Homework 2: Route Finding

Part I. Implementation (6%):

PART1:

```python
def bfs(start, end):
    # Begin your code (Part 1)
    """
    Read the csv file and Iterate each
    row and create the graph
    """
    graph={}
    with open(edgeFile,mode='r') as file:
        reader=csv.reader(file)
        next(reader)
        for row in reader:
            source=int(row[0])
            target=int(row[1])
            distance=float(row[2])

            if source not in graph:
                graph[source]=[]

            graph[source].append((target,distance))
    """
    Initialize some datastructure to implement the bfs
    """
    queue=[(start,0)]
    visit={start:None}
    dist={start:0}
    num_visited=0
    """
    iterate queue and find update the information of node
    use visit to record the parents
    usr dist to caculate the distance to the neighbor
    add the neighbor to the queue and updata the distance
    """
    while queue:
```

```python
    cur_node,cur_dis=queue.pop(0)
    if cur_node==end:
        break
    num_visited += 1
    for neighbor,dis in graph.get(cur_node,[]):
        if neighbor not in visit:
            visit[neighbor]=cur_node
            dist[neighbor]=dis+cur_dis
            queue.append((neighbor,dis+cur_dis))
"""
Start with end node usr visit to construct the path
and reverse it and return path,total_dist,num_visited
"""
path=[]
current=end
while current is not None:
    path.append(current)
    current=visit[current]
path.reverse()
total_dist=dist[end]
num_visited=num_visited
return path,total_dist,num_visited
# End your code (Part 1)
```

PART2:

```python
def dfs(start, end):
    # Begin your code (Part 2)
    """
    Read the csv file and Iterate each
    row and create the graph
    """
    graph = {}
    with open(edgeFile, mode='r') as file:
        reader = csv.reader(file)
        next(reader)
        for row in reader:
            source = int(row[0])
            target = int(row[1])
            distance = float(row[2])

            if source not in graph:
                graph[source] = []

            graph[source].append((target, distance))
    """
    Initialize some datastructure to implement the dfs
    """
    stack = [(start, 0)]
    visit = {start: None}
    dist = {start: 0}
    num_visited = 0
    """
    Implement the DFS using a stack. Each element in the stack is a tuple containing
    the current node and the total distance traveled to reach that node.
    `visit` tracks the parent node of each visited node, enabling path reconstruction.
    `dist` keeps track of the shortest distance from the start node to each visited node.
    """
    while stack:
```

```python
        cur_node, cur_dis = stack.pop()
        if cur_node == end:
            break
        num_visited += 1
        for neighbor, dis in graph.get(cur_node, []):
            if neighbor not in visit:
                visit[neighbor] = cur_node
                dist[neighbor] = dis + cur_dis
                stack.append((neighbor, dis + cur_dis))
    """
    Start with end node usr visit to construct the path
    and reverse it and return path,total_dist,num_visited
    """
    path = []
    current = end
    while current is not None:
        path.append(current)
        current = visit[current]
    path.reverse()
    total_dist = dist[end]
    return path, total_dist, num_visited
    # End your code (Part 2)
```

PART3:

py > ucs

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


def ucs(start, end):
    # Begin your code (Part 3)
    """
    Read the csv file and Iterate each
    row and create the graph
    """
    graph={}
    with open(edgeFile,mode='r') as file:
        reader=csv.reader(file)
        next(reader)
        for row in reader:
            source=int(row[0])
            target=int(row[1])
            distance=float(row[2])

            if source not in graph:
                graph[source]=[]

            graph[source].append((target,distance))
    """
    Initialize some datastructure to implement the ucs
    """
    queue=[(0,start)]
    visited=set()
    parent={start:None}
    cost={start:0}
    num_visited=0
```

