Homework 2: Route Finding

Part I. Implementation (6%):

Part1:

```python
import csv
edgeFile = 'edges.csv'
"""
Save the file 'edges.csv' into the dictionary "graph".
Every row may like [start, end, dist, limit].
Store it into dictionary as: 'start':[end, distance]
"""
graph={}
with open(edgeFile, newline='') as file:
    csvfile = list(csv.reader(file))
    csvfile.pop(0)
    for i in csvfile:
        num1, num2, distance = int(i[0]), int(i[1]), float(i[2])
        if num1 not in graph:
            graph[num1] = [(num2, distance)]
        else:
            graph[num1].append((num2, distance))
        if num2 not in graph:
            graph[num2] = []
```

```python
def bfs(start, end):
    # Begin your code (Part 1)
    """
    Build a queue to store the path with [current_vertex, [path]].
    Every time pop the fisrt element in queue and find its neighbor nodes.
    If we haven't visited the node, cnt++ and append it into queue.
    If the node == end, then we get the path from start to end and we can calculate the distance.
    Add the distance corresponding to the dictionary 'graph', and add each distance one by one.
    After all, we return the path, dist, and cnt.
    """
    dist=0
    cnt=1
    queue=[(start, [start])]
    visited=set()
    while queue:
        vertex, path = queue.pop(0)
        visited.add(vertex)
        for node in graph[vertex]:
            if node[0] == end:
                path += [end]
                st=start
                for v in path:      # st→v
                    if v==st:   continue
                    for n in graph[st]:
                        if n[0]==v:
                            dist += n[1]
                            break
                    st=v
                return path, dist, cnt
            else:
                if node[0] not in visited:
                    cnt+=1
                    visited.add(node[0])
                    queue.append((node[0], path+[node[0]]))

    # raise NotImplementedError("To be implemented")
    # End your code (Part 1)
```

Part2:

```python
import csv
edgeFile = 'edges.csv'
"""
Save the file 'edges.csv' into the dictionary "graph".
Every row may like [start, end, dist, limit].
Store it into dictionary as: 'start':[end, distance]
"""
graph={}
with open(edgeFile, newline='') as file:
    csvfile = list(csv.reader(file))
    csvfile.pop(0)
    for i in csvfile:
        num1, num2, distance = int(i[0]), int(i[1]), float(i[2])
        if num1 not in graph:
            graph[num1] = [(num2, distance)]
        else:
            graph[num1].append((num2, distance))
        if num2 not in graph:
            graph[num2] = []
```

```python
def dfs(start, end):
    # Begin your code (Part 2)
    """
    Build a stack to store the path with [current_vertex, [path]].
    Every time we pop the last element in queue and get the vertex.
    Check if we have visited the vertex before, skip if we have visited.
    If the vertex is not the end, cnt++ and get the neighbor of the node and append it to the stack.
    If the vertex == end, then we get the oath from start to end and we can calculate the distance.
    Add the distance corresponding to the dictionary 'graph', and add each distance one by one.
    After all, we return the path, dist, and cnt.
    """
    dist=0
    cnt=0
    stack = [(start, [start])]
    visited = set()
    while stack:
        vertex, path = stack.pop()
        if vertex not in visited:
            if vertex == end:
                st=start
                for v in list(map(int, path)):      # st→v
                    if v==st:   continue
                    for n in graph[st]:
                        if n[0]==v:
                            dist += n[1]
                            break
                    st=v
                return path, dist, cnt
            visited.add(vertex)
            cnt+=1
            for node in graph[vertex]:
                stack.append((node[0], path+[node[0]]))
    # raise NotImplementedError("To be implemented")
    # End your code (Part 2)
```

Part3:

```python
import csv
import heapq
edgeFile = 'edges.csv'
"""
Save the file 'edges.csv' into the dictionary "graph".
Every row may like [start, end, dist, limit].
Store it into dictionary as: 'start':[end, distance]
"""
graph = {}
with open(edgeFile) as file:
    csvfile = list(csv.reader(file))
    csvfile.pop(0)
    for i in csvfile:
        num1, num2, distance = int(i[0]), int(i[1]), float(i[2])
        if num1 not in graph:
            graph[num1] = [(num2, distance)]
        else:
            graph[num1].append((num2, distance))
        if num2 not in graph:
            graph[num2] = []
```

```python
def ucs(start, end):
    # Begin your code (Part 3)
    """
    Build the dictionary "parent" to store the parent of store, in order to trace back the path at last.
    Build a heapified list like priority queue called "heap" storing tuple (dist, node, parent) in the heap.
    Pop the first tuple of the heap.
    If the node haven't been visited, cnt++, record the node's parent, and get the node's neighbor and push into heap.
    If the node == end, then we find the path and the cost(distance).
    According to "parent", we record the path into "path" and return the path, dist, cnt.
    """
    parent = {}
    heap = [(0, start, None)]
    heapq.heapify(heap)
    visited = set()
    cnt, isfind, dist = 0, 0, 0
    while heap:
        (cost, node, p) = heapq.heappop(heap)
        if node == end:
            dist = cost
            parent[node] = p
            path=[]
            path.append(end)
            while path[-1]!=start:
                path.append(parent[path[-1]])
            path.reverse()
            return path, dist, cnt
        if node not in visited:
            cnt+=1
            visited.add(node)
            parent[node] = p
            for en, dis in graph[node]:
                heapq.heappush(heap, (cost + dis, en, node))
    # raise NotImplementedError("To be implemented")
    # End your code (Part 3)
```

