Homework 2: Route Finding Report

0711257葉長瀚

Part I. Implementation:

Read edge.csv:

```
    '''
    This part is to input the edge data, which is all the same in the 5  route finding algorithms.
    Create a graph using the dictionary structure. The key can be one of the node in the map and
    will point to a numpy array that contain all edges connected and their detail information.
    '''
    fptr=open(edgeFile,'r')
    fptr.readline()
    graph={}
    neighbors=[]
    while 1:
        temp=fptr.readline()
        if not temp:
            break
        temp=temp.split(',')
        key=int(temp[0])
        if key not in graph.keys():
            neighbors=[]
        neighbors.append([int(temp[1]), float(temp[2])])
        graph[key]=neighbors
    fptr.close()
```

BFS:

```
    '''
    This is the bfs part. Use numpy array as a queue to store nodes, a set structure to
    record visited nodes and a dictionary to map the node with its parent node. When there's node
    in the queue, pop it out, and browse all the nodes connected to it at the same time if
    they are not visited nor the end node. Due to the first in first out property of queue,
    we can just push the node in directly to attain our goal. Finally, browse back from the end
    node to start node by the parent dictionary to store the whole path node as a set and also
    calculate the total distance of the path.
    '''
    n=0
    queue=[]
    queue.append(start)
    seen=set()
    seen.add(start)
    parent={start:None}
    while len(queue):
        node=queue.pop(0)
        n+=1
        nei=graph.get(node,-1)
        if nei==-1:
            continue
        for x in nei:
            if x[0]==end:
                n+=1
                parent[x[0]]=node
                break
            elif not (x[0] in seen):
                queue.append(x[0])
                seen.add(x[0])
                parent[x[0]]=node
        if (end in parent):
            break
    path=[]
    v=end
    dist=0
    while 1:
        path.insert(0,v)
        u=parent[v]
        # print(u, v)
        if u==None:
            break
        for x in graph.get(u):
            if x[0]==v:
                dist+=x[1]
                break
        v=u
    num_visited=n
    return path, dist, num_visited
```

DFS:

```
    '''
    This is dfs part. I do the iterative version by implementing the
    stack structure property. It's worth noticed that browsing the neighbor
    nodes with backward order might more fit the way how we demonstrate dfs
    on the real graph. The rest parts are just exaclty same as bfs.
    '''
    n=0
    stack=[]
    stack.append(start)
    seen=set()
    parent={start:None}
    while len(stack):
        node=stack.pop()
        if (node in seen):
            continue
        n+=1
        seen.add(node)
        nei=graph.get(node,-1)
        if nei==-1:
            continue
        for x in nei[::-1]:
            if x[0]==end:
                n+=1
                parent[end]=node
                break
            elif not(x[0] in seen):
                stack.append(x[0])
                parent[x[0]]=node
        if (end in parent):
            break
    path=[]
    v=end
    dist=0
    while 1:
        path.insert(0,v)
        u=parent[v]
        if u==None:
            break
        for x in graph.get(u):
            if x[0]==v:
                dist+=x[1]
                break
        v=u
    num_visited=n
    return path, dist , num_visited
```

UCS class for priority queue:

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

'''
I use the priority queue function in heapq library. Here is the class that i
put in the pq containing the id of the node and the cost to get to it, and also
redine the < operator for the Node class.
'''
class Node(object):
    def __init__(self, Id:int,val: float):
        self.val = val
        self.Id = Id

    def __repr__(self):
        return f'Node id: {self.Id} Node value: {self.val}'

    def __lt__(self, other):
        return self.val < other.val
```

UCS:

```python
    '''
    This is ucs part. Keep popping out the node that has the smallest cost to get to until
    we reach the end node or the pq is empty. Here we don't have to calculate the distance
    of the path, because it will just be equivalent to the end node's cost when we pop it
    from pq.
    '''
    n=0
    pq=[]
    heapq.heappush(pq,Node(start,0))
    seen=set()
    parent={Node(start,0):None}
    while len(pq):
        node=heapq.heappop(pq)
        if node.Id in seen:
            continue
        n+=1
        seen.add(node.Id)
        if node.Id==end:
            dist=node.val
            v=node
            break
        nei=graph.get(node.Id,-1)
        if nei==-1:
            continue
        for x in nei:
            if not (x[0] in seen) :
                temp=Node(x[0], x[1]+node.val)
                heapq.heappush(pq,temp)
                parent[temp]=node
    path=[]
    while v!=None:
        path.insert(0,v.Id)
        v=parent.get(v)
    num_visited=n
    return path, dist, num_visited
```

Astar(distance) class:

```
'''
Class in astar is modified to contain 3 variables, which are id, cost(dist) and
cost+heuristic(dist from start to node + euclidean dist from node to end)
'''
class Node(object):
    def __init__(self, Id:int,d:float,val: float):
        self.val = val
        self.Id = Id
        self.d = d

    def __repr__(self):
        return f'Node id: {self.Id} Node value: {self.val}'

    def __lt__(self, other):
        return self.val < other.val
```

Read Heuristic.csv:

```
    '''
    This part is to input the Euclidean distance data, which will be use in
    both astar(distance) and astar(time) version. I store the data in a dictionary
    of another dictionary. The first key is the id of the end node, and the second
    key is the node id where I want to start from. By this structure, I dont have to
    change anything except for the input arguments of the function.
    '''
    heu={}
    fptr=open(heuristicFile,'r')
    nID=fptr.readline().split(',')
    for i in range(1,4):
        heu[int(nID[i])]={}

    while 1:
        temp=fptr.readline()
        if not temp:
            break
        temp=temp.split(',')
        for i in range(1,4):
            heu[int(nID[i])][int(temp[0])]=float(temp[i])
    fptr.close()
```

Astar(distance):

```python
    '''
    This is the astar(distance) part, which is exactly the same as the ucs one, except
    that we compare two Nodes by dist from start to node + euclidean dist from node to end.
    '''
    n=0
    pq=[]
    heapq.heappush(pq,Node(start,0,heu[end][start]))
    seen=set()
    parent={Node(start,0,heu[end][start]):None}
    while len(pq):
        node=heapq.heappop(pq)
        # print(node)
        if node.Id in seen:
            continue
        n+=1
        if node.Id==end:
            dist=node.d
            v=node
            break
        seen.add(node.Id)
        nei=graph.get(node.Id,-1)
        if nei==-1 :
            continue
        for x in nei:
            if not(x[0] in seen):
                temp=Node(x[0],node.d+x[1],node.d+x[1]+heu[end][x[0]])
                heapq.heappush(pq,temp)
                parent[temp]=node

    path=[]
    while v!=None:
        path.insert(0,v.Id)
        v=parent.get(v)
    num_visited=n
    return path, dist, num_visited
```

Astar(time) class:

```python
'''
Class in astar(time) is modified to contain 3 variables, which are id,
cost(time spend from start to node) and
heuristic(time spend from node to end by euclidean dist and the max speed limit of the map)
'''
class Node(object):
    def __init__(self, Id:int,t:float,hVal: float):
        self.t = t
        self.Id = Id
        self.hVal = hVal

    def __repr__(self):
        return f'Node id: {self.Id} Node t: {self.t} Node g+h: {self.hVal+self.t}'

