Homework 2: Route Finding

110700045 朱冠霖

Part I. Implementation (6%):

Part 1

```
def bfs(start, end):
    # Begin your code (Part 1)
    # raise NotImplementedError("To be implemented")

    Graph = defaultdict(list) # Use a list to store the graph information

    visited_node = 0 # Total nodes we visited
    path = 0.0 # Total distance we have gone through
    road = [] # Store the nodes we have gone through
    visited = set() # To record the nodes we have visited
    From = {} # Store the node where we come from
    queue = [] # A queue for doning BFS

    queue.append(start) # Put the starting point in the queue
    From[start] = start
    visited.add(start)
    find = False

    # Read all the data from edges.csv and construct the grpah
    with open(edgeFile, newline='') as f:
        data = csv.reader(f)
        temp = next(data)
        for line in data:
            s, e, dis, speed = line
            Graph[int(s)].append([int(e), float(dis)])

    # Do BFS (Using queue)
    while len(queue) > 0:
        current = queue.pop(0) # Get the first element in the queue and remove it
        for neighbor, dist in Graph[current]: # The nodes which connect with it
            if neighbor not in visited: # If the node we haven't visited before
                visited_node += 1 # Update the total nodes we visited
                From[neighbor] = [current, dist] # Record where it is from and the distance
                queue.append(neighbor) # Put it in the queue
                visited.add(neighbor) # Update since we have visited it

            if neighbor == end: # If we found our destination
                find = True
                break
        if find:
            break

    # Traverse all the nodes we have visited (start from the destination)
    now = end
    while now != start:
        Previous, dis = From[now] # Get the node where we came from
        road.append(int(now))  # Put the node in the list
        now = Previous # Update the node
        path += dis # Update the distance we have gone through
    road.append(start)

    return road, path, visited_node
    # End your code (Part 1)
```

Part 2

```python
Graph = defaultdict(list)  # Use a list to store the graph information
visited = set()  # To record the nodes we have visited
road = []  # Store the nodes we have gone through
visited_node = 0  # Total nodes we visited
path = 0.0  # Total distance we have gone through


def dfs_recur(now, end):
    global Graph, visited, road, visited_node, path
    visited.add(now)  # Update the node we have visited
    visited_node += 1  # Update the number of nodes we have gone through
    if now == end:  # If the reach our destination
        return True
    else:
        for neighbor, dist in Graph[now]:  # The nodes which connect with it
            if neighbor not in visited:  # If the node we haven't visited before
                if dfs_recur(neighbor, end):  # Recursive call the function
                    road.append(neighbor)
                    path += dist
                    return True
        return False


def dfs(start, end):
    # Begin your code (Part 2)
    # raise NotImplementedError("To be implemented")

    global Graph
    # Read all the data from edges.csv and construct the grpah
    with open(edgeFile, newline='') as f:
        edge = csv.reader(f)
        headers = next(edge)
        for line in edge:
            s, e, dis, speed = line
            s = int(s)
            e = int(e)
            dis = float(dis)
            Graph[s].append([e, dis])

    dfs_recur(start, end)  # Start doing DFS
    road.append(start)

    return road, path, visited_node
    # End your code (Part 2)
```