Part4:

```
import csv
edgeFile = 'edges.csv'
heuristicFile = 'heuristic.csv'
from queue import PriorityQueue
"""
Read the file 'edges.csv' and convert to list and store it into "edges_list", then make a copy called "all_rows".
Pop out the first title row and change the other data into following format:
[start, end, dist, limit, which round, parent_start, parent_end, h()]
"""
with open(edgeFile, newline='') as file: # open edge data
        edge_list = list(csv.reader(file))
        all_rows = edge_list
        all_rows.pop(0) #get rid of the first line
        for r in all_rows:
            # each row : [start,end,distance,speed limit,found @ which round, parent start, parent end, h]
            r[0] = int(r[0])
            r[1] = int(r[1])
            r[2] = float(r[2])
            r.append(int(0))
            r.append(int(0))
            r.append(int(0))
            r.append(int(0))
```

```
def astar(start, end):
    # Begin your code (Part 4)
    """
    Because of different end node, we use 3 key to correspond to 'heuristic.csv'
    Build a list to store h(n) of 3 different end, and append the value of [node, h()]
    Update the h data in 'edge_list'.
    Build a priority queue "bfs_q" and put the [dist+h(), row] of start.
    """
    cnt=0
    node_dist=[]        # h(n)
    if end==1079387396: key=1
    elif end==1737223506:   key=2
    elif end==8513026827:   key=3

    with open(heuristicFile, newline='') as f:
        d=csv.reader(f)
        n=list(d)
        n.pop(0)
        for row in n:
            node_dist.append([int(row[0]),float(row[key])])
    for r in edge_list:
        for n in node_dist:     #n: node, h()
            if n[0]==r[1]:
                r[7] = n[1]
                break

    bfs_q = PriorityQueue()     # g+h, (start, end, dist, limit)
    for r in edge_list:
        if r[0]==start and r[4]==0:
            cnt+=1
            r[4]=1
            bfs_q.put([r[2]+r[7], r])
```

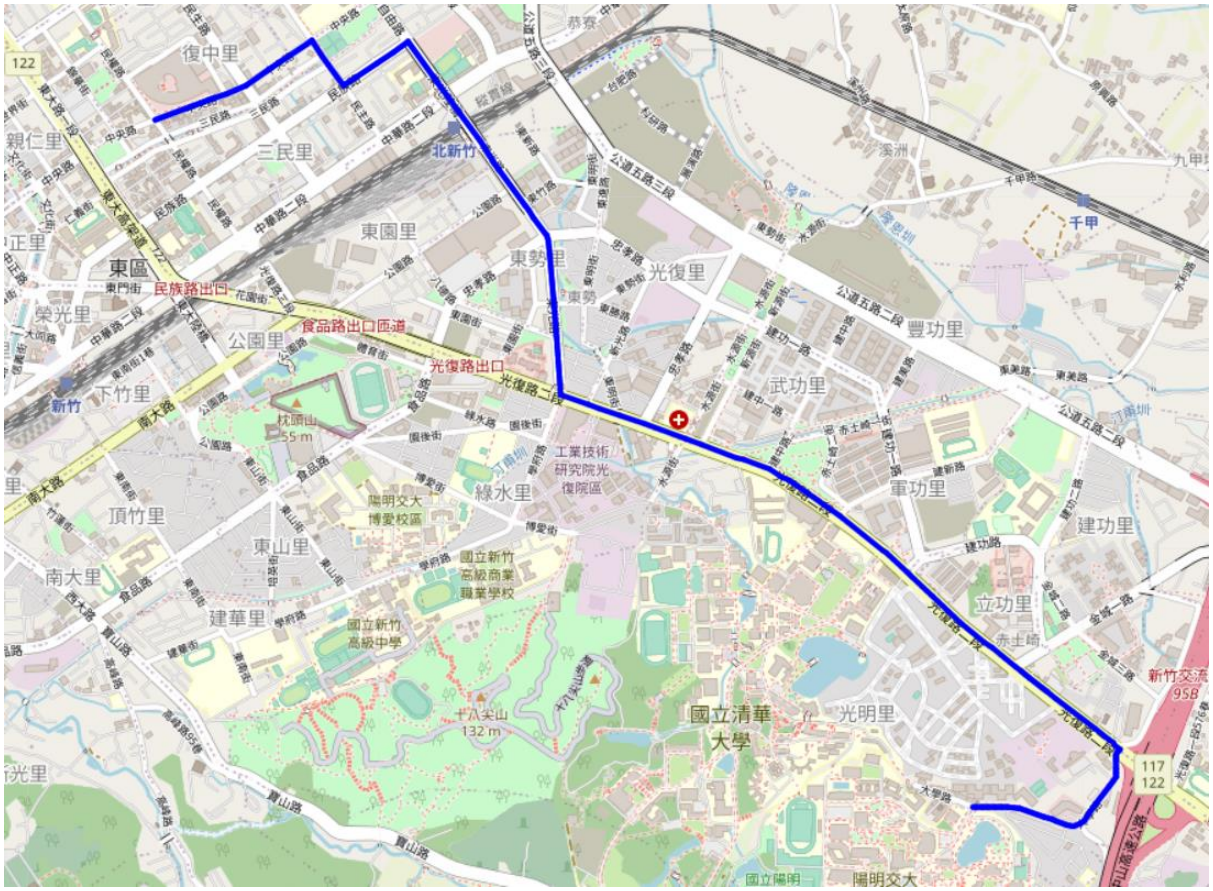
```
"""
While bfs_q is not empty and we haven't found the end:
    Get the highest priority element from "bfs_q" and get the current location.
    Found the location_node in "edge_list" and record the found_round, parent_start, parent_end.
    And then put the new node [original_priority-h(parent)+new_dist+new_h, r] into priority_queue.
    If we find the end then we record the cnt and break.
"""
dest=[]
find=False
while (not bfs_q.empty()) and find==False:
    c = bfs_q.get()
    cur = c[1]
    location = cur[1]

    for r in edge_list:
        if r[0]==location and r[4]==0:
            cnt+=1
            r[4]=cur[4]+1
            r[5]=cur[0]
            r[6]=cur[1]
            bfs_q.put([(c[0]-cur[7]+r[2]+r[7]),r])
            if r[1]==end:
                dest = r
                find = True
                break
"""
```

```
"""
We have recorded the dest so that we can trace back the path by parent_start and parent_end.
After we trace back to start, we append only the node number of the node into "path" and calculate the distance.
After all, return the path, dist, cnt.
"""
d = [dest]
curr = dest
while curr[0]!=start:
    for r in edge_list:
        if r[0]==curr[5] and r[1]==curr[6]:
            d.append(r)
            curr=r
            break
d.reverse()
path=[start]
dist=0
for r in d:
    path.append(r[1])
    dist+=r[2]
return path, dist, cnt
# raise NotImplementedError("To be implemented")
# End your code (Part 4)
```

