# Homework 2: Route Finding

**Please keep the title of each section and delete examples. Note that please keep the questions listed in Part III.**

## Part I. Implementation (6%):

- **Part 1**

```python
def bfs(start, end):
    # Begin your code (Part 1)
    """
    I utilize the `csv.reader` to parse the file, converting it into a list of rows of data.
    For each row, I construct a two-dimensional dictionary to store the edges.
    """
    """
    Implement BFS.
    I employ a queue and Initialize an empty dictionary called 'parent', a list with start ID called 'queue', and set to mark the visited node called 'visited', and 'num_visited' to 0
    Upon reaching the end node, the algorithm terminates.
    Leveraging the parent information, I retrieve the path from the start to the end node, along with the distance.

    *BFS Algorithm:
    1. Enqueue the start node into the queue.
    2. While the queue is not empty:
        - Dequeue the front element.
        - Enqueue its unvisited neighbors and mark them with their parent.
        - Continue until the queue is empty or the end node is detected.
    """

    edges = {}
    with open(edgeFile, newline='') as csvfile:
        rows = csv.reader(csvfile)
        next(rows)  # Skip header
        for row in rows:
            start_node, end_node = map(int, row[:2])
            distance = float(row[2])  # Convert distance to float
            if start_node not in edges:
                edges[start_node] = {}
            edges[start_node][end_node] = distance

    visited = set()
    q = Queue()
    q.put(start)
    visited.add(start)
    parent = {start: None}
    path = []
    dist = 0
    num_visited = 0

    while not q.empty():
        current = q.get()
        num_visited += 1
        if current == end:
            # Reconstruct the path
            node = end
            while node is not None:
                path.append(node)
                node = parent[node]
            path.reverse()
            # Compute distance
            for i in range(len(path) - 1):
                dist += edges[path[i]][path[i+1]]
            return path, dist, num_visited

        for neighbor, distance in edges.get(current, {}).items():
            if neighbor not in visited:
                visited.add(neighbor)
                parent[neighbor] = current
                q.put(neighbor)

    # If end node is not reachable
    return [], 0, num_visited
    # End your code (Part 1)
```

- **Part 2**

```python
def dfs(start, end):
    # Begin your code (Part 2)
    """
    I first use csv.reader to read the file and convert it into a list of rows of data. For each row of data, I store it in a two-dimensional dictionary representing the graph,
    where the keys are the starting nodes and the values are dictionaries containing the adjacent nodes and their distances.

    The DFS algorithm starts by pushing the start node onto the stack. Then, in each iteration, it pops the top element from the stack and explores its neighbors.
    If a neighbor has not been visited yet, it is pushed onto the top of the stack. This process continues until either the stack is empty or the algorithm detects the end node.

    Once the end node is detected, the algorithm finishes. Using the information stored in the parent dictionary,
    the algorithm reconstructs the path from the start node to the end node.
    Additionally, it computes the total distance of the path by summing up the distances between consecutive nodes along the path.

    Overall, the DFS algorithm explores the graph in a depth-first manner, prioritizing exploration of nodes that are deeper in the graph before backtracking.
    This can be useful for certain types of graph traversal problems, such as finding paths or cycles.
    """
    edges = {}
    with open(edgeFile, newline='') as csvfile:
        rows = csv.reader(csvfile)
        next(rows)  # Skip header
        for row in rows:
            start_node, end_node = map(int, row[:2])
            distance = float(row[2])  # Convert distance to float
            if start_node not in edges:
                edges[start_node] = {}
            edges[start_node][end_node] = distance

    visited = set()
    stack = [start]  # Stack to keep track of nodes to visit
    parent = {start: None}
    path = []
    dist = 0
    num_visited = 0

    while stack:
        current = stack.pop()  # Pop the top node from the stack
        num_visited += 1
        if current == end:
            # Reconstruct the path
            node = end
            while node is not None:
                path.append(node)
                node = parent[node]
            path.reverse()
            # Compute distance
            for i in range(len(path) - 1):
                dist += edges[path[i]][path[i+1]]
            return path, dist, num_visited

        visited.add(current)

        # Push unvisited neighbors onto the stack
        for neighbor, distance in edges.get(current, {}).items():
            if neighbor not in visited:
                stack.append(neighbor)
                parent[neighbor] = current

    # If end node is not reachable
    return [], 0, num_visited
    # End your code (Part 2)
```

