Spring 2022

Introduction to Artificial Intelligence

Homework 2: Route Finding 109550159 李驊恩

Part I. Implementation (The explanation of my implementation is in the **notation** of my code.)

Part 1: Breadth-first Search

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

sys.setrecursionlimit(20000)

# Store the imformation of every nodes
# EDGES[node number] = [(reachable node #1, distance #1),(reachable node #2, distance #2)]
EDGES = {}
with open(edgeFile, newline='') as csvfile:

    # read every edges then construct a graph
    rows = csv.DictReader(csvfile)
    for row in rows:
        s = int(row['start'])
        e = int(row['end'])
        d = float(row['distance'])
        if s in EDGES:
            EDGES[s] += [(e, d)]
        else:
            EDGES[s] = [(e, d)]

# Variables have to be put outside of the function because we applied the recursive version
PATH = []
DIST = 0
VISITED = set()
BEGIN = True

def dfs(start, end):
    # Begin your code (Part 2)
    global PATH
    global DIST
    global VISITED

    PATH.append(start)
    VISITED.add(start)

    # The end of the recursion
    if(start == end):
        return PATH, DIST, len(VISITED)+1

    # Visit every node which 'start' can reach and we nver tried
    for nxt_node, d in EDGES[start]:

        if not nxt_node in EDGES:
            continue

        # If the node have been visited, then skip.
        if nxt_node in VISITED:
            continue

        # Keep going
        DIST += d
        ret_path, ret_dist, ret_num_visited = dfs(nxt_node, end)
        if ret_path != []:

            # Find a path successfully, and return the path
            return ret_path, ret_dist, ret_num_visited
        else:

            # No way to go, try the next one
            DIST -= d

    # No way to go
    PATH.pop()
    return [],None,None

    raise NotImplementedError("To be implemented")

    # End your code (Part 2)
```

Part 2: Depth-first Search (recursive)

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

sys.setrecursionlimit(20000)

# Store the imformation of every nodes
# EDGES[node number] = [(reachable node #1, distance #1),(reachable node #2, distance #2)]
EDGES = {}
with open(edgeFile, newline='') as csvfile:

    # read every edges then construct a graph
    rows = csv.DictReader(csvfile)
    for row in rows:
        s = int(row['start'])
        e = int(row['end'])
        d = float(row['distance'])
        if s in EDGES:
            EDGES[s] += [(e, d)]
        else:
            EDGES[s] = [(e, d)]
# Variables have to be put outside of the function because we applied the recursive version
PATH = []
DIST = 0
VISITED = set()
BEGIN = True

def dfs(start, end):
    # Begin your code (Part 2)
    global PATH
    global DIST
    global VISITED

    PATH.append(start)
    VISITED.add(start)

    # The end of the recursion
    if(start == end):
        return PATH, DIST, len(VISITED)+1

    # Visit every node which 'start' can reach and we nver tried
    for nxt_node, d in EDGES[start]:

        if not nxt_node in EDGES:
            continue
        # If the node have been visited, then skip.
        if nxt_node in VISITED:
            continue

        # Keep going
        DIST += d
        ret_path, ret_dist, ret_num_visited = dfs(nxt_node, end)
        if ret_path != []:

            # Find a path successfully, and return the path
            return ret_path, ret_dist, ret_num_visited
        else:

            # No way to go, try the next one
            DIST -= d

    # No way to go
    PATH.pop()
    return [],None,None
    raise NotImplementedError("To be implemented")

    # End your code (Part 2)
```

Part 3: Uniform Cost Search

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


def ucs(start, end):
    # Begin your code (Part 3)
    edges = {}
    with open(edgeFile, newline='') as csvfile:
        rows = csv.DictReader(csvfile)
        for row in rows:
            s = int(row['start'])
            e = int(row['end'])
            d = float(row['distance'])
            if s in edges:
                edges[s] += [(e, d)]
            else:
                edges[s] = [(e, d)]