Part II. Results & Analysis (12%):

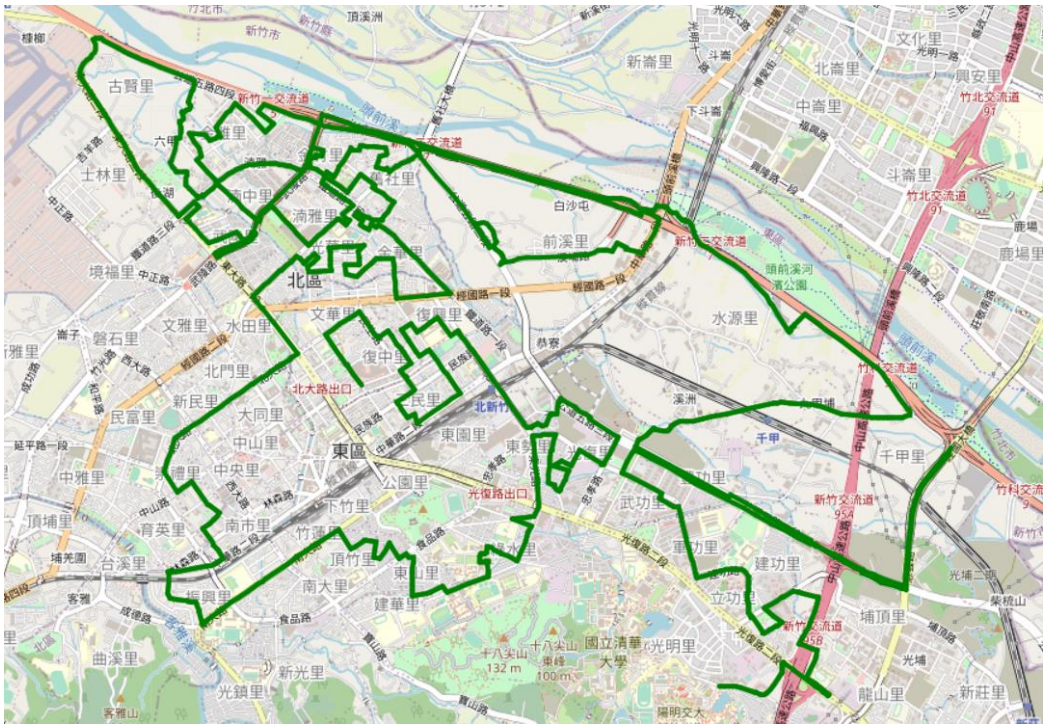
Test1: from National Yang Ming Chiao Tung University (ID: 2270143902)
to Big City Shopping Mall (ID: 1079387396)
BFS:

```
The number of 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
```