```python
    num_visited=0
    """
    Implement the UCS using a priority queue (min-heap). Each element in the queue
    is a tuple containing the total cost to reach the current node and the node itself.
    `visited` is a set that tracks which nodes have been visited to prevent revisiting.
    `parent` stores the parent of each node for path reconstruction.
    `cost` keeps track of the minimum cost to reach each node from the start node.
    """
    while queue:
        cur_cost,cur_node=heapq.heappop(queue)
        num_visited+=1
        if cur_node==end:
            break
        if cur_node in visited:
            continue
        visited.add(cur_node)
        for neighbor,edge_cost in graph.get(cur_node,[]):
            if neighbor not in visited:
                new_cost=cur_cost+edge_cost
                if neighbor not in cost or new_cost <cost[neighbor]:
                    cost[neighbor]=new_cost
                    parent[neighbor]=cur_node
                    queue.append((new_cost,neighbor))

    """
    Start with end node usr visit to construct the path
    and reverse it and return path,total_dist,num_visited
    """
    path = []
```

```python
    current = end
    while current is not None:
        path.append(current)
        current = parent[current]
    path.reverse()

    return path, cost[end], num_visited

    # End your code (Part 3)
```

PART4:

```python
import csv
import heapq
edgeFile = 'edges.csv'
heuristicFile = 'heuristic.csv'


def astar(start, end):
    # Begin your code (Part 4)
    """
    Initialize the graph and heuristic dictionaries from CSV
    the heuristic need to chage when chage ID
    """
    graph = {}
    with open(edgeFile, mode='r') as file:
        reader = csv.reader(file)
        next(reader)
        for row in reader:
            source = int(row[0])
            target = int(row[1])
            distance = float(row[2])

            if source not in graph:
                graph[source] = []
            graph[source].append((target, distance))

    heuristic = {}
    with open(heuristicFile, mode='r') as file:
        reader = csv.reader(file)
        next(reader)
        for row in reader:
            node = int(row[0])
            h_value = float(row[3])#CHANG THE row[?] FOR ID1/2/3
            heuristic[node] = h_value
    """
    Initialize some datastructure to implement the astar
    """
```

```python
    queue = [(heuristic[start], 0, start)]
    visited = set()
    parent = {start: None}
    cost = {start: 0}
    num_visited = 0
    """
    A* search implementation using a priority queue. Each queue entry is a tuple
    containing the total estimated cost (f = g + h), the current cost to reach the node (g),
    and the node itself. The heuristic value (h) is an estimate of the cost to reach the goal
    from the current node.
    """

    while queue:
        _, cur_cost, cur_node = heapq.heappop(queue)
        if cur_node == end:
            num_visited += 1
            break

        if cur_node in visited:
            continue

        visited.add(cur_node)
        num_visited += 1

        for neighbor, edge_cost in graph.get(cur_node, []):
            if neighbor not in visited:
                new_cost = cur_cost + edge_cost
                if neighbor not in cost or new_cost < cost[neighbor]:
                    cost[neighbor] = new_cost
                    parent[neighbor] = cur_node
                    heapq.heappush(queue, (new_cost + heuristic.get(neighbor, float('inf')), new_cost, neighbor))
    """
    Start with end node usr visit to construct the path
    and reverse it and return path,total_dist,num_visited
    """
```

```python
    path = []
    current = end
    if current in parent:
        while current is not None:
            path.append(current)
            current = parent[current]
        path.reverse()
    return path, cost[end], num_visited

    # End your code (Part 4)
```

PART6(Bonus):

```python
def astar_time(start, end):
    # Begin your code (Part 6)
    """
    Read the graph and huristic value from the CSV file, converting distances
    and speeds into time costs.Convert km/h to m/s,and the distance change into
    time_cost by time=distance/speed
    """
    graph = {}
    max_speed = 0
    with open(edgeFile, mode='r') as file:
        reader = csv.reader(file)
        next(reader)
        for row in reader:
            num1, num2, distance, speed = int(row[0]), int(row[1]), float(row[2]), float(row[3])
            speed_m_s = speed * 1000 / 3600
            max_speed = max(max_speed, speed_m_s)
            if num1 not in graph:
                graph[num1] = []
            graph[num1].append((num2, distance / speed_m_s))

    heuristic = {}
    with open(heuristicFile, mode='r') as file:
        reader = csv.reader(file)
        next(reader)
        for row in reader:
            node = int(row[0])
            h_value = float(row[3]) / max_speed #CHANG THE row[?] FOR ID1/2/3
            heuristic[node] = h_value
    """
    Initialize some datastructure to implement the astar_time
    """
```

