = {}
with open(heuristicFile) as file:
    csvfile = csv.reader(file)
    headers = next(csvfile)
    which_column = headers.index(str(end))
    for row in csvfile:
        heuristic[int(row[0])] = float(float(row[which_column]) / max_speed)
```

```
"""
Part III: A* time
1. Create some variables:
    (a) path_visited: a dictionary for storing the distance between visited nodes.
    (b) priority_queue: a heap storing tuples of (h(node), node) to prioritize exploring nodes.
    (c) visited: a set storing the nodes which have been visited.
    (d) num_visited: record the number of visited node for return.
2. Push the start node with its heuristic estimate of distance value to the 'priority_queue'.
3. While the 'priority_queue' is not empty, pop the node with the least cost.
4. If the popped node is the endpoint, compute the path.
   If not, traverse its neighbors. If its neighbor isn't in the 'visited', i.e., hasn't been
   visited, add this node and its new cost from heuristic function to 'priority_queue',
   add this node to 'visited', and store this node and its distance to 'path_visited'.
5. Compute the path:
    (a) Create some variables:
        (i)  path: a list storing the nodes on the path.
        (ii) time: equal to the cost - h(node) of the end point.
    (b) Add the 'end' to the 'path', since I want to compute path from the end point by 'path_visited'.
    (c) Repeat the below step until the first element of 'path' is the start point:
        (i)  Insert the neighbor of the current to the front of 'path'.
    (d) Return 'path', 'time', and 'num_visited'.
6. If A* search time doesn't find the path successfully, return empty path, infinite, 'num_visited'.
"""
path_visited = {}
priority_queue = [(heuristic[start], start)]
visited = set()
num_visited = 0
while priority_queue:
    cost, node = heapq.heappop(priority_queue)
    if node == end:
        path = []
        time = cost - heuristic[node]
        path.append(end)
        while path[0] != start:
            path.insert(0, path_visited[path[0]][0])
        return path, time, num_visited
    if node not in visited:
        num_visited += 1
        visited.add(node)
        for end_node, cos in edges[node]:
            if end_node not in visited:
                new_cost = cost - heuristic[node] + cos + heuristic[end_node]
                heapq.heappush(priority_queue, (new_cost, end_node))
                path_visited[end_node] = (node, cos)
return [], float('inf'), num_visited
# End your code (Part 6)
```

Part II. Results & Analysis

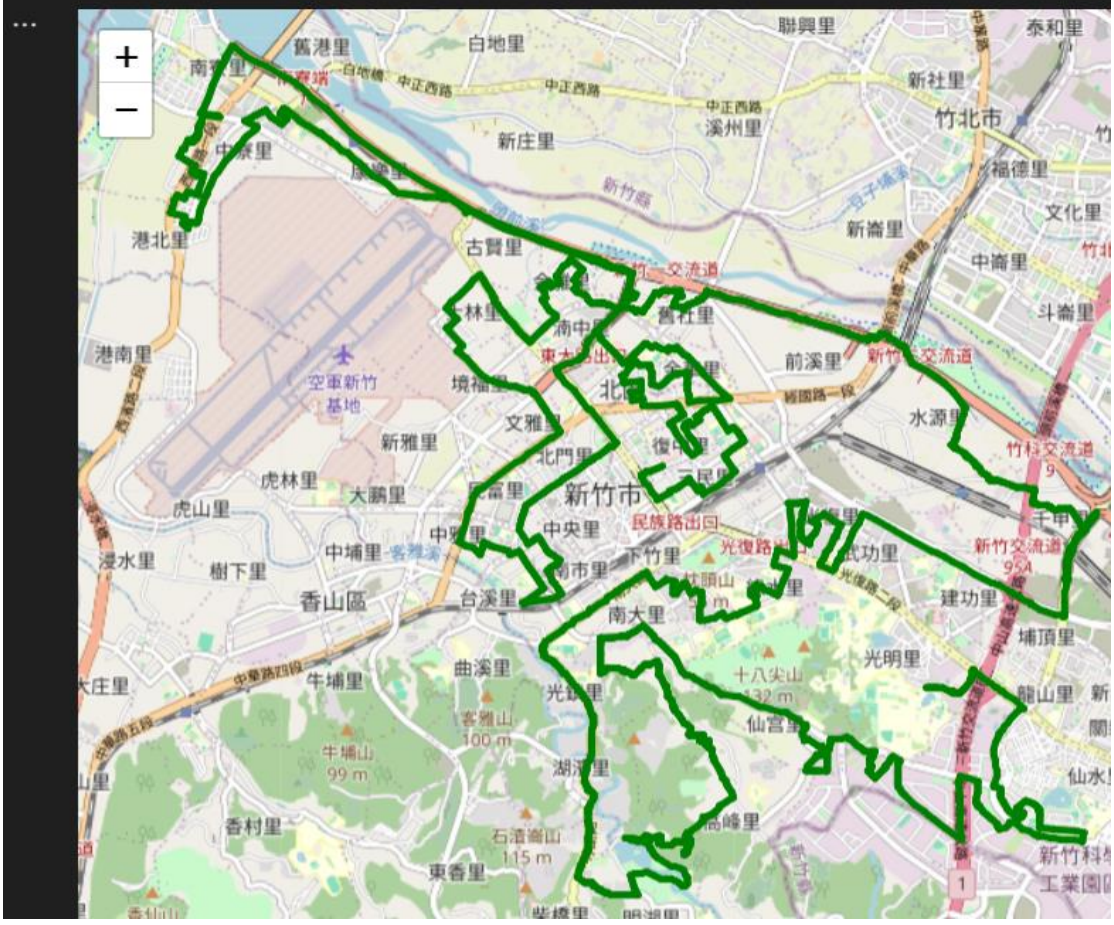
Test 1: National Yang Ming Chiao Tung University (ID: 2270143902) to Big City Shopping Mall (ID: 1079387396)

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: 4273
```