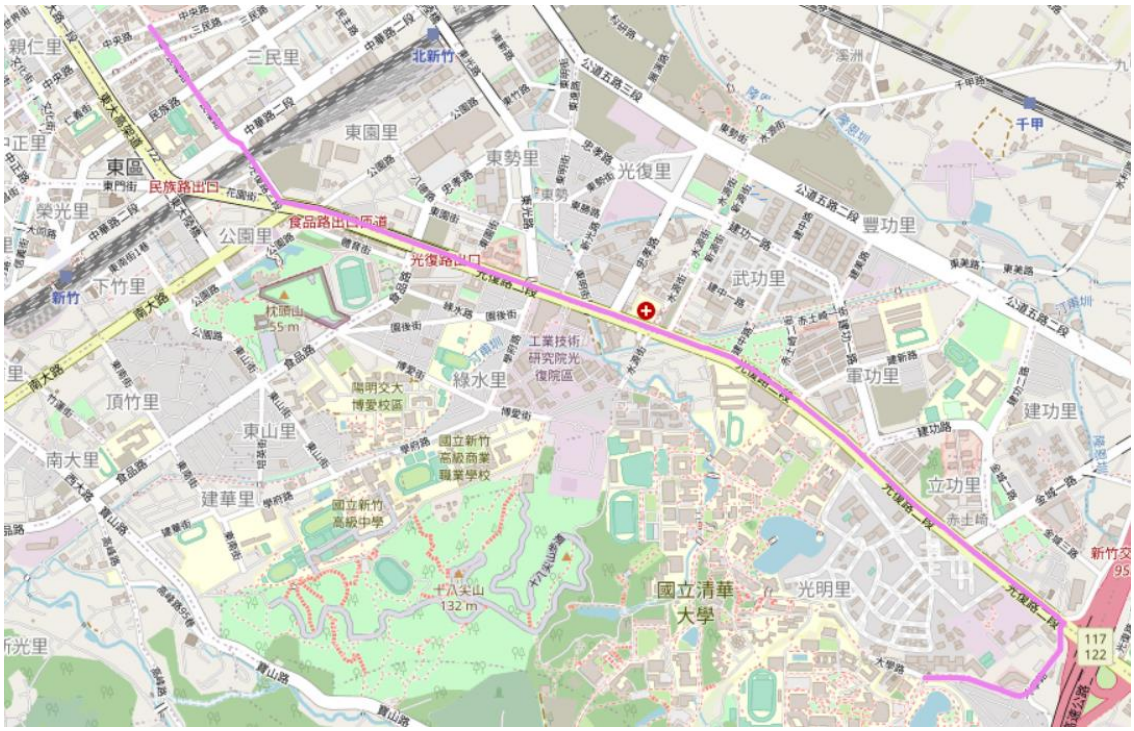
DFS (stack):

```
The number of nodes in the path found by DFS: 1232
Total distance of path found by DFS: 57208.987000000045 m
The number of visited nodes in DFS: 4210
```



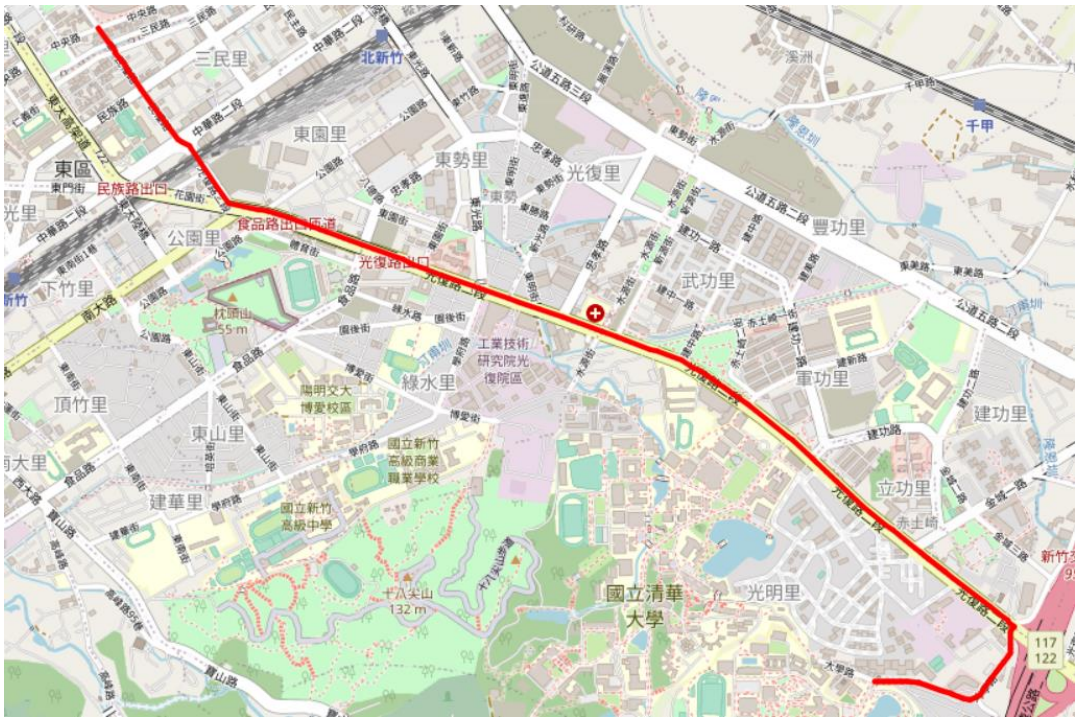
UCS:

```
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: 5085
```