    # dist[node number] = (distance from the starting point, the number of the former node)
    dist = {}
    Q = []

    # Put it in the starting point first, every information in the queue is shown as (node number
    heapq.heappush(Q, (0, start, None))
    while len(Q) > 0:
        cur_dist, cur_node, prev_node = heapq.heappop(Q)

        # exclude those have been visited, because we won't find a shorter path
        if cur_node in dist:
            continue

        # The distance of the starting point and the current node can be decided here
        dist[cur_node] = (cur_dist, prev_node)
        # Find the path
        if cur_node == end:
            break

        # Expand from the cuurent node, and try the nodes that are connected to the current node
        if not cur_node in edges:
            continue

        # Exclude those nodes that have no way to go
        for nxt_node, d in edges[cur_node]:

            # Exclude those have been visited, because we won't find a shorter path
            if nxt_node in dist:
                continue
            heapq.heappush(Q, (cur_dist + d, nxt_node, cur_node))

    # Backtrace from the ending node
    ret_path = [end]
    while dist[ret_path[-1]][1] != None:
        ret_path.append(dist[ret_path[-1]][1])
    ret_dist = dist[end][0]
    ret_num_visited = len(dist)
    return ret_path[::-1], ret_dist, ret_num_visited

    raise NotImplementedError("To be implemented")

    # End your code (Part 3)
```

Part 4: A* Search

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


def astar(start, end):
    # Begin your code (Part 4)

    # Store the imformation of every nodes.
    # edges[node number] = [(reachable node #1, distance #1),(reachable node #2, distance #2)]
    edges = {}
    with open(edgeFile, newline='') as csvfile:

        # read every edges then construct a graph
        rows = csv.DictReader(csvfile)
        for row in rows:
            s = int(row['start'])
            e = int(row['end'])
            d = float(row['distance'])
            if s in edges:
                edges[s] += [(e, d)]
            else:
                edges[s] = [(e, d)]
            if e in edges:
                edges[e] += [(s, d)]
            else:
                edges[e] = [(s, d)]
    # Construct a graph that represent the linear distance of each node to the destination.
    # heuristic[node number] = the linear distance to the destination.
    heuristic = {}
    with open(heuristicFile, newline='') as csvfile:
        # Read in the distance and construct a graph.
        rows = csv.DictReader(csvfile)
        for row in rows:
            heuristic[int(row['node'])] = float(row[str(end)])

    # dist[node number] =
    # (The path's distance with the starting point + The linear distance with the destination)
    dist = {}
    Q = []
    heapq.heappush(Q, (0+heuristic[start], start, None))
    while len(Q) > 0:
        cur_dist, cur_node, prev_node = heapq.heappop(Q)
        # Exclude those have been visited, because we can't find a shorter path.
        if cur_node in dist:
            continue
        dist[cur_node] = (cur_dist, prev_node)
        # Find the path
        if cur_node == end:
            break
        # Expand from the cuurent node, and try the nodes that are connected to the current node.
        for nxt_node, d in edges[cur_node]:
            # Exclude those have been visited, because we won't find a shorter path.
            if nxt_node in dist:
                continue
            heapq.heappush(Q, (cur_dist+d-heuristic[cur_node]+heuristic[nxt_node], nxt_node, cur_node))

    # Backtrace from the ending node
    ret_path = [end]
    while dist[ret_path[-1]][1] != None:
        ret_path.append(dist[ret_path[-1]][1])
    ret_dist = dist[end][0]
    ret_num_visited = len(dist)
    return ret_path[::-1], ret_dist, ret_num_visited
    raise NotImplementedError("To be implemented")
    # End your code (Part 4)
```

Part 6: A* search (fastest path)