Part 3

```
def ucs(start, end):
    # Begin your code (Part 3)
    # raise NotImplementedError("To be implemented")
    Graph = defaultdict(list) # Use a list to store the graph information
    path = 0.0 # Total distance we have gone through
    road = [] # Store the nodes we have gone through
    queue = [] # A "priority queue" for doning UCS
    visited = set() # To record the nodes we have visited
    From = {} # Store the node where we come from
    visited_node = 0 # Total nodes we visited

    # Read all the data from edges.csv and construct the grpah
    with open(edgeFile, newline='') as f:
        data = csv.reader(f)
        temp = next(data)
        for line in data:
            s, e, dis, speed = line
            Graph[int(s)].append([int(e), float(dis)])

    """
    A pirority queue, first element is the node, the second is its weight,
    the third is its parent, the fourth is distance
    """
    queue.append([start, 0.0, start, 0])
    From[start] = [start, 0.0] # First element is where the node come from, and the second is the distance
    visited_node += 1 # Update the number of node we have visited

    while len(queue) > 0:
        weight = float('inf')
        cur = 0
        previous = 0
        d = 0.0

        """
        Get the smallest weight in the priority queue because we always want to
        get the lowest cost in UCS
        """
        for tmp, w, p, dist in queue:
            if w < weight:
                weight = w
                cur = tmp
                previous = p
                d = dist
        queue.remove([cur, weight, previous, d]) # Remove the lowest cost in the priority queue

        if cur in visited: # If we have visited the node, then redo the loop to get another node
            continue
        visited_node += 1 # Update the number of node we have visited
        From[cur] = [previous, d] # Record the distance and how we arrival the node
        visited.add(cur) # Update since we have visited it

        if cur == end: # If we reach the destination
            break
        for neighbor, dist in Graph[cur]: # Put the unvisited nodes of cur's neighbors to the priority queue
            if neighbor not in visited:
                queue.append([neighbor, dist+weight, cur, dist])

    # Traverse all the nodes we have visited (start from the destination)
    now = end
    while now != start:
        Previous, dis = From[now] # Get the node where we came from
        road.append(int(now)) # Put the node in the list
        now = Previous # Update the node
        path += dis # Update the distance we have gone through

    road.append(start)

    return road, path, visited_node
    # End your code (Part 3)
```

Part 4

```python
def astar(start, end):
    # Begin your code (Part 4)
    # raise NotImplementedError("To be implemented")

    Graph = defaultdict(list) # Use a list to store the graph information
    visited_node = 0 # Total nodes we visited
    path = 0.0 # Total distance we have gone through
    road = [] # Store the nodes we have gone through
    visited = set() # To record the nodes we have visited
    From = {} # Store the node where we come from
    queue = [] # A priority queue for doning A* search
    heuristic = {} # Heuristic function

    # Read all the data from edges.csv and construct the grpah
    with open(edgeFile, newline='') as f:
        data = csv.reader(f)
        temp = next(data)
        for line in data:
            s, e, dis, speed = line
            Graph[int(s)].append([int(e), float(dis)])

    # Read all the data from heuristic.csv and use heuristic to record heuristic function
    with open(heuristicFile, newline='') as f:
        lines = csv.reader(f)
        temp = next(lines)
        for line in lines:
            for i in range(1, 4):
                if int(temp[i]) == end:
                    heuristic[int(line[0])] = float(line[i])
    """
    A pirority queue, first element is the node, the second is its
    heuristic value + distance, the third is its parent, the fourth is distance
    """
    queue.append([start, heuristic[start], start, 0])
    From[start] = [start, 0.0] # First element is where the node come from, and the second is the distance

    # Do A* search
    while len(queue) > 0:
        weight = float('inf')
        cur = 0
        previous = 0
        dist = 0.0

        """
        Get the smallest value of heuristic value + distance in the priority
        queue because we always want to get the lowest cost in A* search
        """
        for tmp, w, p, dis in queue:
            if w < weight:
                weight = w
                cur = tmp
                previous = p
                dist = dis
        queue.remove([cur, weight, previous, dist]) # Remove the chosen one

        if cur in visited: # If we have visited the node, then redo the loop to get another node
            continue

        visited.add(cur)
        visited_node += 1
        From[cur] = [previous, dist]
        weight -= heuristic[cur] # Because weight includes the heuristic value

        if cur == end: # If we reach the destination
            break

        # Put the unvisited nodes of cur's neighbors to the priority queue
        for neighbor, dis in Graph[cur]:
            if neighbor not in visited:
                queue.append([neighbor, dis+weight+heuristic[neighbor], cur, dis])

    # Traverse all the nodes we have visited (start from the destination)
    now = end
    while now != start:
        Previous, dis = From[now] # Get the node where we came from
        road.append(int(now)) # Put the node in the list
        now = Previous # Update the node
        path += dis # Update the distance we have gone through

    road.append(start)

    return road, path, visited_node
    # End your code (Part 4)
```

Part 6 (Bonus)