```python
    queue = [(heuristic.get(start, 0), 0, start)]
    visited = set()
    parent = {start: None}
    time_so_far = {start: 0}
    num_visited = 0
    """
    Is same as a star but the heuristic value is implement by speed and the
    distance change into time_cost
    """
    while queue:
        _, current_time, current_node = heapq.heappop(queue)
        if current_node == end:
            break

        if current_node in visited:
            continue

        visited.add(current_node)
        num_visited += 1

        for neighbor, time_cost in graph.get(current_node, []):
            if neighbor not in visited:
                new_time = current_time + time_cost
                if neighbor not in time_so_far or new_time < time_so_far[neighbor]:
                    time_so_far[neighbor] = new_time
                    parent[neighbor] = current_node
                    heapq.heappush(queue, (new_time + heuristic.get(neighbor, float('inf')), new_time, neighbor))

    """
    Start with end node usr visit to construct the path
    and reverse it and return path,total_time,num_visited
    """
    path = []
```

```python
    total_time = 0
    current = end
    if current in parent:
        while current is not None:
            path.append(current)
            current = parent[current]
        path.reverse()
        total_time = time_so_far[end]
    return path, total_time, num_visited
    # End your code (Part 6)
```

Part II. Results & Analysis (12%):

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

BFS:

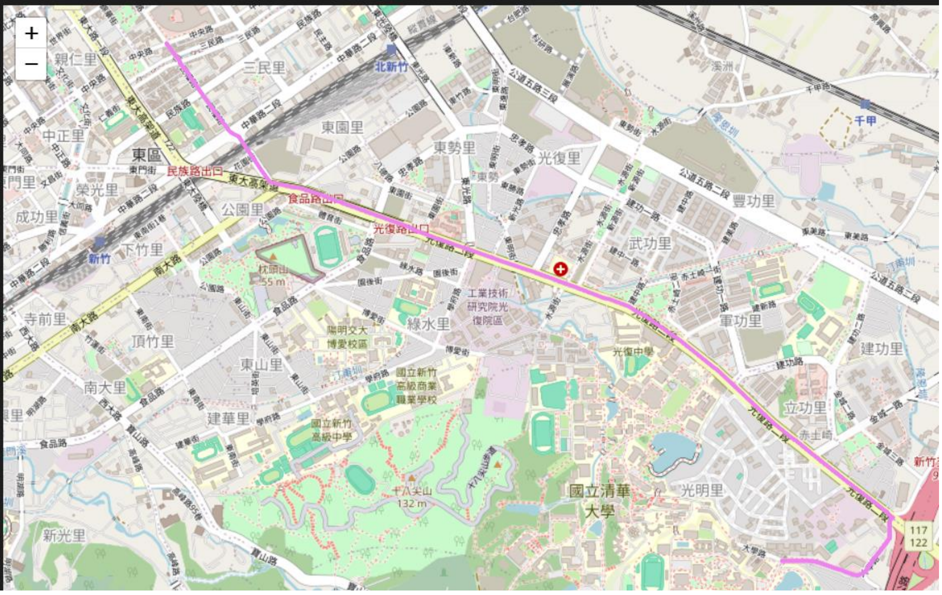
```
The number of nodes in the path found by BFS: 88
Total distance of path found by BFS: 4978.8820000000005 m
The number of visited nodes in BFS: 4273
```

DFS(stack)