- **Part 3**

```python
def ucs(start, end):
    # begin your code (Part 3)
    """
    I first use csv.reader to read the file and convert it into a list of rows of data.
    For each row of data, I store it in a two-dimensional dictionary representing the graph,
    where the keys are the starting nodes and the values are dictionaries containing the adjacent nodes and their distances.

    I use a priority_queue (heapq) to implement the Uniform Cost Search (UCS) algorithm,
    where each element in the queue consists of a tuple containing the distance from the start point, node ID, and ID of the node's parent.
    When the UCS algorithm detects the end node, it finishes.

    Using the information from the node's parent, I can reconstruct the path from the start to the end node,
    as well as calculate the total distance of the road.

    * Uniform Cost Search algorithm:
    First, I put the tuple (0, start node) into the priority_queue.
    For every iteration, I extract the most prioritized element (the closest) from the priority_queue,
    and push its neighbors (along with the cumulative distance) that haven't been visited into the priority queue,
    until the priority queue is empty or the end node is detected.
    Additionally, I use a dictionary "distances" to record the distance of nodes that are in the priority queue.
    If the distance of the newly explored node is larger than the previously explored distance,
    I do not add it to the priority queue.
    """
    edges = {}
    with open(edgeFile, newline='') as csvfile:
        rows = csv.reader(csvfile)
        next(rows)  # Skip header
        for row in rows:
            start_node, end_node = map(int, row[:2])
            distance = float(row[2])  # Convert distance to float
            if start_node not in edges:
                edges[start_node] = {}
            edges[start_node][end_node] = distance
    # Priority queue to store nodes based on cumulative distance
    priority_queue = [(0, start)]
    parent = {start: None}
    # Dictionary to store the cumulative distance for each node
    distances = {start: 0}
    path = []
    num_visited = 0
    while priority_queue:
        # Pop node with minimum cumulative distance
        cumulative_distance, current = heapq.heappop(priority_queue)
        num_visited += 1
        if current == end:
            # Reconstruct the path
            node = end
            while node is not None:
                path.append(node)
                node = parent[node]
            path.reverse()
            # Total distance is the cumulative distance to the end node
            dist = distances[end]
            return path, dist, num_visited

        # Explore neighbors of the current node
        for neighbor, distance in edges.get(current, {}).items():
            cumulative_distance_to_neighbor = cumulative_distance + distance
            if neighbor not in distances or cumulative_distance_to_neighbor < distances[neighbor]:
                distances[neighbor] = cumulative_distance_to_neighbor
                heapq.heappush(
                    priority_queue, (cumulative_distance_to_neighbor, neighbor))
                parent[neighbor] = current

    # If end node is not reachable
    return [], 0, num_visited
    # end your code (Part 3)
```

- **Part 4**

```python
def load_heuristic():
    """
    I use csv.reader to read the heuristic.csv file and convert it into a dictionary.
    For each row of data, I store the node ID and its heuristic value.
    The heuristic values represent the estimated cost from each node to the goal node.
    """
    heuristic = {}
    with open(heuristicFile, newline='') as csvfile:
        rows = csv.reader(csvfile)
        next(rows)  # Skip header
        for row in rows:
            node = int(row[0])
            h_value = float(row[1])
            heuristic[node] = h_value
    return heuristic
```

```python
def astar(start, end):
    # begin your code (Part4)
    """
    I first read the edge data from the edges.csv file and construct the graph.
    Then, I load the heuristic values from the heuristic.csv file using the load_heuristic function.
    I use a priority_queue to implement the A* algorithm, where each element contains the heuristic value,
    node ID, ID of the node's parent, and true distance from the start node.
    The heuristic value is calculated using the straight distance to the destination plus the true distance from the start point.
    For each iteration, I extract the most prioritized element (the one with the least heuristic value) from the priority_queue.
    I explore the neighbors of the current node, updating their true distances if a shorter path is found.
    I then calculate the new heuristic value for each neighbor and add it to the priority queue if it's not already visited.
    The algorithm continues until the priority queue is empty or the end node is detected.
    """
    edges = {}
    with open(edgeFile, newline='') as csvfile:
        rows = csv.reader(csvfile)
        next(rows)  # Skip header
        for row in rows:
            start_node, end_node = map(int, row[:2])
            distance = float(row[2])  # Convert distance to float
            if start_node not in edges:
                edges[start_node] = {}
            edges[start_node][end_node] = distance

    heuristic_values = load_heuristic()

    # Priority queue to store nodes based on cumulative distance + heuristic
    priority_queue = [(0, start)]
    parent = {start: None}
    # Dictionary to store the cumulative distance for each node
    distances = {start: 0}
    path = []
    num_visited = 0

    while priority_queue:
        # Pop node with minimum cumulative distance + heuristic
        _, current = heapq.heappop(priority_queue)
        num_visited += 1
        if current == end:
            # Reconstruct the path
            node = end
            while node is not None:
                path.append(node)
                node = parent[node]
            path.reverse()
            # Total distance is the cumulative distance to the end node
            dist = distances[end]
            return path, dist, num_visited

        # Explore neighbors of the current node
        for neighbor, distance in edges.get(current, {}).items():
            cumulative_distance_to_neighbor = distances[current] + distance
            # Update the distance if a shorter path to the neighbor is found
            if neighbor not in distances or cumulative_distance_to_neighbor < distances[neighbor]:
                distances[neighbor] = cumulative_distance_to_neighbor
                priority = cumulative_distance_to_neighbor + \
                    heuristic_values.get(neighbor, 0)
                heapq.heappush(priority_queue, (priority, neighbor))
                parent[neighbor] = current

    # If end node is not reachable
    return [], 0, num_visited
    # end your code (Part4)
```