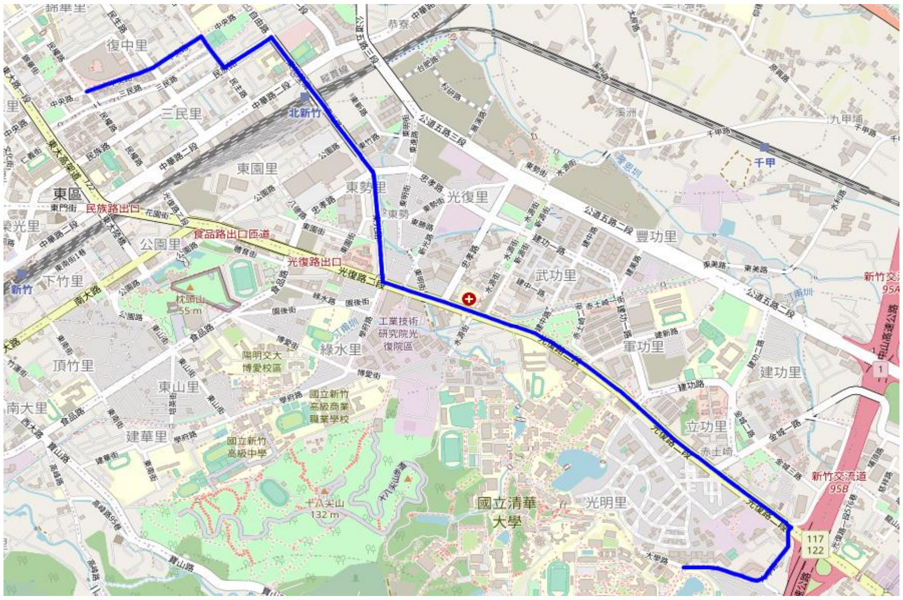
```python
def astar_time(start, end):
    # Begin your code (Part 6)
    # raise NotImplementedError("To be implemented")
    """
      This part is very similiar to part 4 and the three differences are the heuristic
      value, g(n) value and we want to get the maximum speed value.
      At this part, the heuristics value is straight line distance between
      strat node and end node / maximum speed, g(n) is the fastest time we need
      to go to this node which is equal to distance/speed limit
    """
    Graph = defaultdict(list)  # Use a list to store the graph information
    visited_node = 0  # Total nodes we visited
    path = 0.0  # Total distance we have gone through
    road = []  # Store the nodes we have gone through
    visited = set()  # To record the nodes we have visited
    From = {}  # Store the node where we come from
    queue = []  # A priority queue for doning A* time search
    heuristic = {}  # Heuristic function
    Max_Speed = 0  # Record the maximum speed

    # Read all the data from edges.csv and construct the grpah
    with open(edgeFile, newline='') as f:
        data = csv.reader(f)
        temp = next(data)
        for line in data:
            s, e, dis, speed = line
            Max_Speed = max(Max_Speed, float(speed)/3.6)
            time = float(dis)/(float(speed)/3.6)
            Graph[int(s)].append([int(e), time])

    # Read all the data from heuristic.csv and use heuristic to record heuristic function
    with open(heuristicFile, newline='') as f:
        lines = csv.reader(f)
        temp = next(lines)
        for line in lines:
            for i in range(1, 4):
                if int(temp[i]) == end:
                    heuristic[int(line[0])] = float(line[i])/Max_Speed

    """
    A pirority queue, first element is the node, the second is its
    heuristic value + time, the third is its parent, the fourth is time
    """
    queue.append([start, heuristic[start], start, 0])
    From[start] = [start, 0.0]
    while len(queue) > 0:
        weight = float('inf')
        cur = 0
        previous = 0
        time = 0.0

        """
        Get the smallest value of heuristic value + time in the priority
        queue because we always want to get the lowest cost in A* search
        """
        for tmp, w, p, t in queue:
            if w < weight:
                weight = w
                cur = tmp
                previous = p
                time = t
        queue.remove([cur, weight, previous, time])  # Remove the chosen one

        if cur in visited:
            continue

        From[cur] = [previous, time]
        weight -= heuristic[cur]  # Because weight includes the heuristic value
        visited.add(cur)
        visited_node += 1

        if cur == end:  # If we have visited the node, then redo the loop to get another node
            break

        # Put the unvisited nodes of cur's neighbors to the priority queue
        for neighbor, t in Graph[cur]:
            queue.append(
                [neighbor, t+weight+heuristic[neighbor], cur, t])

    # Traverse all the nodes we have visited (start from the destination)
    now = end
    while now != start:
        Previous, time = From[now]  # Get the node where we came from
        road.append(int(now))  # Put the node in the list
        now = Previous  # Update the node
        path += time  # Update the distance we have gone through
    road.append(start)

    return road, path, visited_node

    # 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)

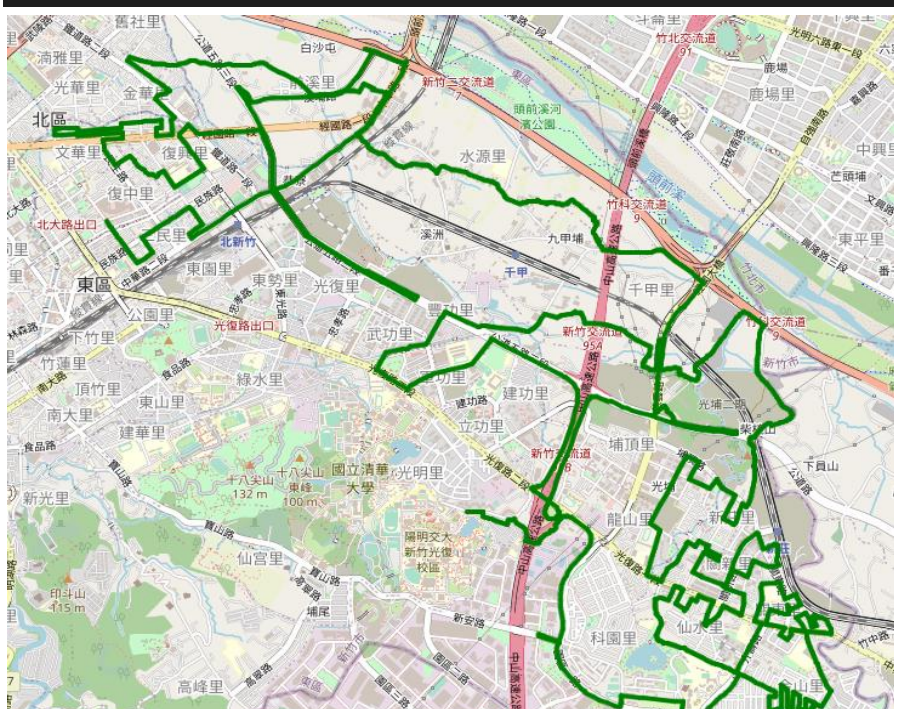
BFS:

```
The number of nodes in the path found by BFS: 88
Total distance of path found by BFS: 4978.881999999998 m
The number of visited nodes in BFS: 4273
```



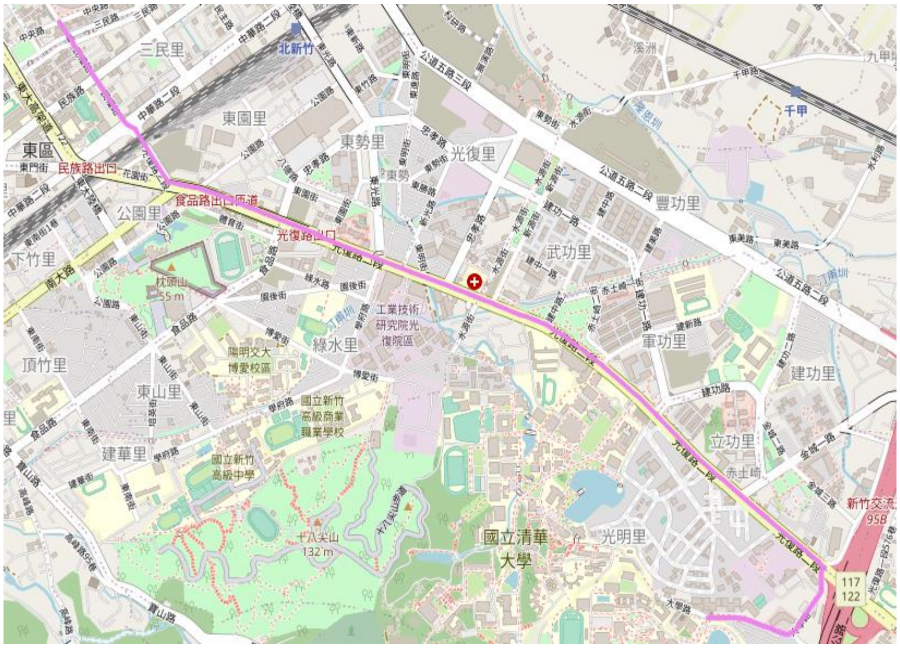
DFS (recursion):