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


def astar_time(start, end):
    # Begin your code (Part 6)
    # Store the imformation of every nodes.
    # edges[node number] = [(reachable node #1, distance #1),(reachable node #2, distance #2)]
    edges = {}
    max_speed = 0
    with open(edgeFile, newline='') as csvfile:
        # read every edges then construct a graph
        rows = csv.DictReader(csvfile)
        for row in rows:
            s = int(row['start'])
            e = int(row['end'])
            d = float(row['distance'])
            t = d/(float(row['speed limit'])/3.6)
            if s in edges:
                edges[s] += [(e, d, t)]
            else:
                edges[s] = [(e, d, t)]
            if float(row['speed limit']) > max_speed:
                max_speed = float(row['speed limit'])
    # Construct a graph that represent the linear distance of each node to the destination.
    # heuristic[node number] = the linear distance to the destination.
    heuristic = {}
    with open(heuristicFile, newline='') as csvfile:
        # Read in the distance and construct a graph.
        rows = csv.DictReader(csvfile)
        for row in rows:
            heuristic[int(row['node'])] = float(row[str(end)])/(max_speed/3.6)
    # times[node number] =
    # (The path's travel time with the starting point + The linear distance with the destination /Maximum speed s/m
    #  , the previous node)
    times = {}
    Q = []
    heapq.heappush(Q, (0+heuristic[start], start, None))
    while len(Q) > 0:
        cur_time, cur_node, prev_node = heapq.heappop(Q)
        # Exclude those have been visited, because we can't find a shorter path.
        if cur_node in times:
            continue
        times[cur_node] = (cur_time, prev_node)
        # Find the path
        if cur_node == end:
            break
        # Exclude those node that have no way to go
        if not cur_node in edges:
            continue
        # Expand from the cuurent node, and try the nodes that are connected to the current node.
        for nxt_node, d, t in edges[cur_node]:
            # Exclude those have been visited, because we won't find a shorter path.
            if nxt_node in times:
                continue
            heapq.heappush(Q, (cur_time+t-heuristic[cur_node]+heuristic[nxt_node], nxt_node, cur_node))

    # Backtrace from the ending node
    ret_path = [end]
    while times[ret_path[-1]][1] != None:
        ret_path.append(times[ret_path[-1]][1])
    ret_time = times[end][0]
    ret_num_visited = len(times)
    return ret_path[::-1], ret_time, ret_num_visited
    # Begin your code (Part 6)
```

Part II. Results & Analysis

Part 5: Test my implementation

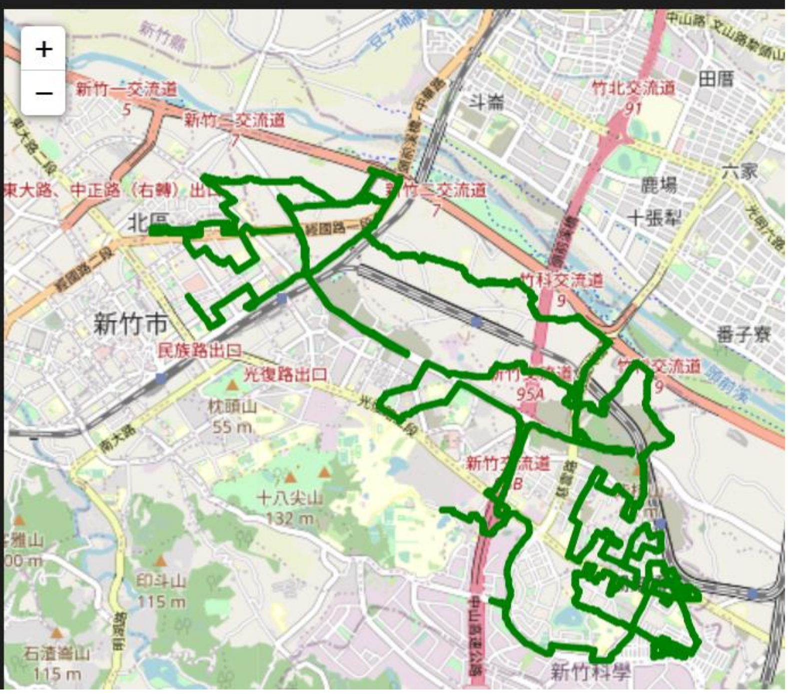
Test1: from National Yang Ming Chiao Tung University (ID: 2270143902) to Big City Shopping Mall (ID: 1079387396)

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: 12054
```