- **Part 6**

```python
def astar_time(start, end):
    # begin your code (Part 6)
    """
    I first use csv.reader to read the files, and convert it to a list of rows of data. For each data, I store it into a two-dimensional dictionary,
    including edges (where I convert the speed from km/h to m/sec) and heuristic values.
    I use a priority_queue to implement the A* algorithm, and store every node's parent into "From". When the A* algorithm detects the end node, it finishes.
    Using the information from the node's parent, I can get the path from start to end, and the distance of the road.

    In the A* algorithm:
        - First, I put the (float(heur[start][end])/(60/3.6), start, -1, 0) into the priority_queue. The structure in the priority_queue is (heuristic value, nodeID, ID of the node's parent, true distance).
        The heuristic function is the straight distance to the destination divided by the average speed (converted to m/sec) plus the true time from the start point.
        - For every iteration, I get the most prioritized (the least heuristic value) element out and push its neighbors (heuristic value and distance) which haven't been detected into the priority queue until the priority queue is empty or the end node is detected.
        - Additionally, I add "dis" (a dictionary) to record the heuristic value of nodes which are in the priority queue. If the heuristic value of a newly explored node is larger than the previous explore, I do not add it into the priority queue.
        """
    # Load edge data from CSV
    edges = {}
    max_speed_limit = 0  # Initialize maximum speed limit
    with open(edge_file, newline='') as csvfile:
        reader = csv.reader(csvfile)
        next(reader)  # Skip header
        for row in reader:
            # Convert node IDs to integers
            start_node, end_node = map(int, row[:2])
            distance = float(row[2])
            speed_limit = float(row[3])
            # Convert speed limit from km/hr to m/s
            speed_limit = speed_limit * 1000 / 3600

            # Update maximum speed limit
            max_speed_limit = max(max_speed_limit, speed_limit)
            # Calculate edge cost based on speed limit
            edge_cost = distance / (speed_limit)
            edges.setdefault(start_node, {})[end_node] = edge_cost

    # Load heuristic values
    heuristic_values = load_heuristic()

    # Update heuristic values based on maximal speed limit
    for node, h_value in heuristic_values.items():
        # Convert speed limit from km/hr to m/s
        heuristic_values[node] = h_value / (max_speed_limit * 1000 / 3600)

    # Priority queue to store nodes based on cumulative distance + heuristic
    priority_queue = [(0, start)]
    parent = {start: None}
    # Dictionary to store the cumulative distance for each node
    distances = {start: 0}
    path = []
    num_visited = 0

    while priority_queue:
        # Pop node with minimum cumulative distance + heuristic
        _, current = heapq.heappop(priority_queue)
        num_visited += 1
        if current == end:
            # Reconstruct the path
            node = end
            while node is not None:
                path.append(node)
                node = parent[node]
            path.reverse()
            # Total time is the cumulative distance to the end node
            time = distances[end]
            return path, time, num_visited

        # Explore neighbors of the current node
        for neighbor, edge_cost in edges.get(current, {}).items():
            cumulative_distance_to_neighbor = distances[current] + edge_cost
            # Update the distance if a shorter path to the neighbor is found
            if neighbor not in distances or cumulative_distance_to_neighbor < distances[neighbor]:
                distances[neighbor] = cumulative_distance_to_neighbor
                priority = cumulative_distance_to_neighbor + \
                    heuristic_values.get(neighbor, 0)
                heapq.heappush(priority_queue, (priority, neighbor))
                parent[neighbor] = current

    # If end node is not reachable
    return [], 0, num_visited
    # end your code (Part 6)
```