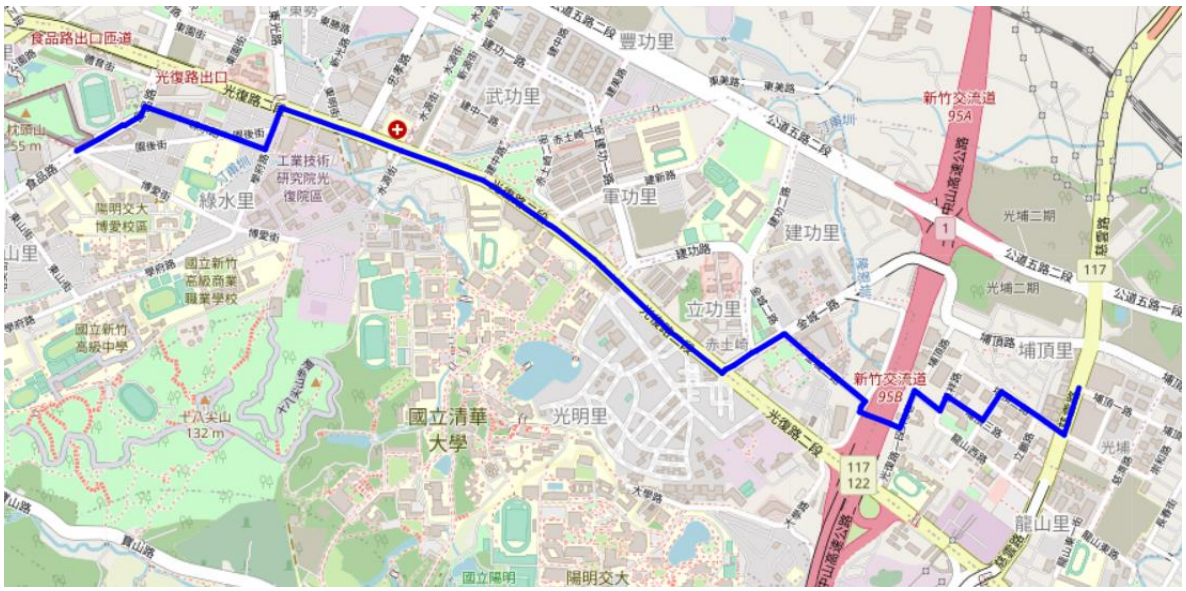
A*:

```
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: 523
```



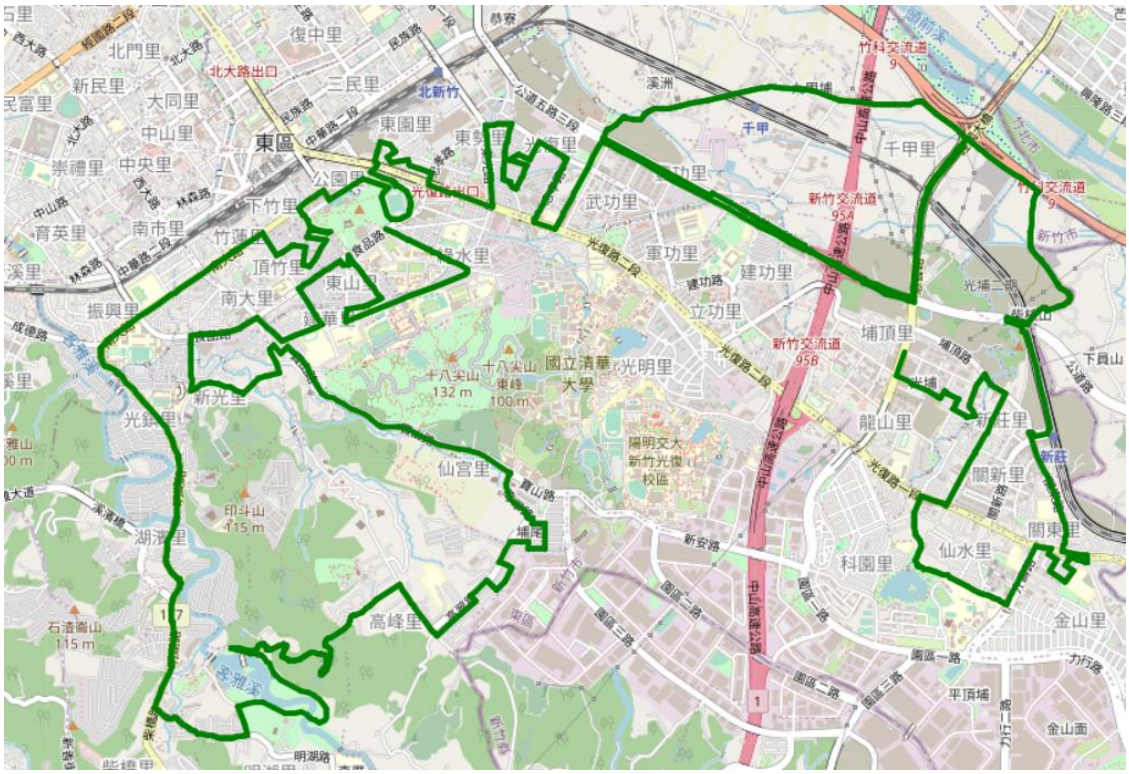
Test2: from Hsinchu Zoo (ID: 426882161)
to COSTCO Hsinchu Store (ID: 1737223506)
BFS:

```
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: 4606
```