DFS (recursion) :

```
The number of nodes in the path found by DFS: 1311
Total distance of path found by DFS: 48954.32100000007 m
The number of visited nodes in DFS: 3514
```

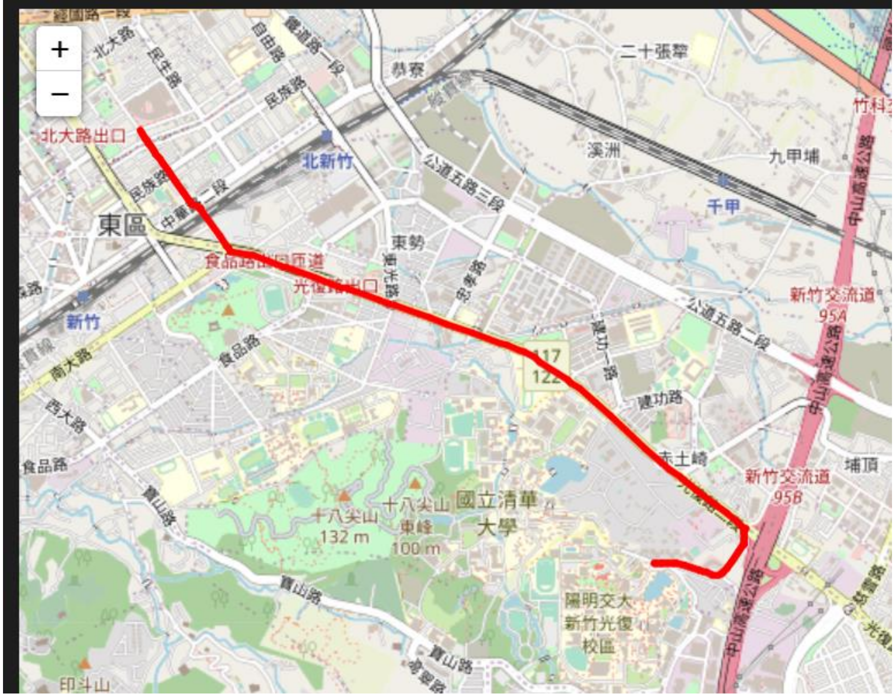


Uniform Cost Search :

```
The number of nodes in the path found by UCS: 89
Total distance of path found by UCS: 4367.881 m
The number of visited nodes in UCS: 5086
```

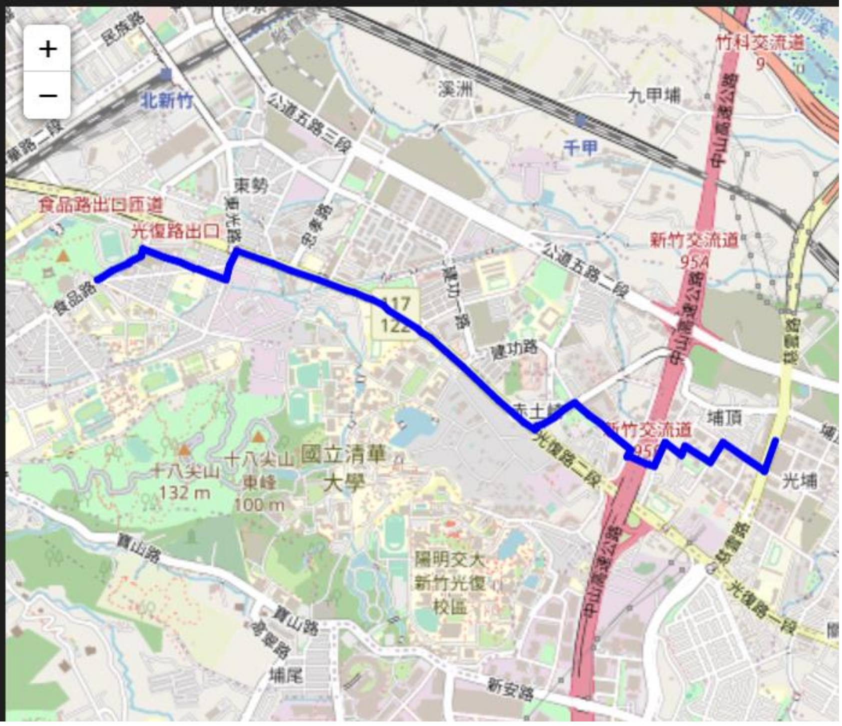
A* Search :

```
The number of nodes in the path found by A* search: 85
Total distance of path found by A* search: 4354.451000000003 m
The number of visited nodes in A* search: 318
```