```
The number of nodes in the path found by DFS: 1718
Total distance of path found by DFS: 75504.31499999983 m
The number of visited nodes in DFS: 4711
```

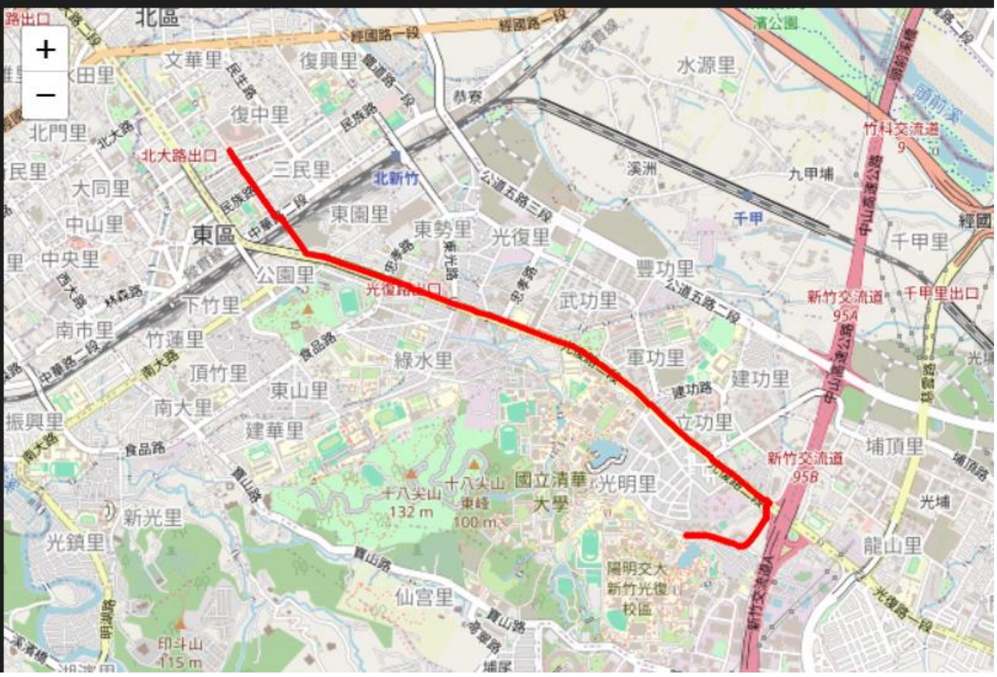
UCS:

```
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: 5233
```



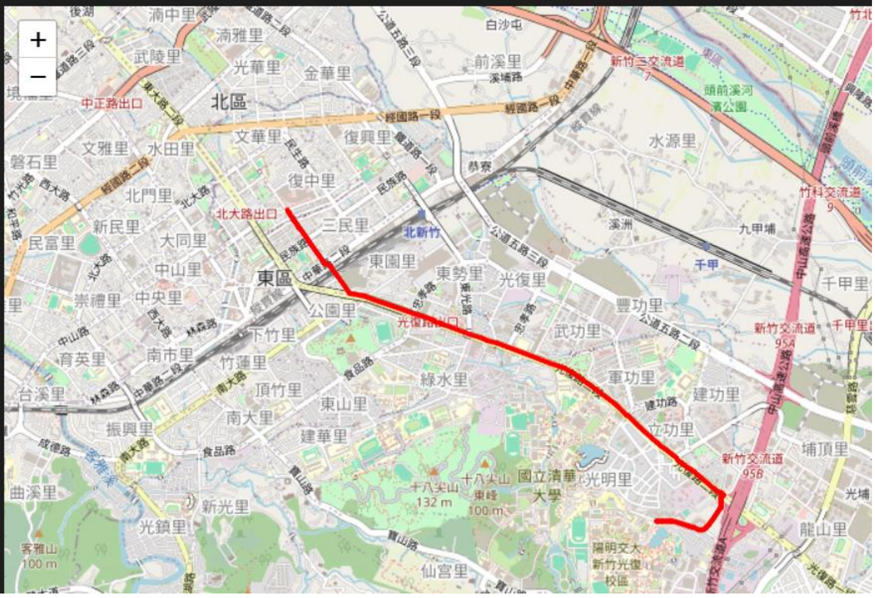
A*:

```
The number of nodes in the path found by A* search: 89
Total distance of path found by A* search: 4367.881 m
The number of visited nodes in A* search: 261
```



A*(time):

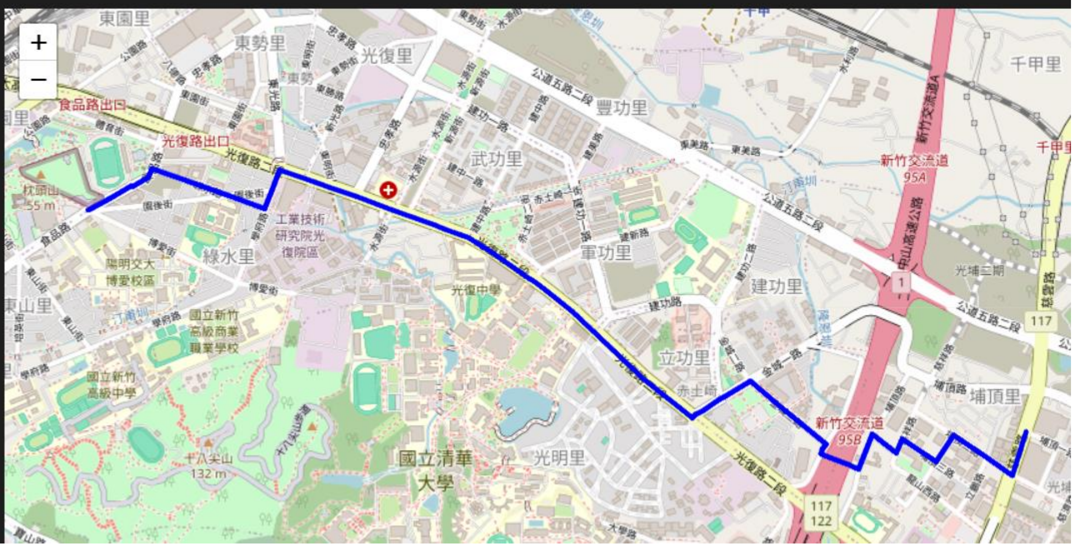
```
[4]  ✓ 0.4s

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: 1933
```



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

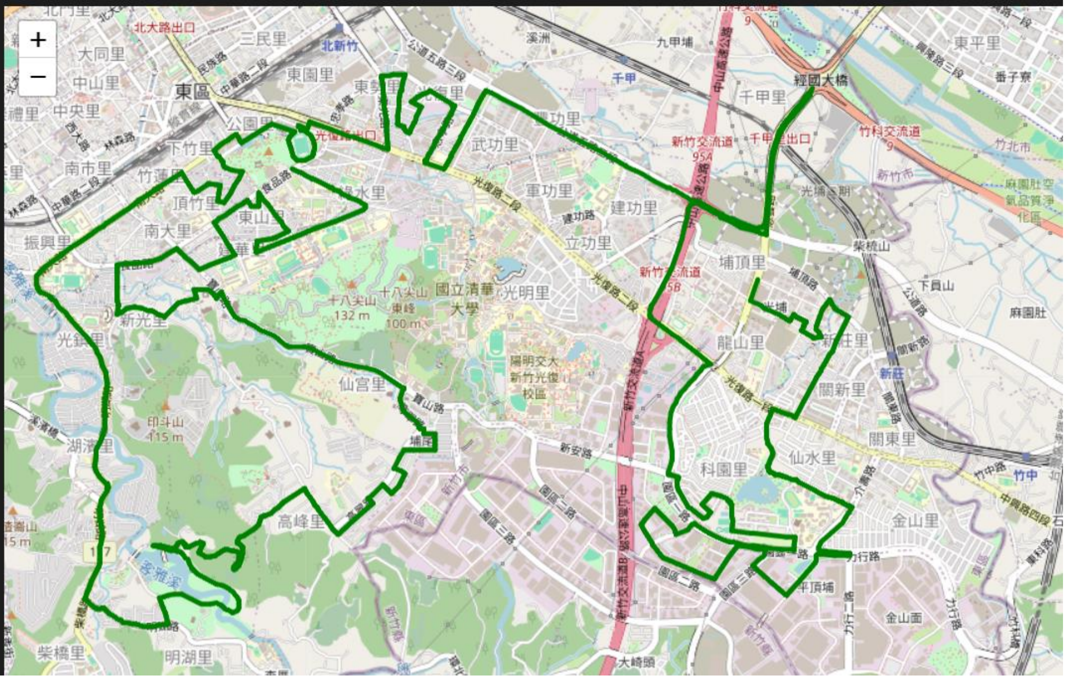
BFS:

```
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
```



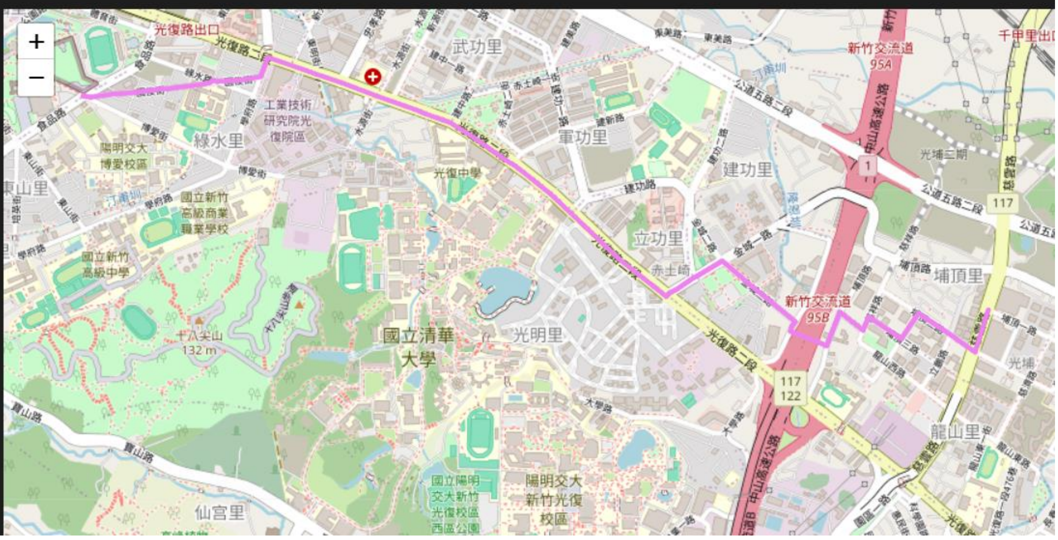
DFS(stack)