    def __lt__(self, other):
        return self.t+self.hVal < other.t+other.hVal
```

Astar(time):

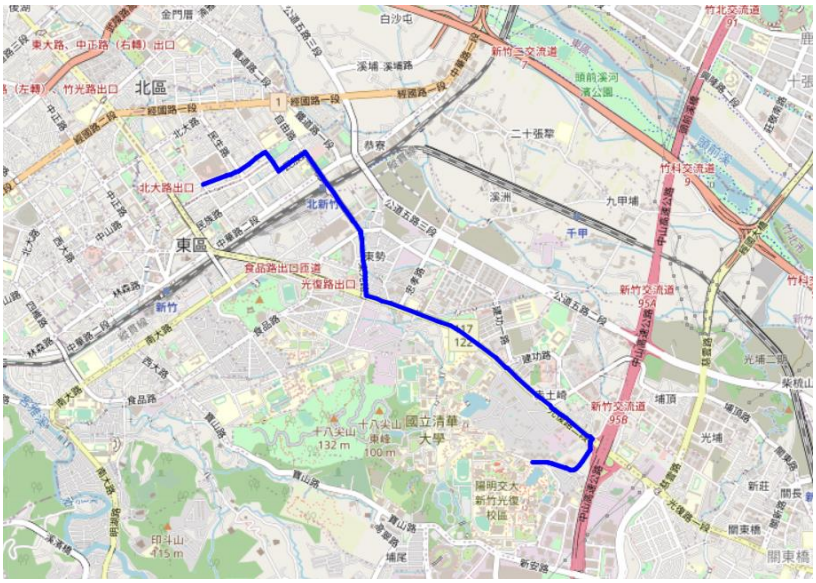
```
    '''
    This is astar(time) part. The comparison factor become
    cost(time spend from start to node) +
    heuristic(time spend from node to end by euclidean dist and the max speed limit of the map)
    here. For the Heuristic part, I calculate the time this way to ensure that its admissible,
    and we can get the max speed limit while inputting the edges data.
    '''
    n=0
    pq=[]
    heapq.heappush(pq,Node(start,0,heu[end][start]/speed))
    seen=set()
    parent={Node(start,0,heu[end][start]/speed):None}
    while len(pq):
        node=heapq.heappop(pq)
        # print(node)
        if node.Id in seen:
            continue
        n+=1
        if node.Id==end:
            time=node.t
            time*=3.6
            v=node
            break
        seen.add(node.Id)
        nei=graph.get(node.Id,-1)
        if nei==-1 :
            continue
        for x in nei:
            if not(x[0] in seen):
                temp=Node(x[0],(node.t+x[1]/x[2]),heu[end][x[0]]/speed)
                heapq.heappush(pq,temp)
                parent[temp]=node

