Homework 2: Route Finding

Part I. Implementation (6%):

- The program use `rev` to store the node it comes from.
  It can help the program finds the path.
- `vis_count` calculate the number of node the program visits.
- The program starts from `end`, and then it use `rev` to find the correct path to `start`.
  Finally, the program reverse the path list to get the correct path.

BFS

- I read Store the graph

```
MAP = {}
dist = {}
num_visited = {}
vis_count = 0
rev = {}
with open(edgeFile,'r') as f:
    MAP_tmp = list(csv.reader(f,delimiter=','))
    for i in range(1,len(MAP_tmp)):
        edge = MAP_tmp[i]
        edge[0] = int(edge[0])
        edge[1] = int(edge[1])
        edge[2] = float(edge[2])
        if edge[0] in MAP:
            MAP[edge[0]].append((edge[1],edge[2]))
        else:
            MAP[edge[0]] = [(edge[1],edge[2])]
        dist[edge[0]] = math.inf
        dist[edge[1]] = math.inf
        num_visited[edge[0]] = 0
        num_visited[edge[1]] = 0
```

- The program use queue to store the node haven’t been explored, and use `num_visited` to check if it is visited or it is explored.

```
q = queue.Queue()
q.put(start)
dist[start] = 0
path = []
flag = False
while q.empty() == False:
    now = q.get()
    if num_visited[now] == 2:
        continue
    num_visited[now] = 2
    if now not in MAP:
        continue
    for (next1,DIS) in MAP[now]:
        if num_visited[next1] == 0:
            dist[next1] = dist[now] + DIS
            q.put(next1)
            rev[next1] = now
            num_visited[next1] = 1
            vis_count = vis_count + 1
        if next1 == end:
            flag = True
            break
    if(flag==True):
        break
rev_now = end
path.append(end)
while rev_now != start:
    rev_now = rev[rev_now]
    path.append(rev_now)
path.reverse()
return path,dist[end],vis_count
```

DFS(stack)

- Store the graph

```
MAP = {}
dist = {}
num_visited = {}
vis_count = 0
rev = {}

with open(edgeFile,'r') as f:
    MAP_tmp = list(csv.reader(f,delimiter=','))
    for i in range(1,len(MAP_tmp)):
        edge = MAP_tmp[i]
        edge[0] = int(edge[0])
        edge[1] = int(edge[1])
        edge[2] = float(edge[2])
        if edge[0] in MAP:
            MAP[edge[0]].append((edge[1],edge[2]))
        else:
            MAP[edge[0]] = [(edge[1],edge[2])]

        dist[edge[0]] = math.inf
        dist[edge[1]] = math.inf
        num_visited[edge[0]] = 0
        num_visited[edge[1]] = 0
```

- The program use list to implement stack, and do the dfs to find the answer

```
q = []
q.append(start)
dist[start] = 0
path = []
flag = False
while len(q) != 0:
    now = q.pop()
    if num_visited[now] == 2:
        continue
    num_visited[now] = 2
    if now not in MAP:
        continue
    for (next1,DIS) in MAP[now]:
        if num_visited[next1] == 0:
            dist[next1] = dist[now] + DIS
            q.append(next1)
            rev[next1] = now
            num_visited[next1] = 1
            vis_count = vis_count + 1
        if next1 == end:
            flag = True
            break
    if(flag==True):
        break
rev_now = end
path.append(end)
while rev_now != start:
    rev_now = rev[rev_now]
    path.append(rev_now)
path.reverse()
return path,dist[end],vis_count
```

UCS

- Store the graph

```
MAP = {}
dist = {}
num_visited = {}
rev = {}
vis_count = 1
with open(edgeFile,'r') as f:
    MAP_tmp = list(csv.reader(f,delimiter=','))
    for i in range(1,len(MAP_tmp)):
        edge = MAP_tmp[i]
        edge[0] = int(edge[0])
        edge[1] = int(edge[1])
        edge[2] = float(edge[2])
        if edge[0] in MAP:
            MAP[edge[0]].append((edge[1],edge[2]))
        else:
            MAP[edge[0]] = [(edge[1],edge[2])]
        dist[edge[0]] = math.inf
        dist[edge[1]] = math.inf
        num_visited[edge[0]] = 0
        num_visited[edge[1]] = 0
```

- Use `heapq` to implement the priority queue in UCS.
  The program only put the node which can has shorter path into `heapq`.

```
q = []
heapq.heappush(q,(0,start))
dist[start] = 0
path = []
while len(q) != 0:
    (DDIS,now) = heapq.heappop(q)
    if num_visited[now] == 2:
        continue
    num_visited[now] = 2
    if now == end:
        break
    if now not in MAP:
        continue
    for (next1,DIS) in MAP[now]:
       if (num_visited[next1]!=2):
           if(dist[next1] > dist[now] + DIS):
               dist[next1] = dist[now] + DIS
               heapq.heappush(q,(dist[next1],next1))
               rev[next1] = now
               vis_count = vis_count + 1

rev_now = end
path.append(end)
while rev_now != start:
    rev_now = rev[rev_now]
    path.append(rev_now)
path.reverse()
return path,dist[end],vis_count
```