```
The number of nodes in the path found by DFS: 930
Total distance of path found by DFS: 38752.30799999996 m
The number of visited nodes in DFS: 9365
```



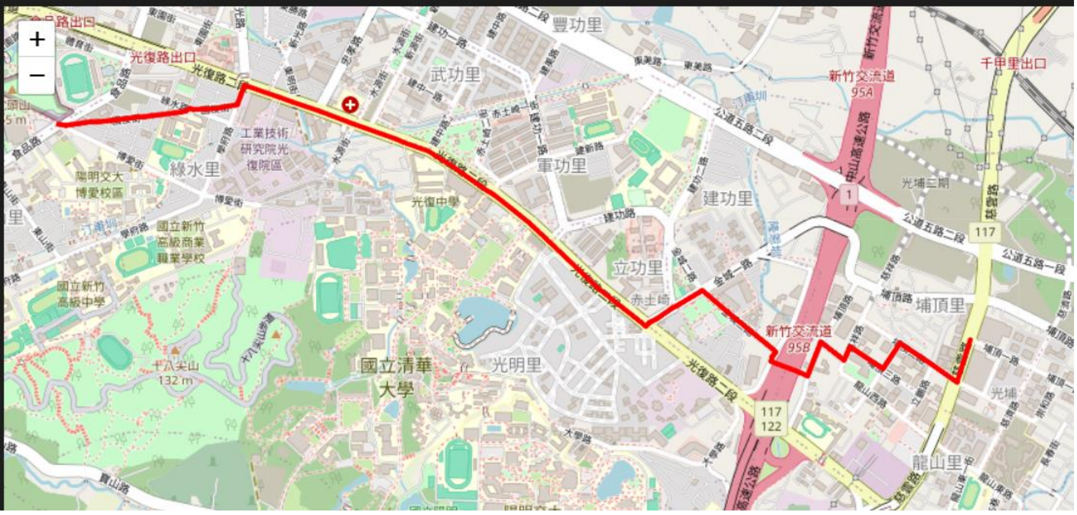
UCS:

```
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: 7453
```



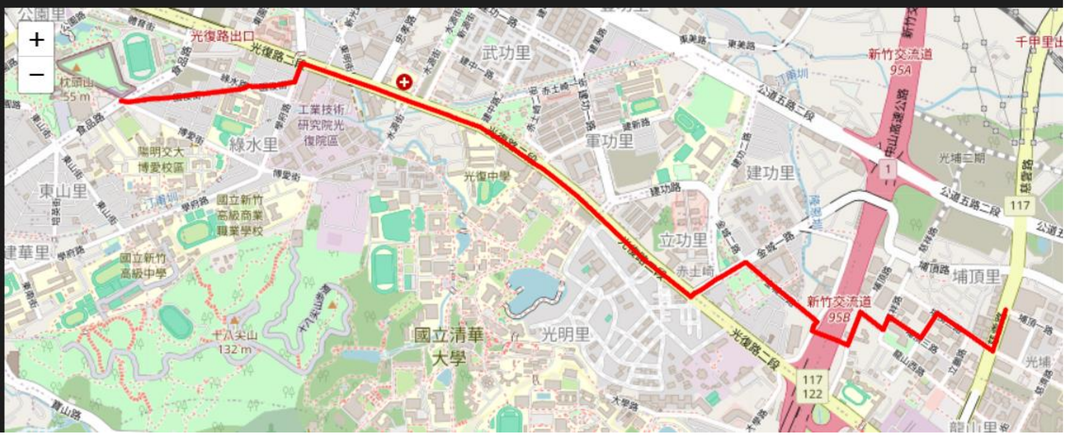
A*:

```
The number of nodes in the path found by A* search: 63
Total distance of path found by A* search: 4101.84 m
The number of visited nodes in A* search: 1172
```



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: 2869
```



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

BFS:

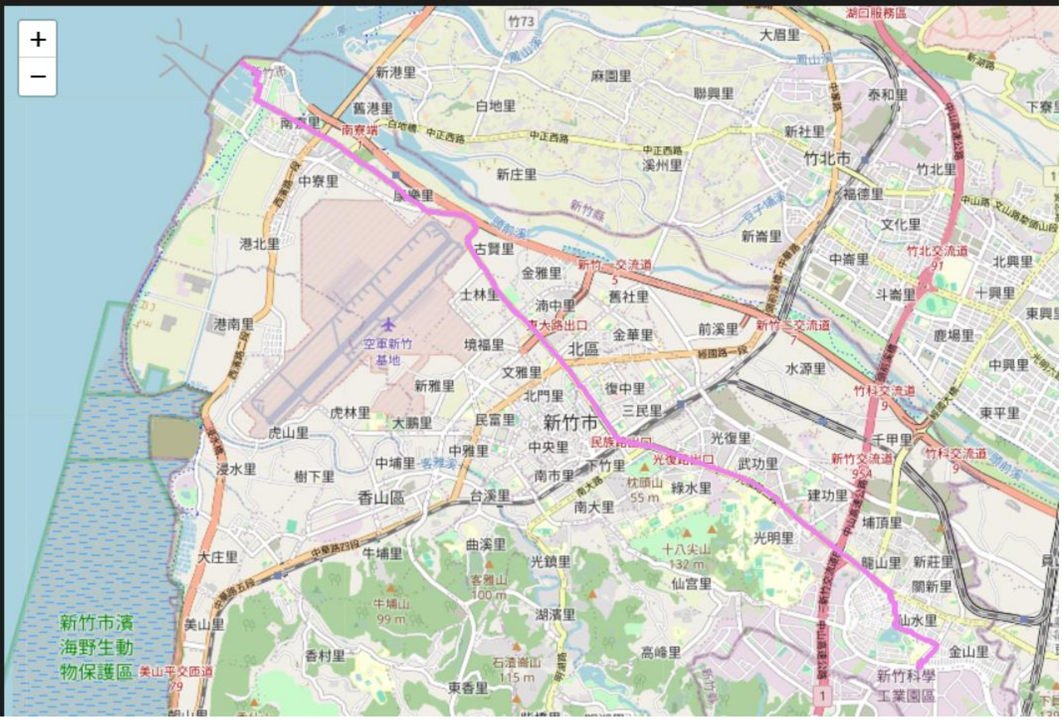
```
The number of nodes in the path found by BFS: 183
Total distance of path found by BFS: 15442.395000000002 m
The number of visited nodes in BFS: 11241
```

DFS(stack)

```
The number of nodes in the path found by DFS: 900
Total distance of path found by DFS: 39219.99299999996 m
The number of visited nodes in DFS: 2247
```

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: 12315
```



A*:

```
The number of nodes in the path found by A* search: 288
Total distance of path found by A* search: 14212.412999999997 m
The number of visited nodes in A* search: 7073
```

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: 8457
```

Based on the results, we can draw the following conclusions:

1. Depth-First Search (DFS) is the least efficient method for route finding, often resulting in longer and more circuitous routes, making it unsuitable for optimal pathfinding.

2. Breadth-First Search (BFS) provides near-optimal routes with a more reliable path than DFS, although it doesn't always produce the shortest distance, indicating a balance between accuracy and efficiency.

3. Both Uniform Cost Search (UCS) and A* Search algorithms are superior for finding the most reasonable and shortest routes. However, UCS's performance degrades with larger maps or greater

4. A* Search is the most efficient algorithm for route finding, particularly with a heuristic function based on distance and maximal speed limit, which is admissible but can be optimized to reduce the number of visited points and enhance efficiency in large-scale searches.

Part III. Question Answering (12%):

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

1.First,I don't know how to deal with the data .Therefore, I surf lots of website. Once I know how to do it is easier process the other part.

2.My DFS is quite different from the given result, but I think it is because a little bit difference in DFS implementation will cause different result.

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.

Traffic condition: When traffic congestion is heavy, it increases the travel time. Traffic condition is essential for route finding in the real world. Incorporating real-time traffic data helps avoid congestion, leading to faster and more reliable routes.

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

Mapping: Use satellites to create the detail maps we want.

Localization: Use GPS to determine a vehicle's location in real time.

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

Prep Time: Time taken to prepare the food.

Time to Customer: Estimated travel time based on the distance and the average speed.

Traffic Delay: Additional time cost based on the traffic conditions.

Priority Adjustment: Time subtracted or added based on the delivery priority.

ETA=Prep Time + Time to Customer + Traffic Delay + Priority Adjustment