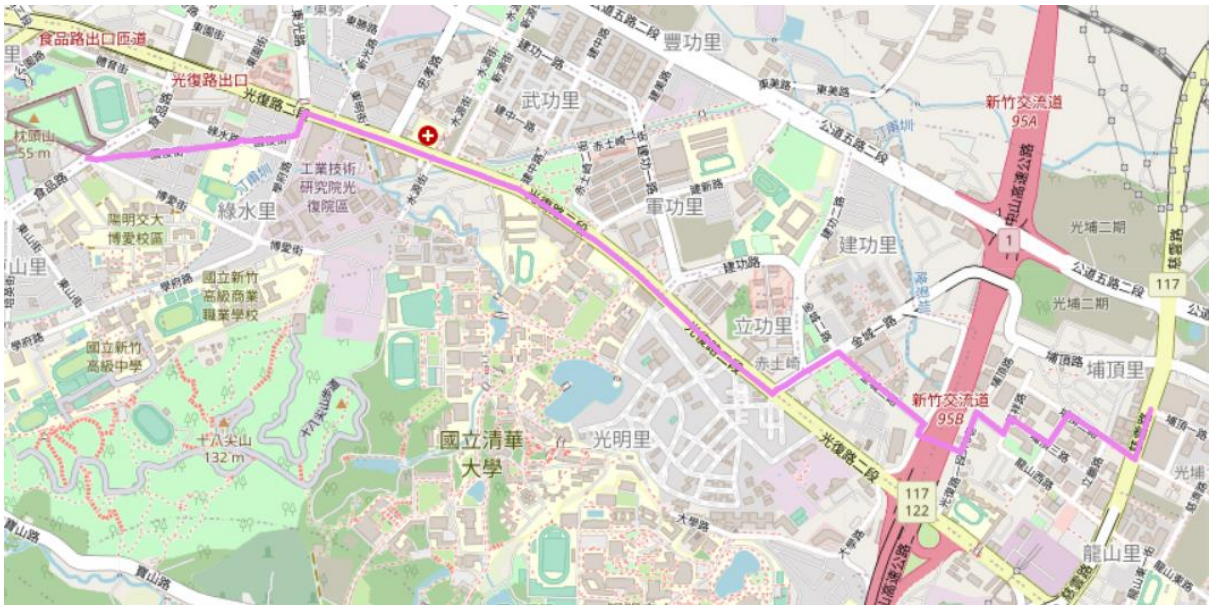
DFS (stack):

```
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: 8030
```



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: 7212
```



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: 2429
```

Test3:

from National Experimental High School At Hsinchu Science Park (ID: 1718165260)

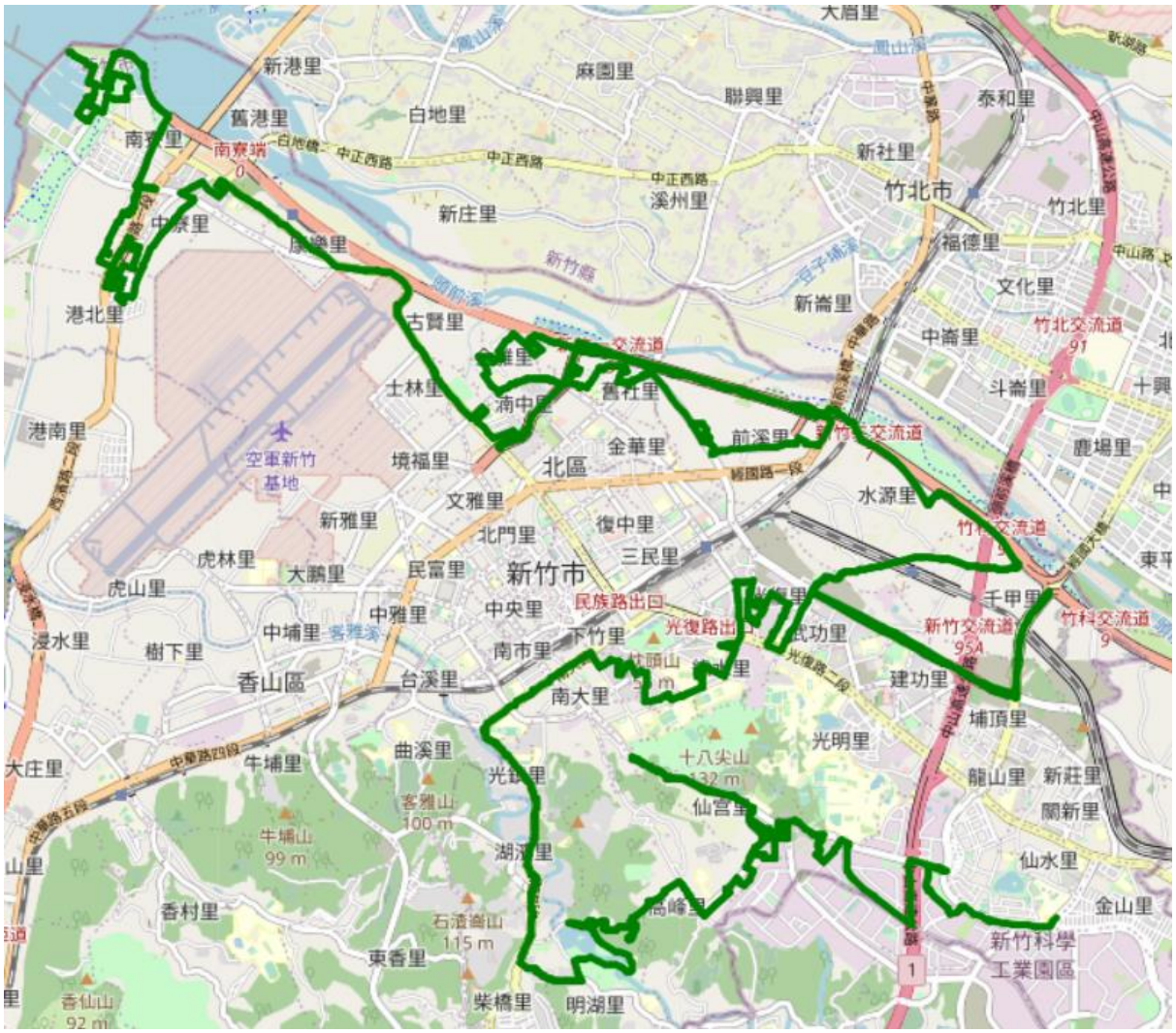
to Nanliao Fighing Port (ID: 8513026827)