```
The number of nodes in the path found by DFS: 1311
Total distance of path found by DFS: 48954.32099999998 m
The number of visited nodes in DFS: 3519
```



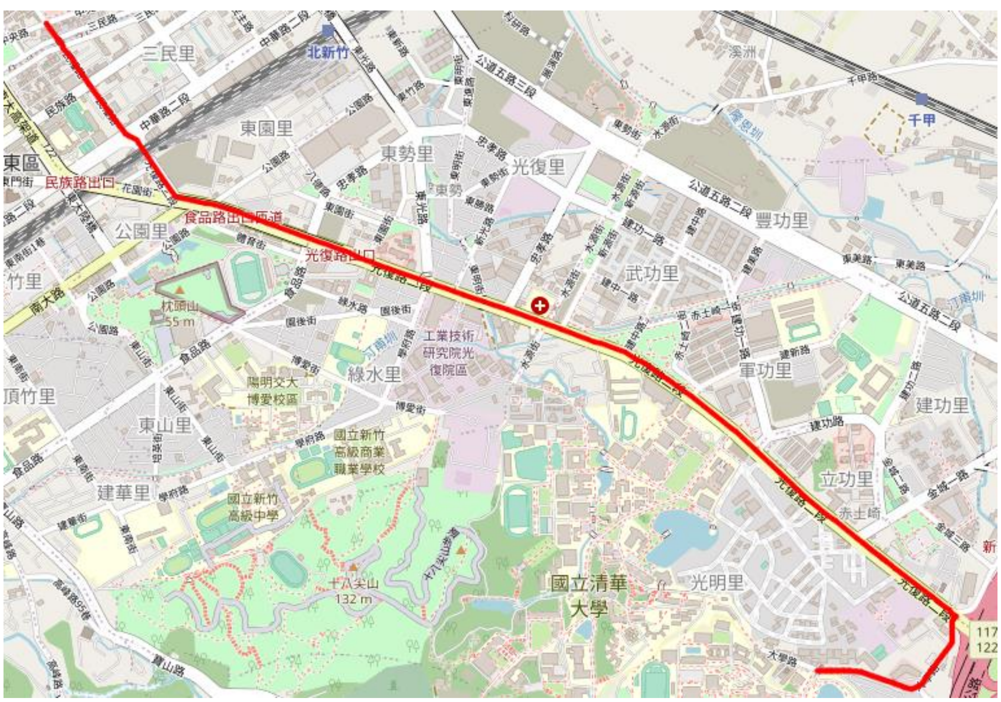
Uniform Cost Search:

```
The number of nodes in the path found by UCS: 89
Total distance of path found by UCS: 4367.8809999999985 m
The number of visited nodes in UCS: 5087
```



A* search:

```
The number of nodes in the path found by A* search: 89
Total distance of path found by A* search: 4367.8809999999985 m
The number of visited nodes in A* search: 261
```



A* time search (Bonus):