    path=[]
    while v!=None:
        path.insert(0,v.Id)
        v=parent.get(v)
```

Part II. Results & Analysis:

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

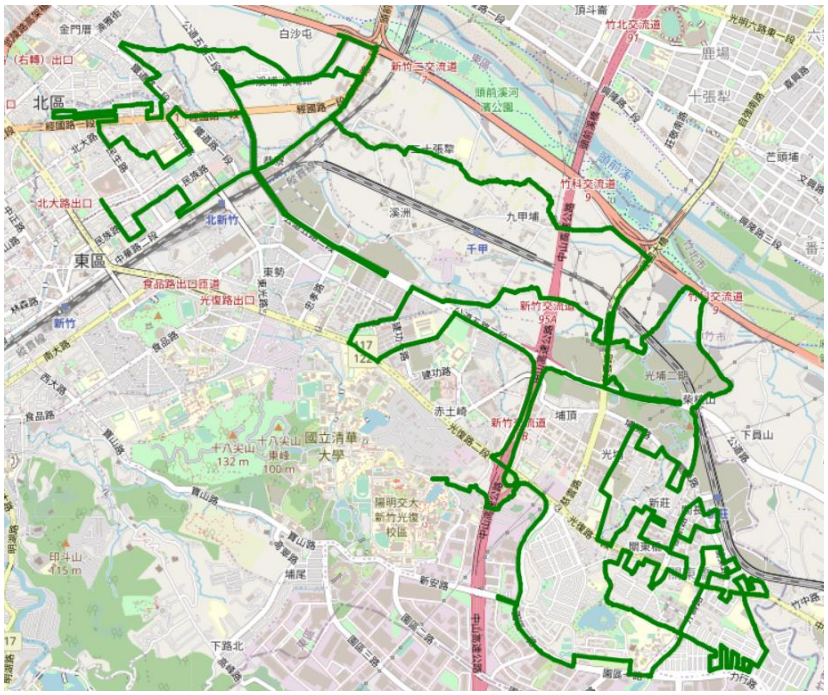
BFS:

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: 4131



DFS(iterative):

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

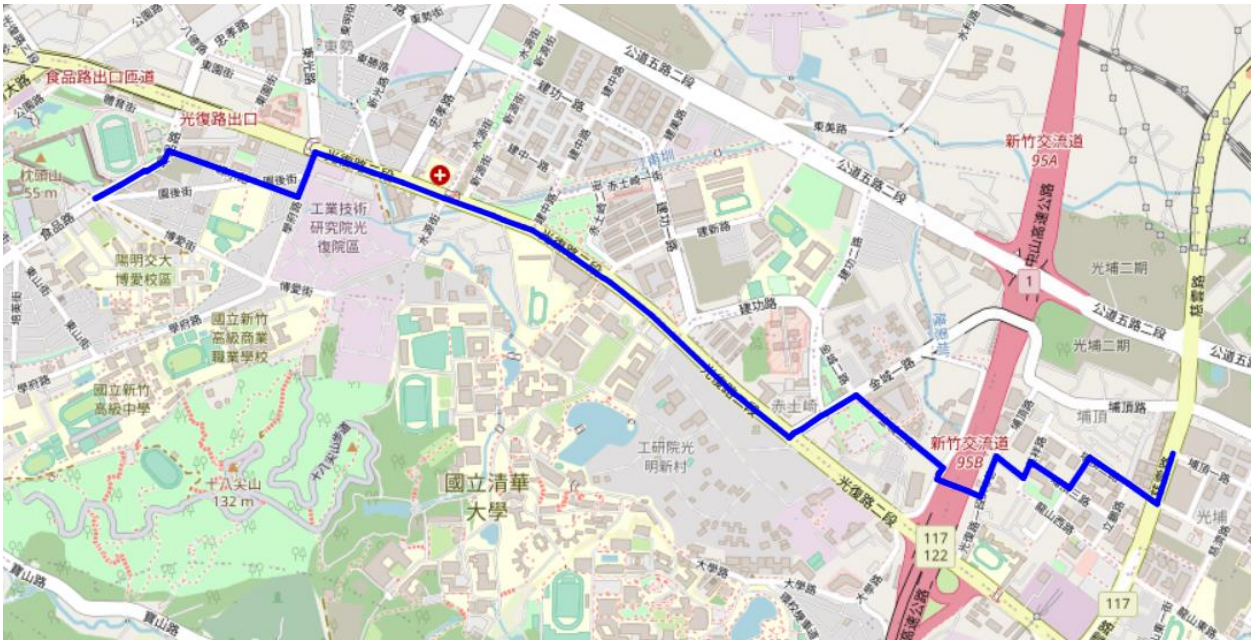


UCS:
The number of nodes in the path found by UCS: 89
Total distance of path found by UCS: 4367.881 m
The number of visited nodes in UCS: 5086

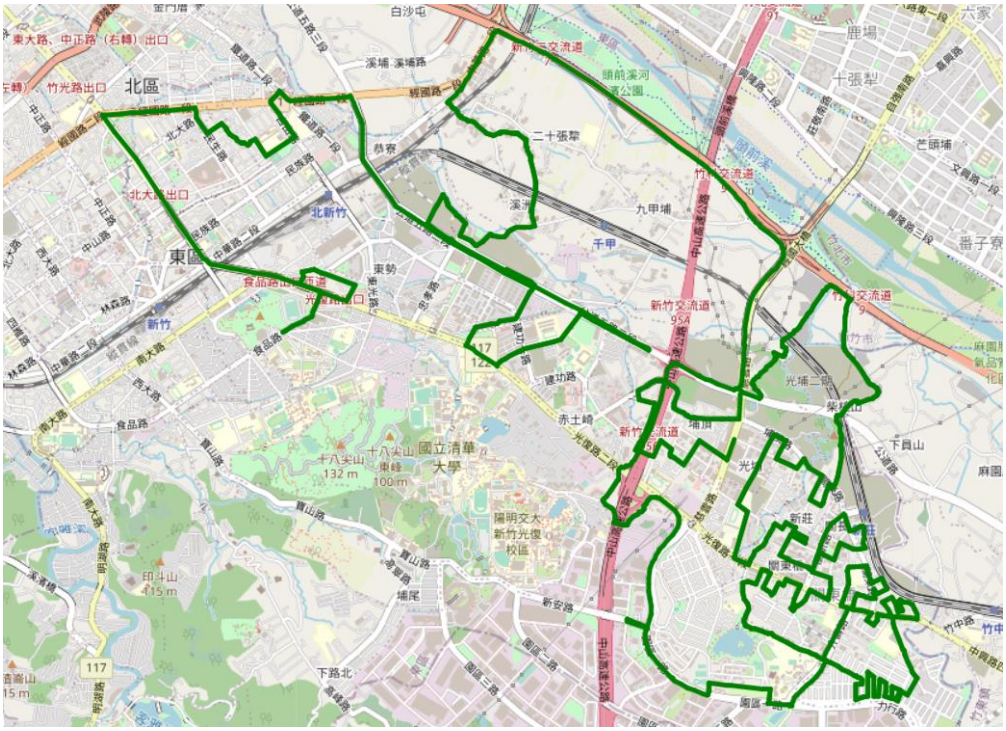
Astar(distance):
The number of nodes in the path found by A* search: 89
Total distance of path found by A* search: 4367.881 m
The number of visited nodes in A* search: 261

Astar(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: 790