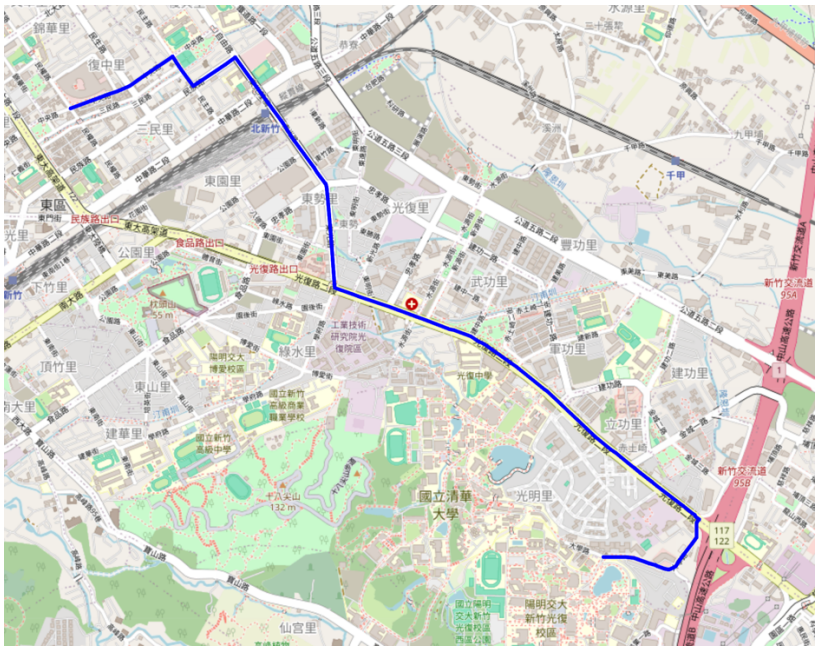
**Load heuristic function is same as Part 4**

## Part II. Results & Analysis (12%):

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

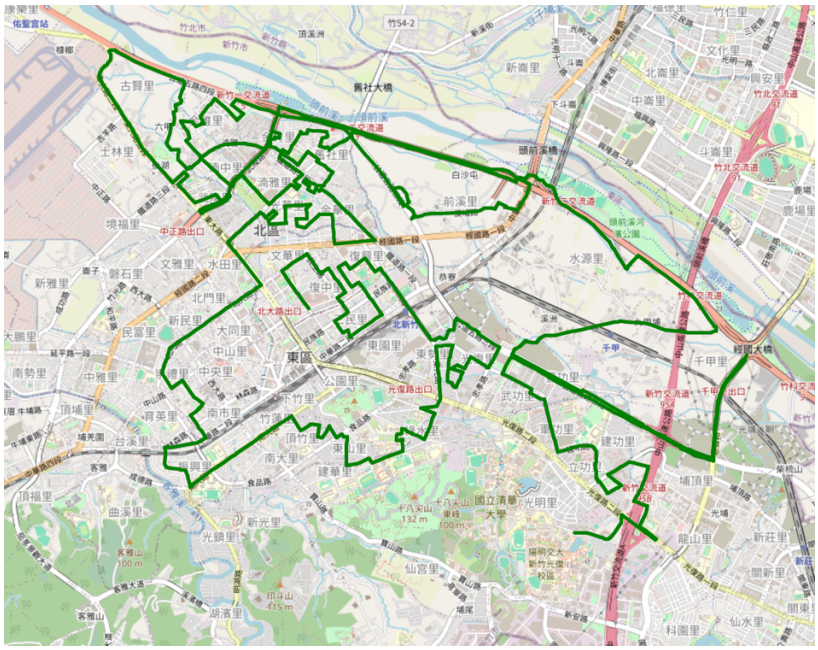
BFS:

```
The number of path nodes: 88
Total distance of path: 4978.8820000000005
The number of visited nodes: 4274
```