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: 4711
```



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: 5085
```

A* Search

```
The number of nodes in the path found by A* search: 89
Total distance of path found by A* search: 4367.880999999999 m
The number of visited nodes in A* search: 260
```

A* Search Time

```
The number of nodes in the path found by A* search: 89
Total second of path found by A* search: 320.8782316308316 s
The number of visited nodes in A* search: 1933
```

Test 2: 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: 4606
```

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: 9365
```

UCS

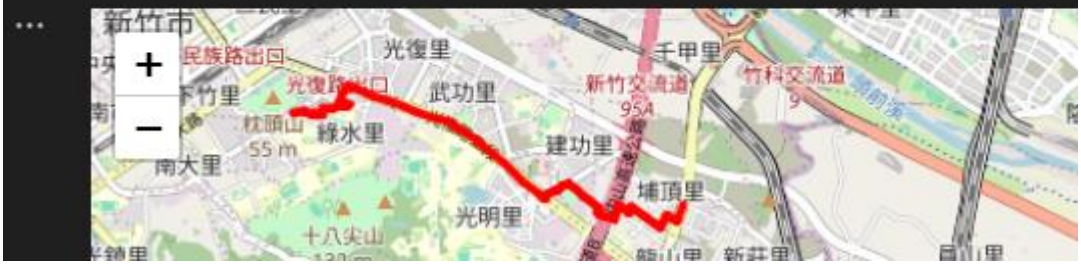
```
The number of nodes in the path found by UCS: 72
Total distance of path found by UCS: 4101.84 m
The number of visited nodes in UCS: 7212
```

A* Search

```
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: 1171
```

A* Search Time

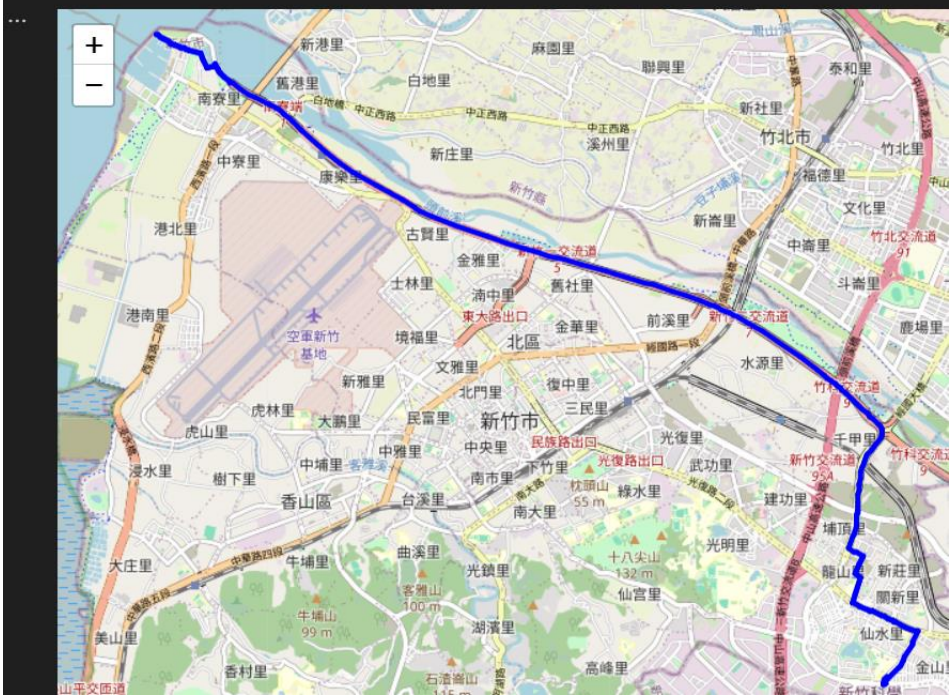
```
The number of nodes in the path found by A* search: 70
Total second of path found by A* search: 304.4436634360302 s
The number of visited nodes in A* search: 2869
```