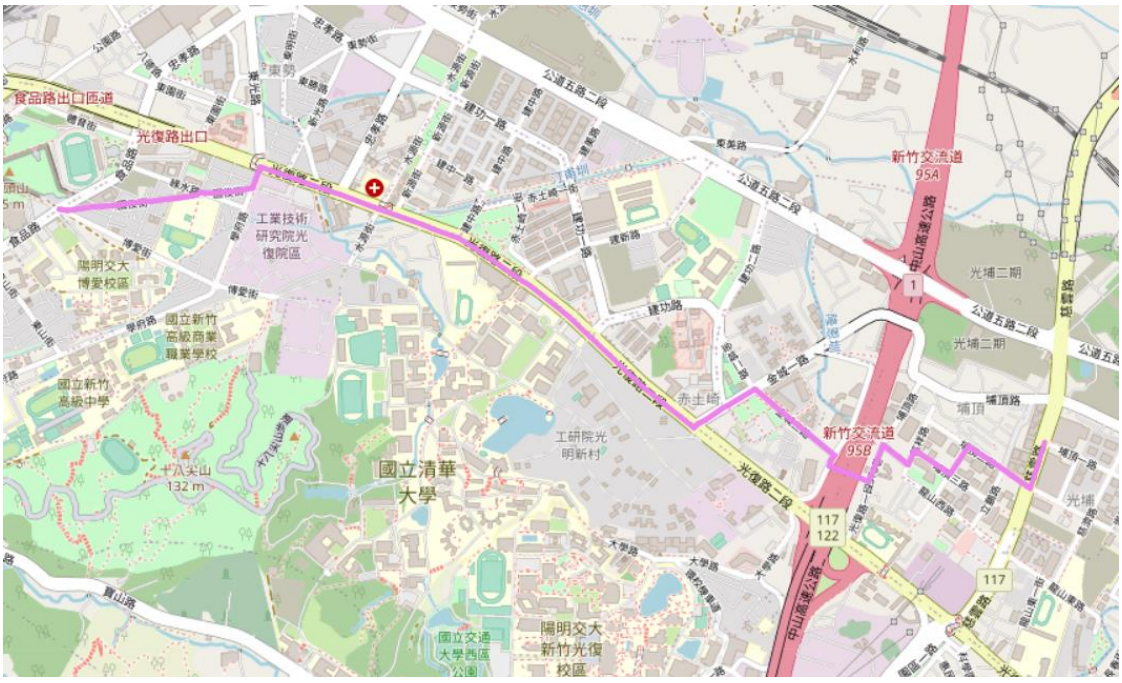
Test 2: from **Hsinchu Zoo** (ID: 426882161) to **COSTCO Hsinchu Store** (ID: 1737223506)
BFS:
The number of nodes in the path found by BFS: 60
Total distance of path found by BFS: 4215.521000000001 m
The number of visited nodes in BFS: 4469



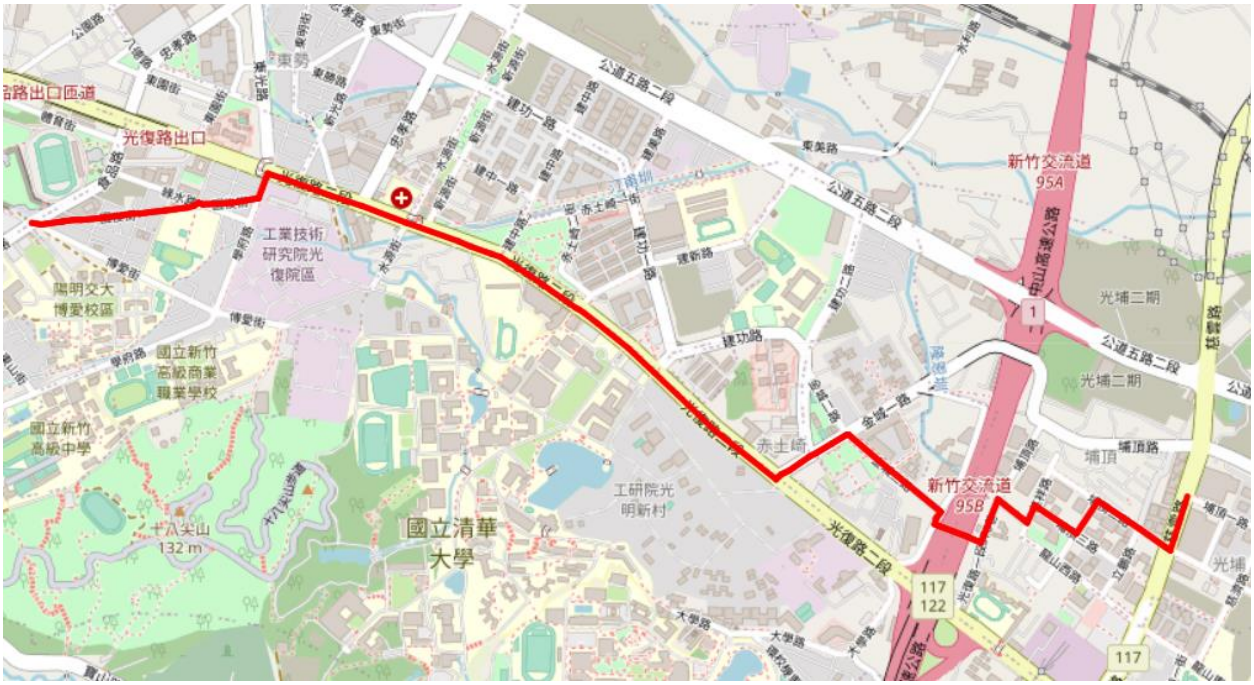
DFS(iterative):
The number of nodes in the path found by DFS: 1016
Total distance of path found by DFS: 43504.76899999997 m
The number of visited nodes in DFS: 10623



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: 7213



Astar(distance):
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

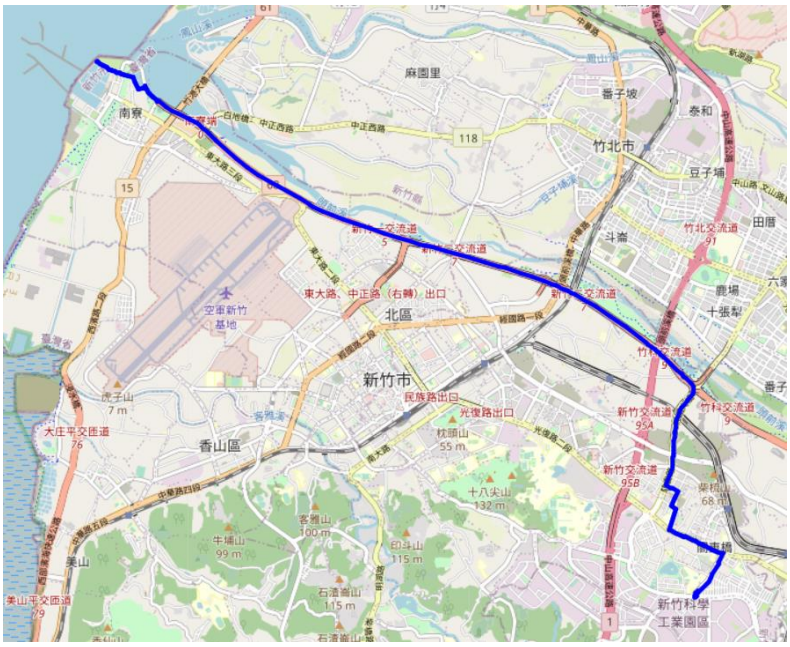


Astar(time):
The number of nodes in the path found by A* search: 63
Total second of path found by A* search: 304.44366343603014 s