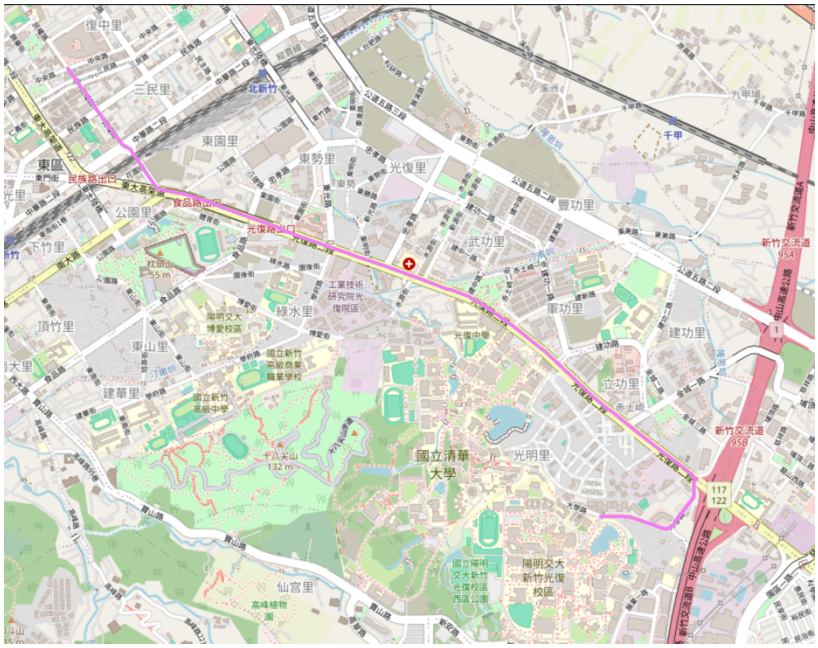

DFS (stack):

```
The number of path nodes: 1232
Total distance of path: 57208.987000000045
The number of visited nodes: 4381
```



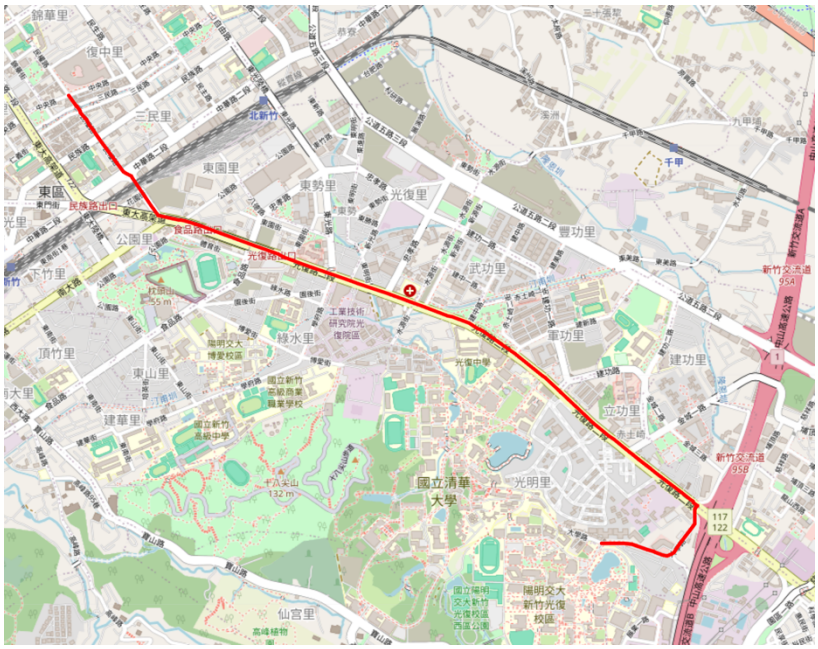
UCS:

```
The number of path nodes: 89
Total distance of path: 4367.881
The number of visited nodes: 5232
```