```
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: 1934
```

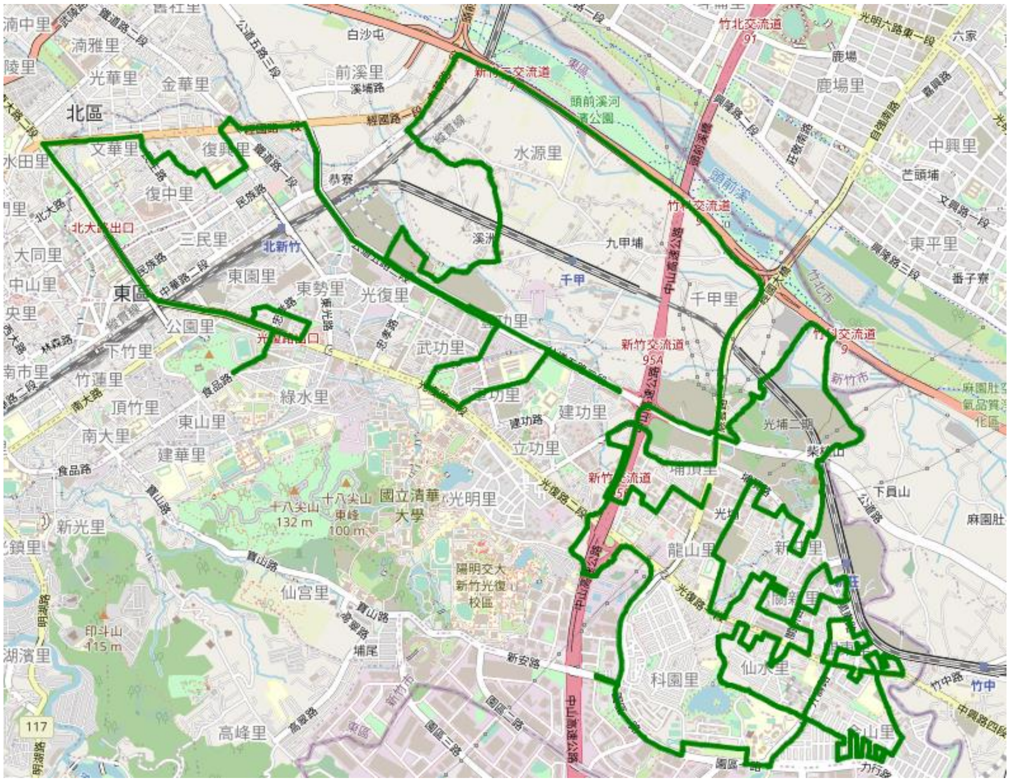
Test2: From Hsinchu Zoo (ID: 426882161) to COSTCO Hsinchu Store (ID: 1737223506)

BFS:

```
The number of nodes in the path found by BFS: 60
Total distance of path found by BFS: 4215.521000000001 m
The number of visited nodes in BFS: 4606
```

DFS (Recursive):

```
The number of nodes in the path found by DFS: 1019
Total distance of path found by DFS: 43504.76899999997 m
The number of visited nodes in DFS: 10626
```

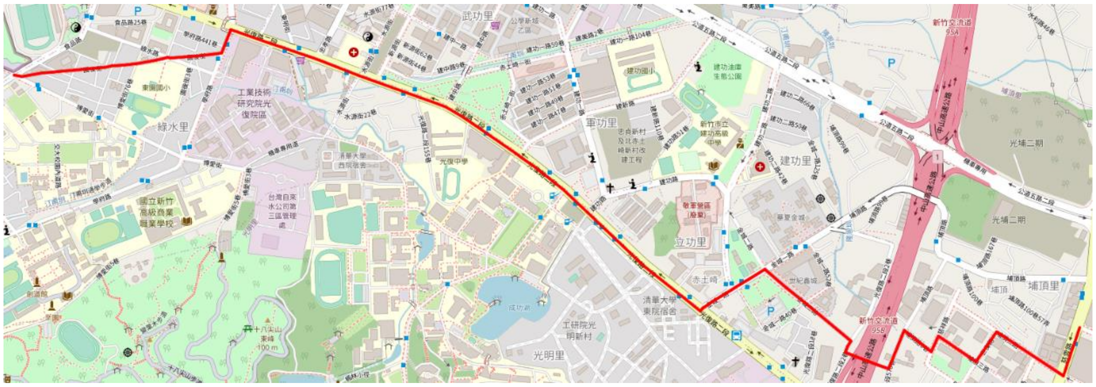


Uniform Cost Research:

```
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: 7214
```

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
```