The number of visited nodes in A* search: 1604

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

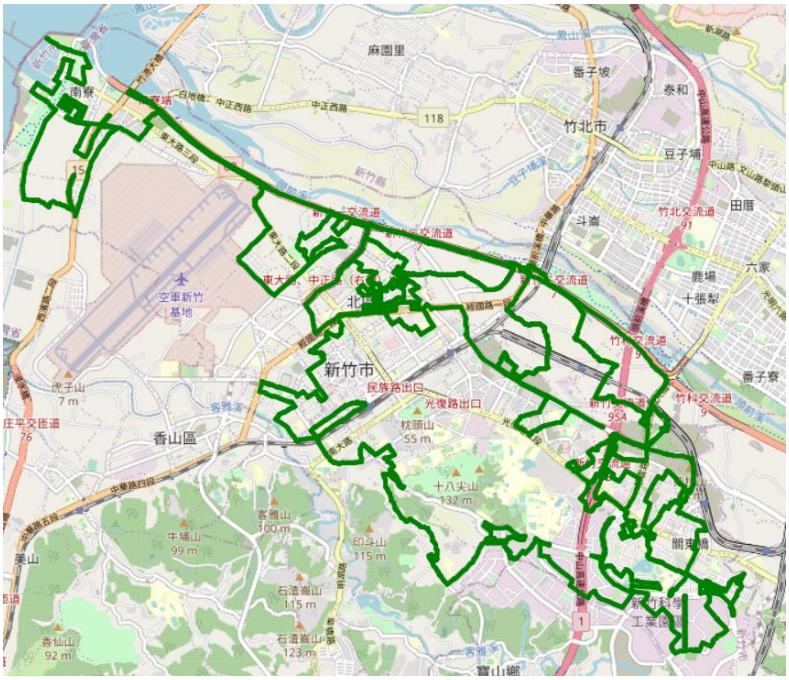
BFS:

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: 11217

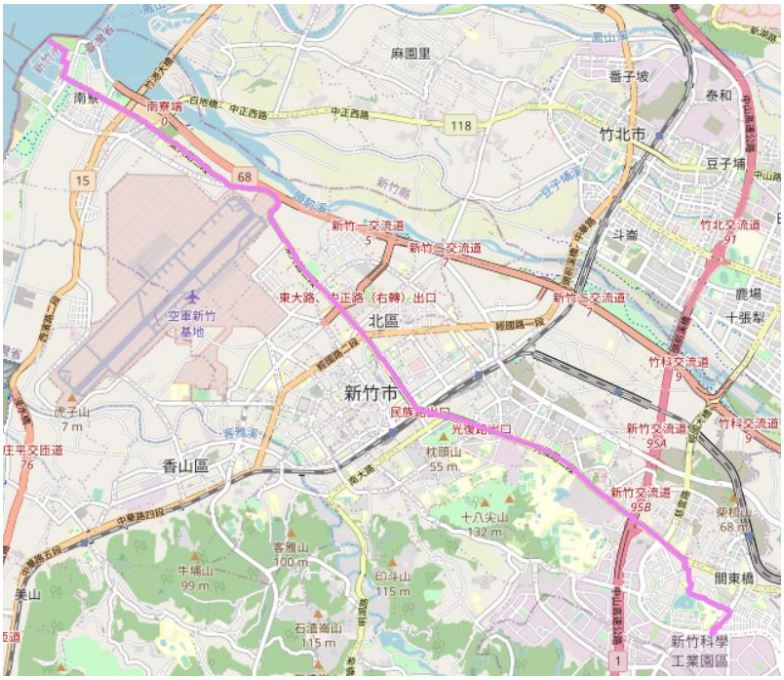


DFS(iterative):

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

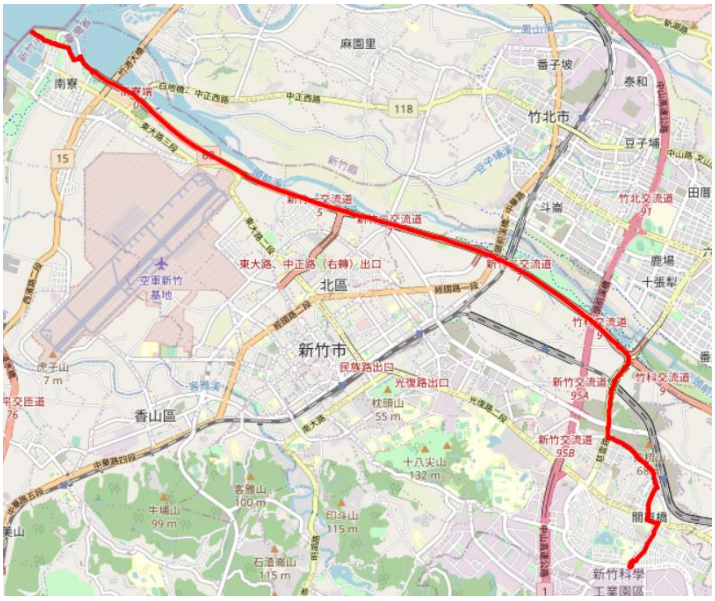


UCS:
The number of nodes in the path found by UCS: 288
Total distance of path found by UCS: 14212.412999999997 m
The number of visited nodes in UCS: 11926



Astar(distance):
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

Astar(time):
The number of nodes in the path found by A* search: 209
Total second of path found by A* search: 779.5279228368482 s
The number of visited nodes in A* search: 8000



From the results, we can see that UCS can get the same shortest path with Astar (distance), but the latter need much fewer visited nodes to figure out the same result, because we take another factor (heuristic function) into consideration, which will reduce the time of route finding. For the comparison between Astar (distance) and Astar (time), the former get the path with the shortest distance while the latter get one with the shortest consuming time, since these 2 are different quantities, I think we can't easily say that which is better, but we can tell that if today one is in hurry to get to the destination than Astar (time) will satisfy him whereas one wants to cut down the gasoline usage than Astar (distance) will be a better choice.

Part III. Answer the questions:

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