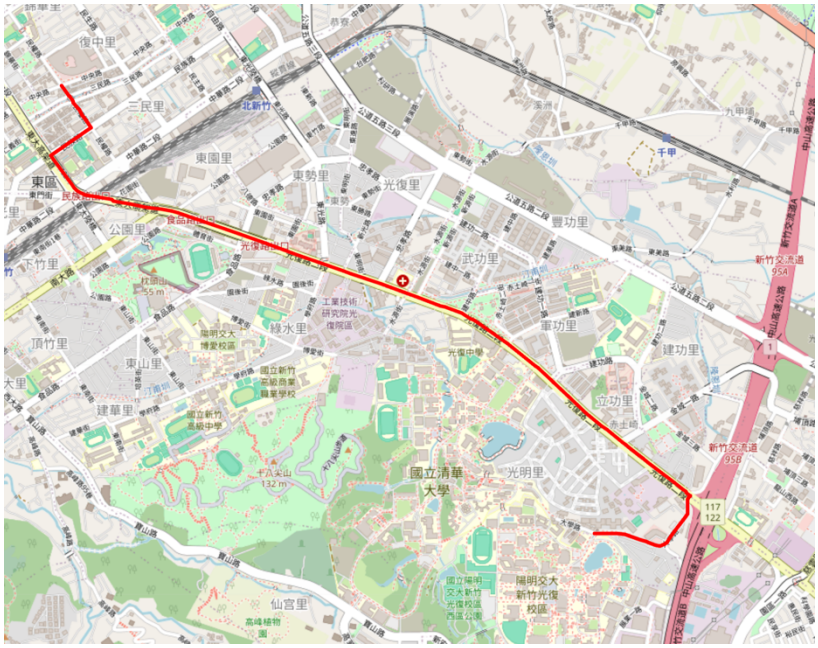

A*:

```
The number of path nodes: 89
Total distance of path: 4367.881
The number of visited nodes: 262
```



A*(time):

```
The number of path nodes: 93
Total second of path: 340.93138176229775
The number of visited nodes: 167
```



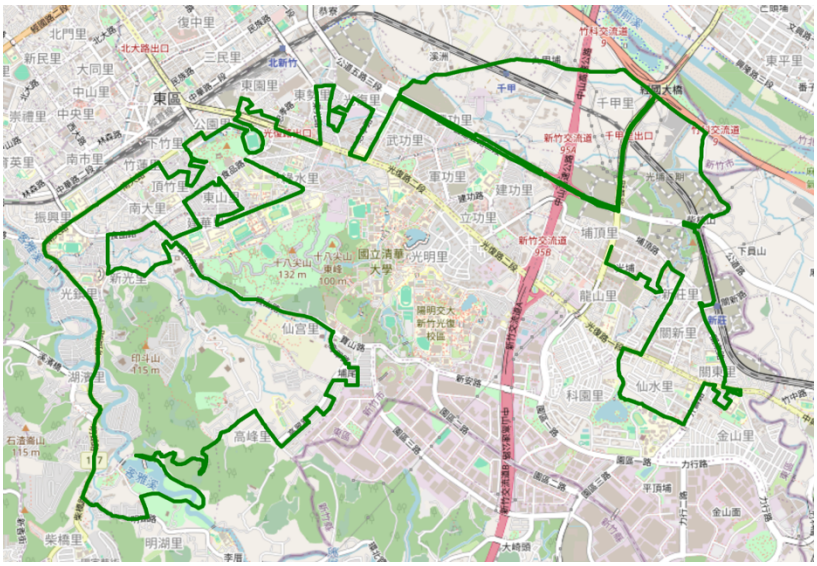

**Test 2: from Hsinchu Zoo (ID: 426882161) to COSTCO Hsinchu Store (ID: 1737223506)**

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
```

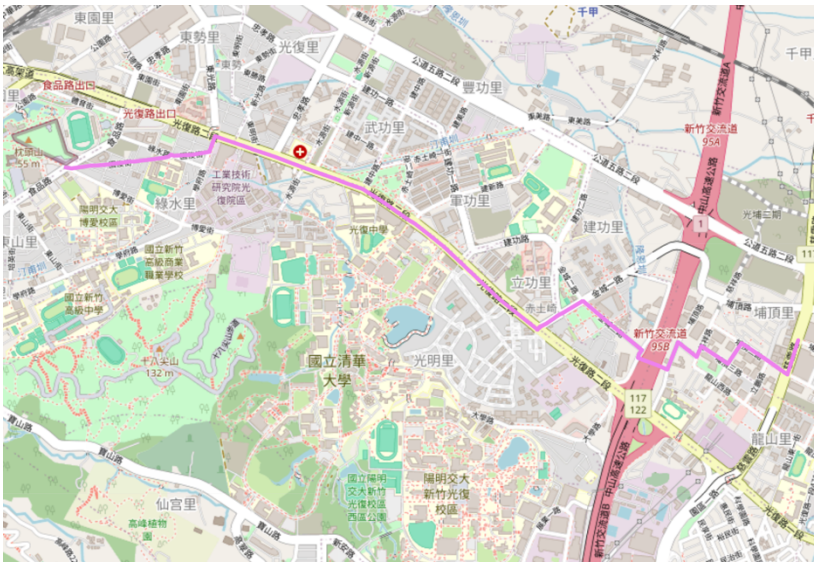
DFS (stack):