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
```

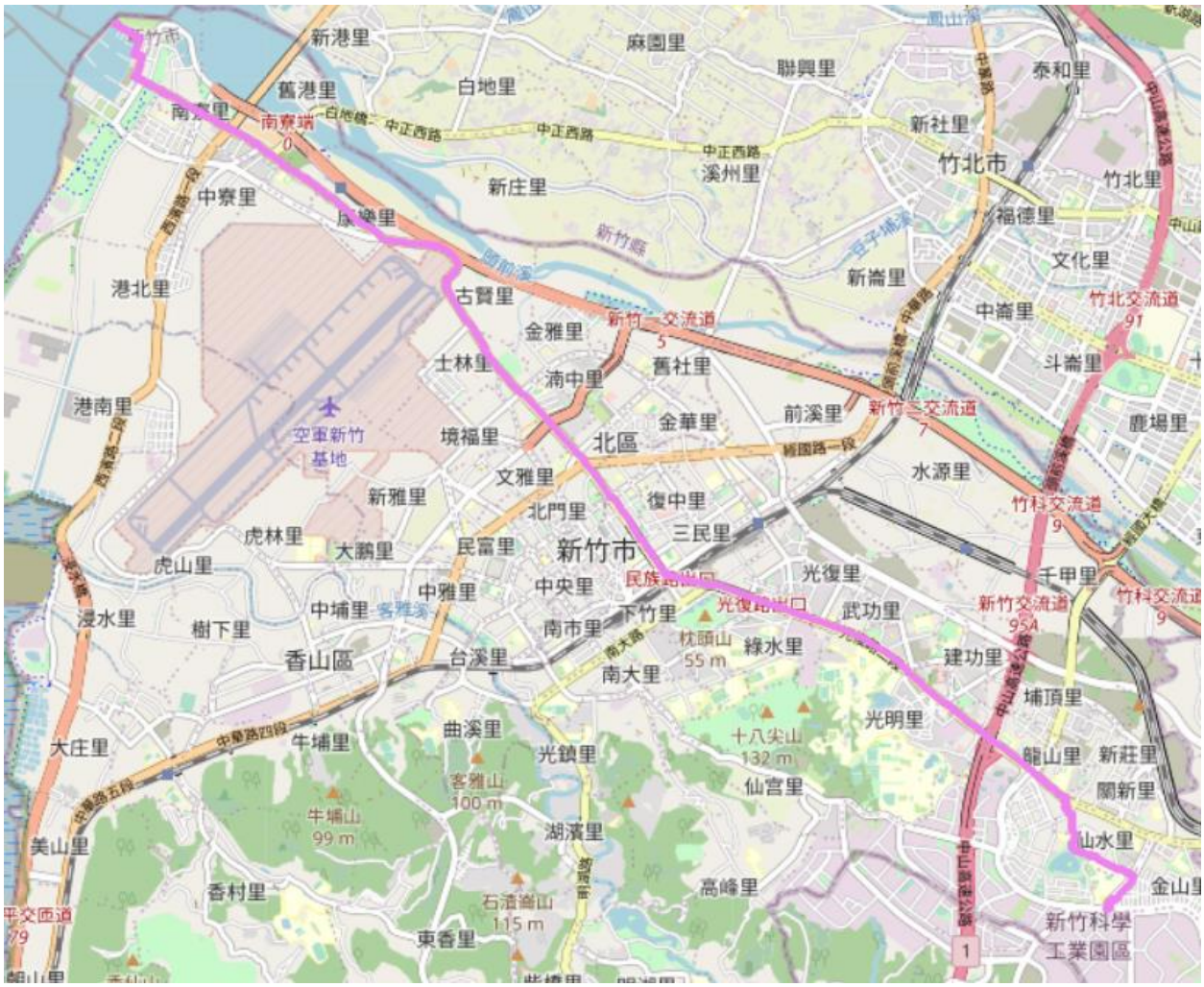
DFS (stack):