```
32      parent={start:None}
56      while 1:
57          path.insert(0,v)
58          u=parent[v]
59          if u==None:
60              break
```

In both bfs and dfs function, I use u=parent[v] to get the dictionary value.

```
49      parent={Node(start,0):None}
69      while v!=None:
70          path.insert(0,v.Id)
71          v=parent.get(v)
```

However, when it comes to ucs part, I couldn't get the None value correctly with the former way, but it works when I use *v=parent.get(v)*. I am still confused of this now…

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.

I think the average time that one will have to stay in a road to wait for the light green is also important, especially when he is commuting in the city. For example, if it costs 15 minute of ride to get to the destination, I believe time spent on waiting for the traffic light will definitely account for at least 2 minute.

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:
For a given set of vertices, we wish to summarize them as a set of landmarks. While summarizing the landmarks, we consider the following four rules.

- Geometric Representation: Landmarks must be well distributed geometrically
- Visual Representation: Landmarks must be useful for localizing images using their visual features
- Navigation Assurance: Landmarks must support navigation from any source to any target location, using only visual features
- Map Compactness: The number of landmarks must be small

Localization:
Given a sequence of images and landmarks along a path, the task of self-localization is equivalent to finding the most consistent match. We assume that an ordered sequence of images captured along a path are given. We wish to localize these images by matching them to landmarks. We formulate self-localization as a graph matching problem. For the purpose of self-localization, we want the matching process to favor the following two rules.

- Visual Matching: The visual distance between matched pairs must be minimized
- Geometric Matching: Neighbours of vertex must be matched to the neighbours of itself

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.

Heuristic function = (avg. time for waiting traffic light + dist. from current place to the destination/ max speed)
As answered in question 2, I think time spent on traffic light should be considered if want to be more accurate. Also, the max speed here has to vary with present time, because whether it's rush hour or not will definitely affect the speed during commuting.