```
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: 8628
```



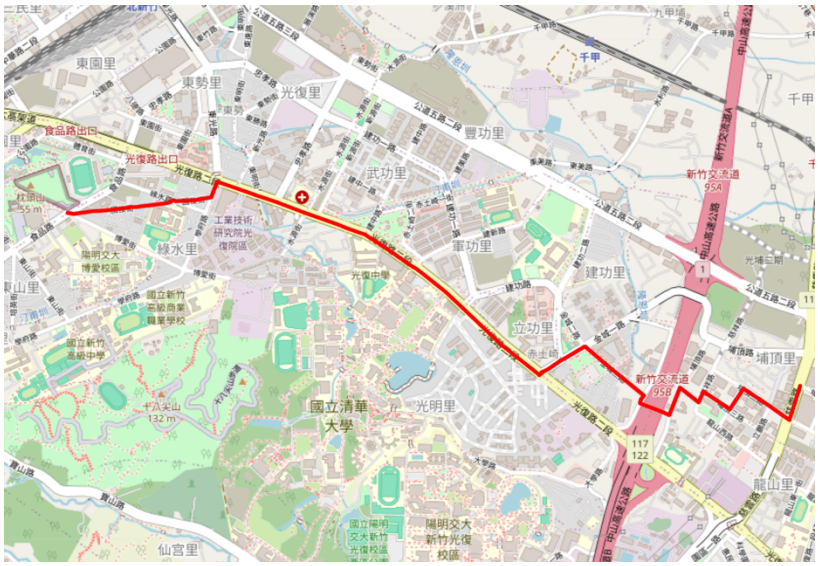

UCS:

```
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: 7454
```



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: 8187
```




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: 8479
```

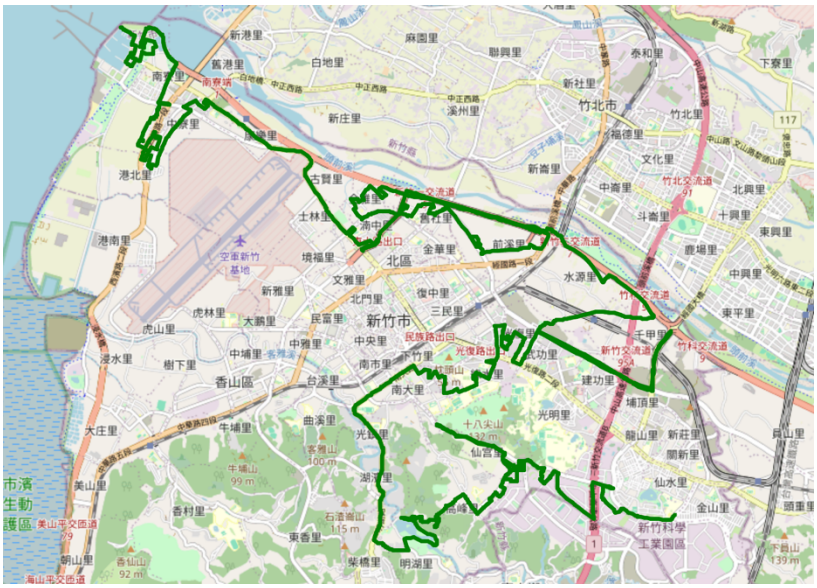
### Test 3: from National Experimental High School at Hsinchu Science Park (ID: 1718165260) to Nanliao Fighing Port (ID: 8513026827)

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: 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: 3371
```



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: 12312
```


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: 12582
```