```
The number of nodes in the path found by DFS: 1521
Total distance of path found by DFS: 64821.60399999987 m
The number of visited nodes in DFS: 3291
```



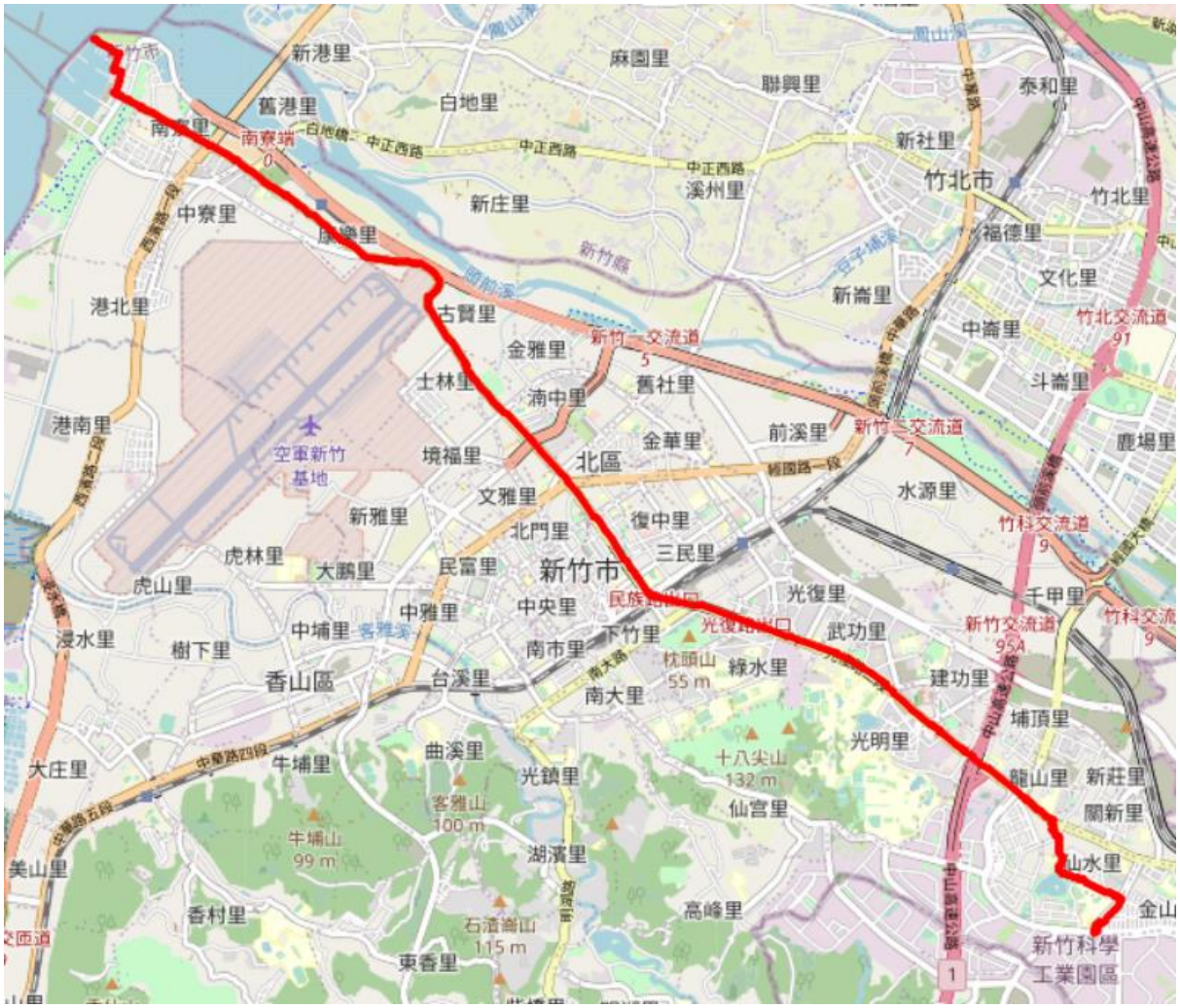
UCS:

```
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: 11925
```



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: 14428
```



Part III. Question Answering (12%):

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

   I can not successfully install the jupyter notebook at first and my drive have some stange problems at the same time, so I think for a long time how to solve it. After all, I chose still use vscode in my laptop and write a main.py to run it, and I use fmap.save to save the map into html file.

2. Besides speed limit and distance, could you please come up with another attribute that is es sential for route finding in the real world? Please explain the rationale.

The preference for every different road.

Some drivers prefer to drive on a road which is straight and wide instead of the twisty road or narrow road.

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: Collect data like the name of roads or stores. Construct a coordinate system and put the items on it. Convert the coordinate into the map we can see.

   localization: Collect the data from sensors like GPS on ourselves. Combine with the place on the map.

4. The estimated time of arrival (ETA) is one of the features of Uber Eats. To provide accurat e estimates for users, Uber Eats needs to dynamically update ETA based on their mechanis m. Please define a **dynamic heuristic equation** for ETA and explain the rationale of your de sign. Hint: You can consider meal prep time, delivery priority, multiple orders, etc.

   ETA = M + D / V + T + W + O

   M: meal preparing time
   D: remaining distance
   V: average velocity of the vehicle
   T: traffic or congestion factor
   W: weather effect
   O: others