Test 3: 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: 15442.394999999995 m
The number of visited nodes in BFS: 11241
```

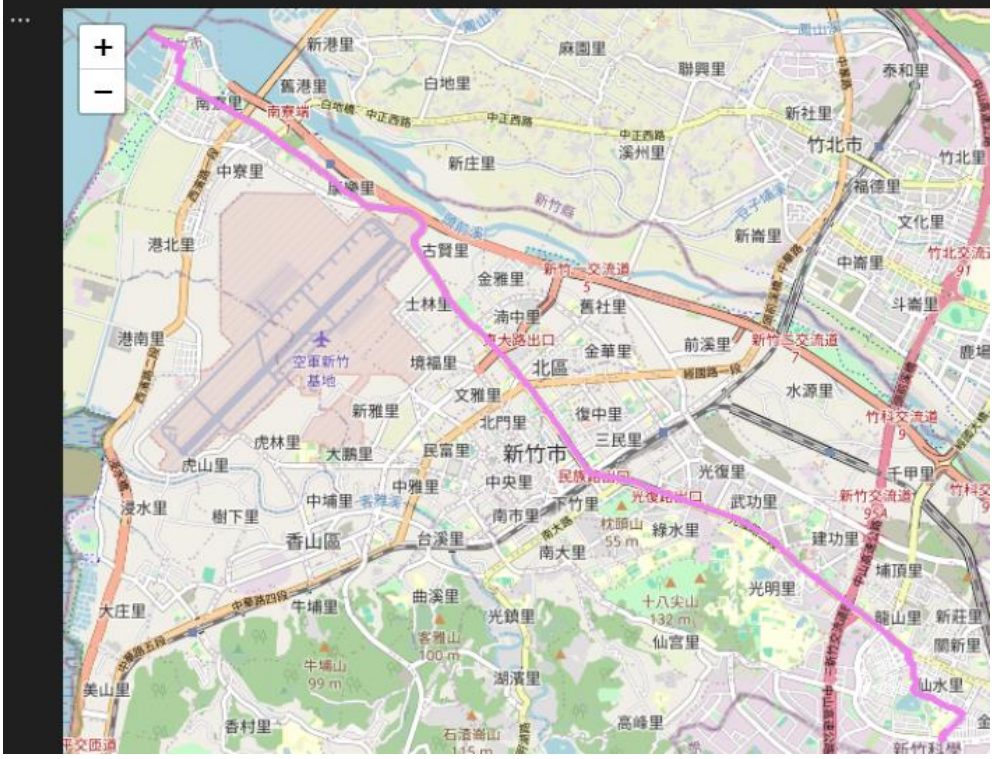


DFS (stack)

```
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: 2247
```

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: 11925
```



A* Search

```
The number of nodes in the path found by A* search: 288
Total distance of path found by A* search: 14212.413000000008 m
The number of visited nodes in A* search: 7072
```

A* Search Time

```
The number of nodes in the path found by A* search: 217
Total second of path found by A* search: 779.5279228368487 s
The number of visited nodes in A* search: 8457
```

Observing all the results above, it can be noticed that due to its pursuit of depth-first search (DFS), the paths found by this algorithm tend to have the longest distances. Consequently, its performance is comparatively poorer when searching for the shortest path. Next, in the case of breadth-first search (BFS), the paths obtained are close to the shortest path we desire. However, because BFS does not consider the prioritization of minimum cost during computation, its performance is somewhat inferior to Uniform Cost Search (UCS) and A* Search. UCS and A* Search are capable of obtaining the shortest paths. However, when computing longer routes, it becomes evident that UCS requires more execution time. Hence, A* Search emerges as a more ideal algorithm among the aforementioned algorithms.