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: 24385
```

## Part III. Question Answering (12%):

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

    - A little bit difference in DFS implementation will cause different result in route finding, for example, my DFS results are quite different from the given results.

    - In the beginning, I had no idea to deal with the data, since the data format isn't what I expected. It took time to think a good way for processing the data, but once the process was done, it became easier to start with.

    - Tracking path in Python took me for a while to think and design. Finally, I consider using dictionary is a more suitable way for record their parents.

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.

    - Incorporating real-time traffic flow data is essential for efficient route finding in real-world scenarios. Understanding the traffic flow of each road segment aids in determining the optimal route. High traffic flow often leads to congestion, causing delays and impacting travel schedules.
    - By integrating real-time traffic flow information into route-finding algorithms, users can access current traffic conditions, enabling them to adjust their routes and avoid potential delays. This ensures faster and more efficient travel, with a more accurate estimation of travel times.
    - In conclusion, integrating traffic flow data enhances navigation accuracy and reliability, empowering users to make informed decisions when planning their routes, ultimately leading to a smoother travel experience.

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 component, one possible solution is to use satellite imagery or aerial photography to create high-resolution maps. These maps can be generated using computer vision techniques, such as image segmentation and object detection, to identify and label different features, such as roads, buildings, and landmarks. The maps can also be updated in real-time using crowdsourcing or sensor data from vehicles or drones.

- For the localization component, one possible solution is to use GPS or other positioning technologies, such as Bluetooth beacons or Wi-Fi triangulation, to determine the location of the user or vehicle. However, GPS signals can be affected by interference or obstacles, such as tall buildings or tunnels, which can result in inaccurate or unreliable location estimates.

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.

   A dynamic heuristic equation for ETA in Uber Eats can be defined as follows:

   1. If there is no order on the nearest deliver

      **ETA = Meal preparation time + (Time to travel from the location of nearest deliver to restaurant) + (Time to travel from restaurant to customer)**

   2. If there is one or more order on the nearest deliver and the priority of this new order is higher than the previous one

      **ETA = Meal preparation time + Time to travel from the location of the nearest delivery point to the restaurant + Time to travel from the restaurant to the customer with the highest priority + Average interaction time between deliveries and customers + Time to travel from the customer with the highest priority to the second priority customer + Average interaction time between deliveries and customers, repeated for each subsequent customer until reaching the customer who placed the current order…**

      Considering that VIP customers always hold the highest priority, it's unnecessary to factor in the previous meal with lower priority. The logical flow a general delivery service would follow is as follows:

      1. Turn on the App and check the nearest order and the destination.
      2. Click accept the order and go to the restaurant.
      3. While waiting the meal, check whether there is an order that will have a short distance between the deliver and the restaurant, and a short distance between two destinations. If so, click accept.
      4. Deliver the meal to the customer.

      This might be a relatively easy logic flow. In the real world, the nearest deliver may not to accept the meal due to the time and the money. Or the deliver may not accept another order while waiting the meal due to some reasons.

When the nearest deliver has no orders, we can easily define the ETA.

- Meal preparation time: The amount of time it takes for the restaurant to prepare the order. This factor can vary depending on the type of cuisine, the complexity of the order, and the restaurant's current workload.
- Time to travel from the location of nearest deliver to restaurant: If we need to implement this, there are too many attributes we need to take into consideration. For example, the traffic light waiting time and the current traffic flow on each road. I think when Uber needs to implement this, they can simply get a google map API to know the possible travel time of the deliver to the restaurant.
- Time to travel from restaurant to customer: This is the same problem asTime to travel from the location of nearest deliver to restaurant".

However, if there are one or more orders on the deliver, I think we need to take the priority of each customer. For example, Uber Plus subscribers may have the highest priority, the normal user has the second high priority, and the user who may ordered the meal and didn't pay for it may have the lowest priority. I assume that it had been decided by the Uber's App.

  - Meal preparation time + Time to travel from the nearest delivery location to the restaurant: This mirrors the previous calculation.
  - Time to travel from the restaurant to the highest-priority customer: This resembles the previous "Time to travel from restaurant to customer" scenario.
  - Average interaction time between delivery drivers and customers: Considering Uber may possess data on this interaction, it's crucial for accurate estimations.
  - (Time to travel from the highest-priority customer to the second-priority customer) + Average interaction time between deliveries and customers + ... + (Time to travel from the previous customer to the customer who placed this order): This combines aspects of the previous two problems. If two meals have equal priority, prioritize the closer one to the delivery location.

Noted that meal preparation time may overlap with the time to travel from the location of nearest deliver to restaurant. In my view, the value of admissible heuristic must be smaller than the true cost to the nearest goal.