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: 4215.521 m
The number of visited nodes in BFS: 12054
```

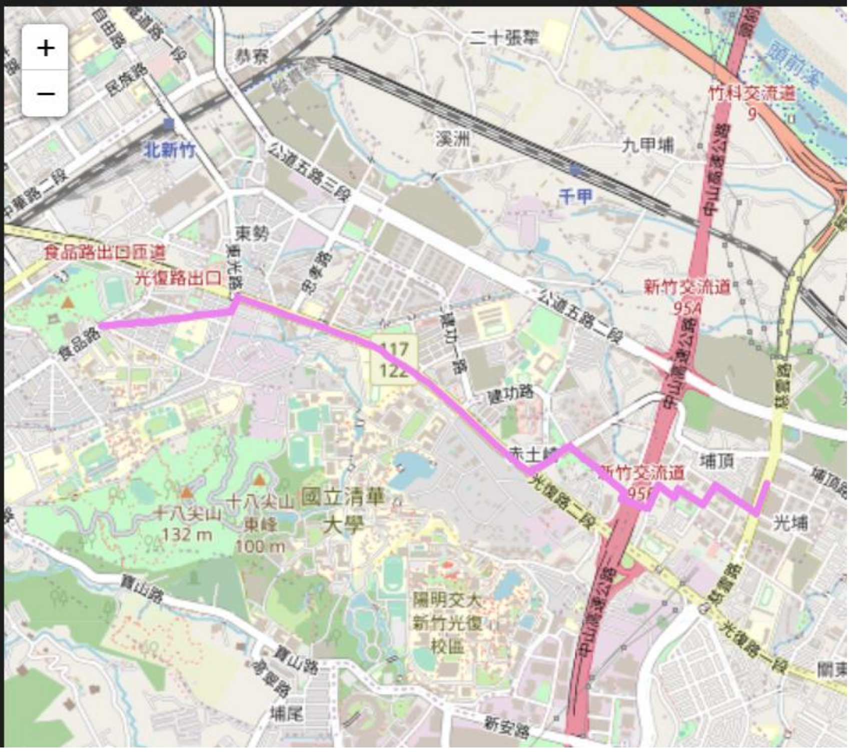


DFS (recursion) :