When working on the bonus part, I stored the cost as the distance between the two points divided by the speed limit between those points, which yielded the time required to pass through those points. In defining the heuristic function, I also divided it by the speed limit to convert it into time units. However, because the value of the heuristic function needs to be smaller than the actual value, I divided it by the maximum speed limit in the .csv file. This ensures that for each route, the expected time in the heuristic function is less than or equal to the actual time.

Comparing the results of Part 4 A* Search and Part 6 A* Search (time), it can be observed that A* Search time does not necessarily choose the shortest path. For example, in test 3, A* Search (time) selected the orange-red roads along the way. Because highways or expressways have higher speed limits, even though their distance may be longer, they can be traversed in a shorter time. Hence, the results of Part 4 and Part 6 may differ.

Part III. Answer the questions

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

1. Originally, I intended to use nested dictionaries to store the distance relationships between two points. However, I found that using nested dictionaries made the code less smooth when implementing various algorithms. Therefore, I changed the way I stored the CSV file data to dictionaries encapsulating lists, which made the subsequent coding smoother.
2. Initially, I noticed a significant discrepancy between my DFS results and the example outputs. Consequently, I spent considerable time searching for errors within my code. However, after some contemplation, I realized that the order of points in DFS significantly influences the result of the path. This realization explained the significant difference between my results and the example.
3. When working on the bonus part, I initially had no idea how to define a new heuristic function. However, after discussing with classmates and reading through slides and searching online, I gained a clearer understanding of heuristic function definition. I also came up with the idea of dividing by the maximum speed limit in the entire file as a method to plan the heuristic function.

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.

1. Accessibility:
   Certain road conditions, such as closures or detours, may render certain routes inaccessible. Therefore, considering road condition information ensures that selected routes are accessible and feasible for travel.
2. Traffic Congestion
   Sometimes, traffic congestion can lead to increased travel time to reach the destination. Therefore, when selecting a route, being aware of the traffic conditions and considering their impact can optimize route selection.

3. **As mentioned in the introduction, a navigation system involves mapping, localization, and route finding. Please suggest possible solutions for mapping and localization components?**
    1. Mapping:
        (i) Lidar sensors mounted on vehicles or drones can capture detailed 3D maps of the environment, including buildings, roads, and terrain. Lidar data can be processed to create maps.
        (ii) Use satellite to take picture and get the satellite image of Earth. By processing it, we can create accurate maps for navigation systems.
    2. Localization:
        Utilizing GPS (Global Positioning System) technology for localization by receiving signals from satellites to determine the device's position on Earth.
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 consider some component in my equation:

    A. Meal Prep Time:
        This represents the time required for the restaurant to prepare the order. The higher the meal prep time, the longer the ETA.
    B. Distance to Delivery Location / Speed Limit:
        This factor accounts for the travel time of the delivery driver. The longer distance or the lower speed limit will cause the ETA become longer.
    C. Delivery Priority:
        Some orders may have higher priority due to factors like delivery time commitments, customer preferences, or special circumstances (e.g., urgent orders, VIP customers). Assigning a priority value to each order allows the system to adjust the ETA accordingly.
    D. Number of Orders:
        If a delivery driver is assigned multiple orders for simultaneous delivery (batching), the system needs to consider the total number of orders and their respective delivery locations. More orders generally increase the delivery time due to additional stops and route optimization requirements.

    Then I define my dynamic heuristic equation as:

$$\mathrm{ETA} = \mathrm{A} + \mathrm{B} \times \mathrm{C} \times \mathrm{D}$$

The dynamic heuristic equation incorporates various real-time factors that can affect delivery time, such as meal prep time, distance, priority, and order volume. As the meal prep time can be determined before the delivery driver departs, while the delivery priority, time to arrival, and number of orders can change dynamically during the delivery process, I multiply the latter three factors together and then add the meal prep time. By considering these factors dynamically, the system can adapt to changing conditions and provide more accurate ETA predictions, accomplish customization and improving user experience.