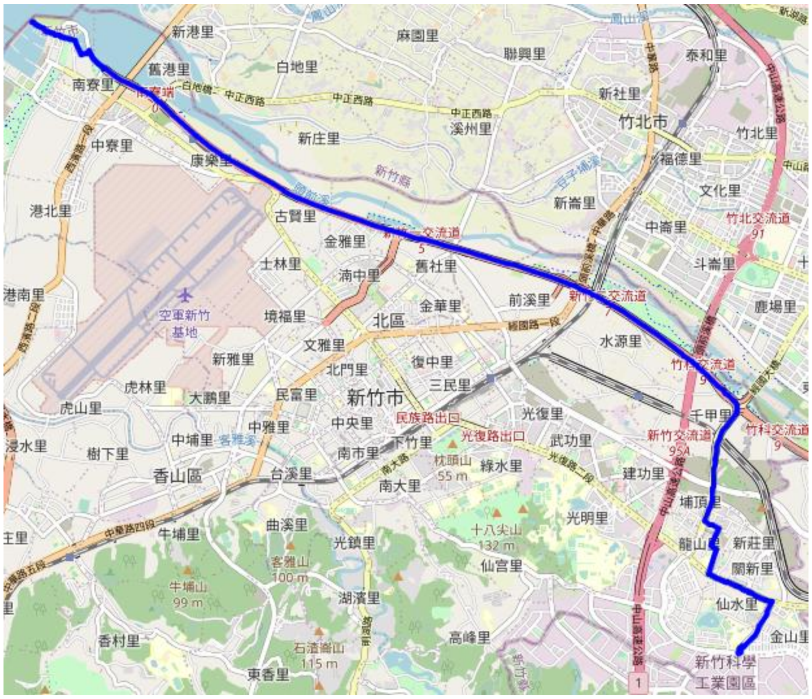
A* time search (Bonus):

```
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: 2870
```

Test3: From National Experimental High School At Hsinchu Science Park (ID: 1718165260) to Nanliao Fishing Port (ID: 8513026827)

BFS:

```
The number of nodes in the path found by BFS: 183
Total distance of path found by BFS: 15442.394999999995 m
The number of visited nodes in BFS: 11241
```



DFS (Recursive):

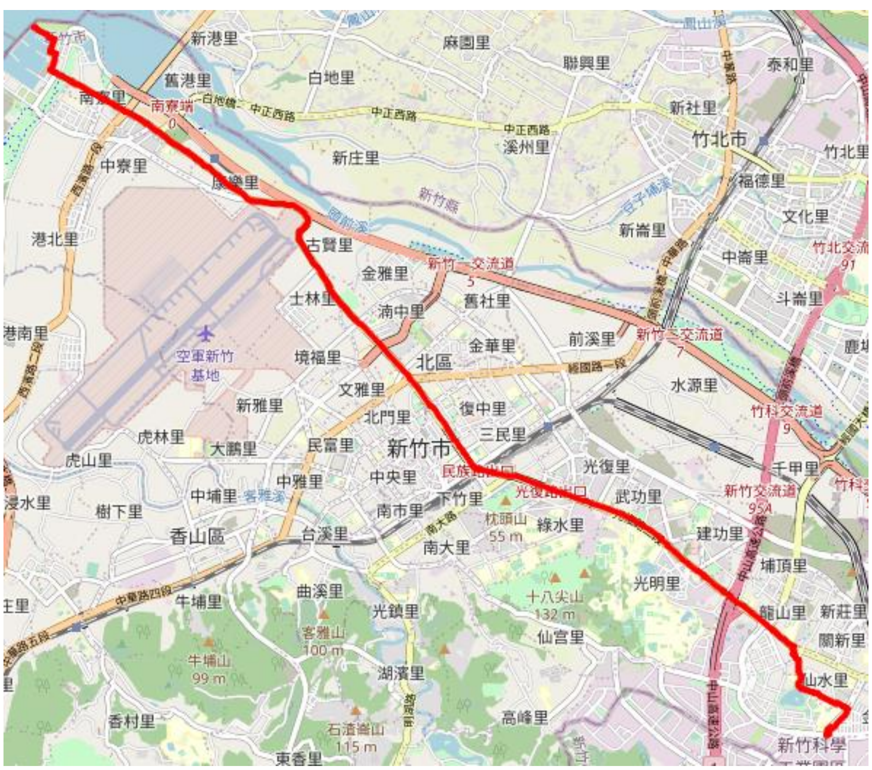
```
The number of nodes in the path found by DFS: 2635
Total distance of path found by DFS: 120440.44300000017 m
The number of visited nodes in DFS: 7518
```

Uniform Cost Search:

```
The number of nodes in the path found by UCS: 288
Total distance of path found by UCS: 14212.413 m
The number of visited nodes in UCS: 11927
```