```
The number of nodes in the path found by DFS: 1016
Total distance of path found by DFS: 43504.768999999935 m
The number of visited nodes in DFS: 10610
```

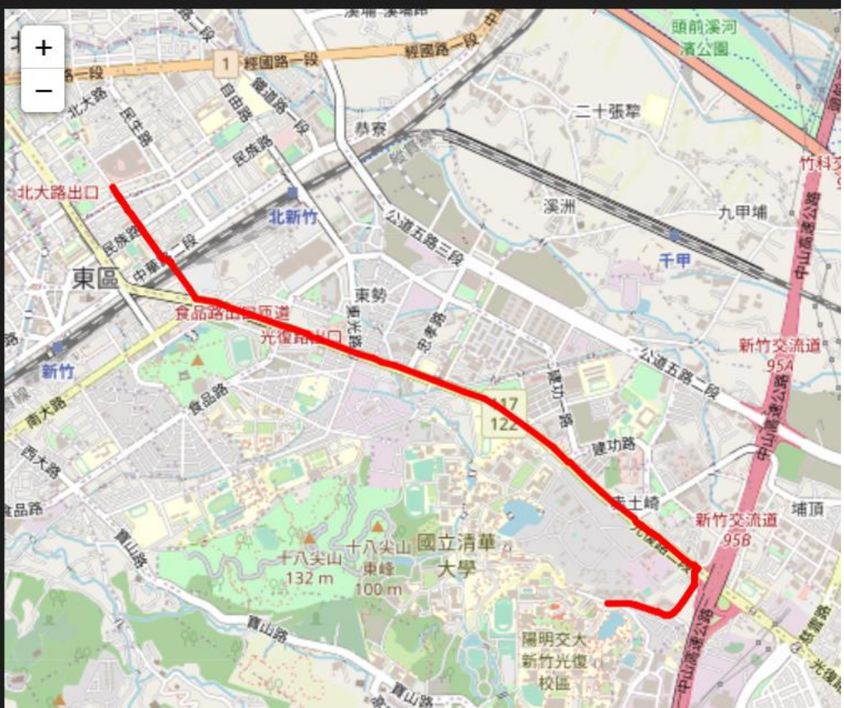
Uniform Cost Search :

```
The number of nodes in the path found by UCS: 63
Total distance of path found by UCS: 4101.84 m
The number of visited nodes in UCS: 7213
```



A* search :

```
The number of nodes in the path found by A* search: 89
Total distance of path found by A* search: 4367.881000000001 m
The number of visited nodes in A* search: 261
```



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

BFS :

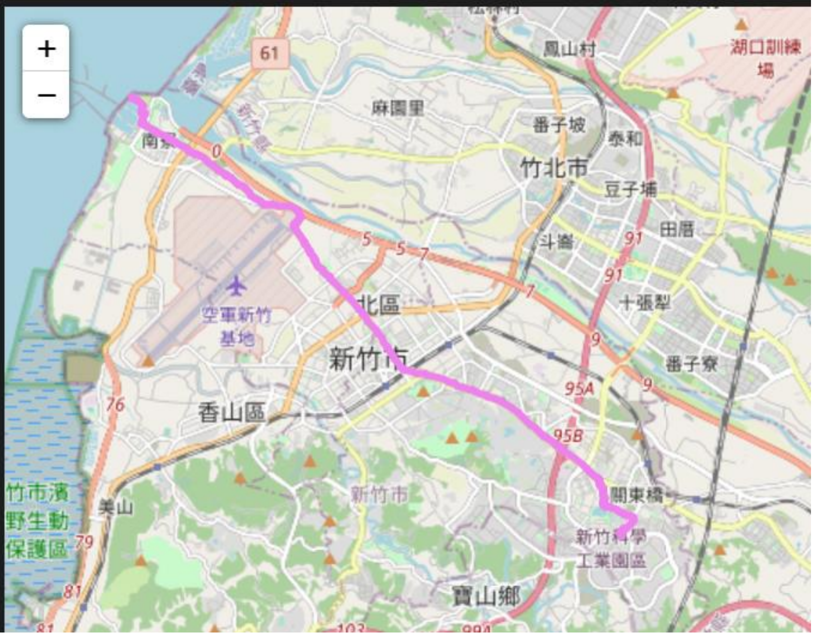
```
The number of nodes in the path found by BFS: 183
Total distance of path found by BFS: 15442.395000000002 m
The number of visited nodes in BFS: 12054
```

DFS (recursion) :

```
The number of nodes in the path found by DFS: 2635
Total distance of path found by DFS: 120440.44299999985 m
The number of visited nodes in DFS: 7510
```

Uniform Cost Search :

```
The number of nodes in the path found by UCS: 288
Total distance of path found by UCS: 14212.412999999997 m
The number of visited nodes in UCS: 11926
```



A* search :

```
The number of nodes in the path found by A* search: 288
Total distance of path found by A* search: 14212.413000000002 m
The number of visited nodes in A* search: 7073
```

Part 6: Search with a different heuristic (Bonus)

Test 1:

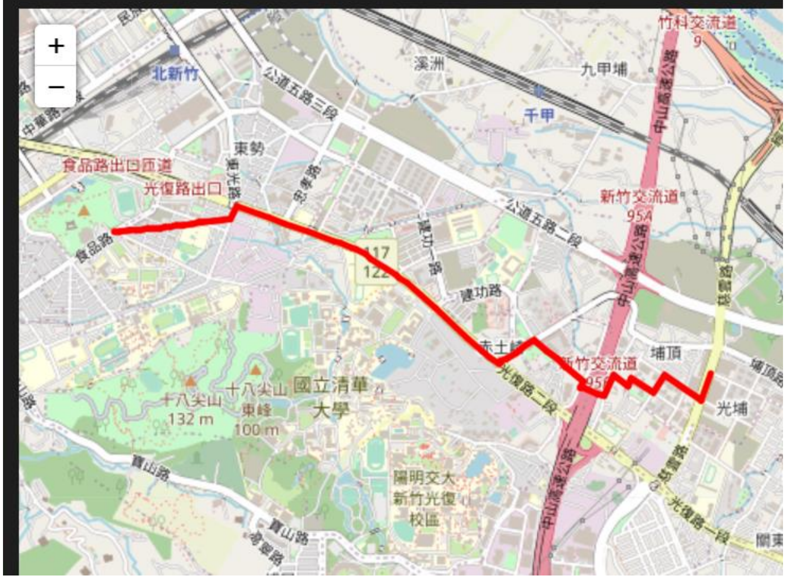
from National Yang Ming Chiao Tung University (ID: 2270143902) to Big City Shopping Mall (ID: 1079387396)

```
The number of nodes in the path found by A* search: 89
Total second of path found by A* search: 320.87823163083146 s
The number of visited nodes in A* search: 1934
```

Test 2:

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

```
The number of nodes in the path found by A* search: 63
Total second of path found by A* search: 304.4436634360301 s
The number of visited nodes in A* search: 2870
```



Test 3 :

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

```
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
```

Compare the results with results obtained in Part 4 :

Compared with the A* search algorithm we have done in part 4, this program calculates the total second of path. Thus, we have to consider the speed limit of every road. In this program, the number of visited nodes when searching is far bigger than the one of part 4's.

Part III. Question Answering

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

I had problem executing jupyter notebook in the beginning, my surrounding of my computer was not set up well so I can't execute it in Spyder successfully. Finally, I execute my programs on Visual Studio Code, and it work successfully.

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.

Besides speed limit and distance, I think time is also an attribute. There are different traffic flow between day and night. In the morning, people are likely to go to work, so traffic jam may occurs on the roads lead to these business district. So the route may no the shortest-distance path. Thus, we also have to consider what time it is to find a route in the real world.

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 mapping, we can get materials of map online and turn the whole map into a directed weighted map. Intersections become nodes and every road is an edge, the speed limit is inversely proportional to the weight of an edge.

For localization, we can use GPS system to confirm our current position precisely. Sometime it has to be connected to the internet to execute the system.

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 based on other attributes. Please define a dynamic heuristic function for ETA. Please explain the rationale of your design.

We can first construct a list that represents the distance of each intersection and the destination. Then as we heading for the destination, we calculate the time we need to reach the destination dynamically, which means we have to consider real-time condition of the path we will go through. It could contain many attributes such as weather, traffic flow, road accident etc.