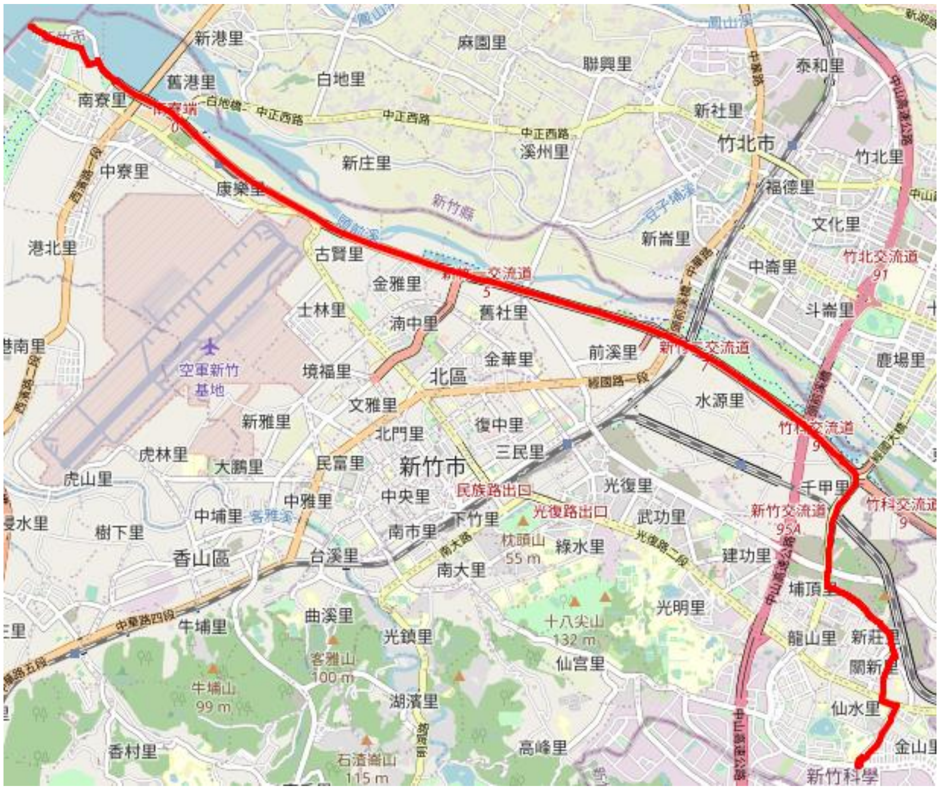
A* search:

```
The number of nodes in the path found by A* search: 288
Total distance of path found by A* search: 14212.413 m
The number of visited nodes in A* search: 7073
```



A* time search (Bonus):

```
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: 8458
```



Conclusion:

- The number of nodes in the path: BFS has the smallest number of nodes, while DFS has the largest.
- Total distance of the path: UCS and A* search have the smallest distance while DFS has the largest.
- The number of visited nodes: A* search has the smallest visited nodes while UCS has the largest.

DFS is not suitable for path finding, however, A* search is the best.

In Part 6, although it has a larger number of visited nodes, it is faster than Part 5.

Part III. Question Answering:

1. Please describe a problem you encountered and how you solved it.
    - Because my Algorithm course teacher didn't have enough time last semester to teach us DFS and BFS, I spent a lot of reading it and understanding it.
    - I don't know how to get all the nodes that I have gone through, so I used a dictionary called "From" to record it.

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 light, when we encounter a red light, we must stop our vehicles or we will make the red-light violation, thus, if this road has more traffic light, this path may be slower.
    - Velocity, in real world it has traffic jam that could affect our speed, so the speed limit may not be suitable in real world. I think we can calculate the average speed for every 5 km and update it periodically, it will be more accurate than using speed limit.

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

      GPS is a prevalent technology used for mapping; it uses signals from satellites to ascertain the location of the device.
    - Localization:

      Our smartphone can use GPS receivers to receive signals from GPS satellites and determine the device's location.

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.
    - I think ETA includes meal prep time and delivery time. In meal prep time, we can collect data to estimate the time that the restaurant needs, of course, the more food you order, the more prep time will be. And in delivery time, the traffic condition will definitely affect the time, so I think we can calculate the average speed and update it periodically, and it can be defined as a dynamic heuristic equation.