A*

- Store the graph

```
MAP = {}
dist = {}
h_dist={}
num_visited = {}
rev = {}
vis_count = 1
with open(edgeFile,'r') as f:
    MAP_tmp = list(csv.reader(f,delimiter=','))
    for i in range(1,len(MAP_tmp)):
        edge = MAP_tmp[i]
        edge[0] = int(edge[0])
        edge[1] = int(edge[1])
        edge[2] = float(edge[2])
        if edge[0] in MAP:
            MAP[edge[0]].append((edge[1],edge[2]))
        else:
            MAP[edge[0]] = [(edge[1],edge[2])]
        dist[edge[0]] = math.inf
        dist[edge[1]] = math.inf
        num_visited[edge[0]] = 0
        num_visited[edge[1]] = 0
```

- Compared with UCS, the program use the data in `heuristic.csv` to estimate the distance.
  `h_dist` is the data in `heuristic.csv`.

```
index = 0
with open('heuristic.csv','r') as f:
    MAP_tmp = list(csv.reader(f,delimiter=','))
    for i in range(1,len(MAP_tmp[0])):
        if end == int(MAP_tmp[0][i]):
            index = i
    for i in range(1,len(MAP_tmp)):
        h_dist[int(MAP_tmp[i][0])] = float(MAP_tmp[i][index])
q = []
heapq.heappush(q,(h_dist[start],start))
dist[start] = 0
path = []
while len(q) != 0:
    (DDIS,now) = heapq.heappop(q)
    if num_visited[now] == 2:
        continue
    num_visited[now] = 2
    if now == end:
        break
    if now not in MAP:
        continue
    for (next1,DIS) in MAP[now]:
       if (num_visited[next1]!=2):
            if(dist[next1] > dist[now] + DIS):
                dist[next1] = dist[now] + DIS
                heapq.heappush(q,(dist[next1]+h_dist[next1],next1))
                rev[next1] = now
                vis_count = vis_count + 1

rev_now = end
path.append(end)
while rev_now != start:
    rev_now = rev[rev_now]
    path.append(rev_now)
path.reverse()
return path,dist[end],vis_count
```

Part 6 A* time

- Store the graph and transform the unit of speed

```
MAP = {}
dist = {}
h_dist={}
num_visited = {}
rev = {}
vis_count = 1
with open(edgeFile,'r') as f:
    MAP_tmp = list(csv.reader(f,delimiter=','))
    for i in range(1,len(MAP_tmp)):
        edge = MAP_tmp[i]
        edge[0] = int(edge[0])
        edge[1] = int(edge[1])
        edge[2] = float(edge[2])
        edge[3] = float(edge[3])*1000/3600
        if edge[0] in MAP:
            MAP[edge[0]].append((edge[1],edge[2],edge[3]))
        else:
            MAP[edge[0]] = [(edge[1],edge[2],edge[3])]
        dist[edge[0]] = math.inf
        dist[edge[1]] = math.inf
        num_visited[edge[0]] = 0
        num_visited[edge[1]] = 0
```

- Compared with A*, the program use the data in `heuristic.csv` to estimate the distance. The program let $h(x) = h_{dist}(x)/speedOfTheRoad$ to estiimate the time it may take.

```
index = 0
with open('heuristic.csv','r') as f:
    MAP_tmp = list(csv.reader(f,delimiter=','))
    for i in range(1,len(MAP_tmp[0])):
        if end == int(MAP_tmp[0][i]):
            index = i
    for i in range(1,len(MAP_tmp)):
        h_dist[int(MAP_tmp[i][0])] = float(MAP_tmp[i][index])
q = []
heapq.heappush(q,(h_dist[start],start))
dist[start] = 0
path = []
while len(q) != 0:
    (DDIS,now) = heapq.heappop(q)
    if num_visited[now] == 2:
        continue
    num_visited[now] = 2
    if now == end:
        break
    if now not in MAP:
        continue
    for (next1,DIS,SP) in MAP[now]:
       if (num_visited[next1]!=2):
            if(dist[next1] > dist[now] + DIS/SP):
                dist[next1] = dist[now] + DIS/SP
                heapq.heappush(q,(dist[next1]+(h_dist[next1])/SP,next1))
                rev[next1] = now
                vis_count = vis_count + 1

rev_now = end
path.append(end)
while rev_now != start:
    rev_now = rev[rev_now]
    path.append(rev_now)
path.reverse()
return path,dist[end],vis_count
```

Part II. Results & Analysis (12%):

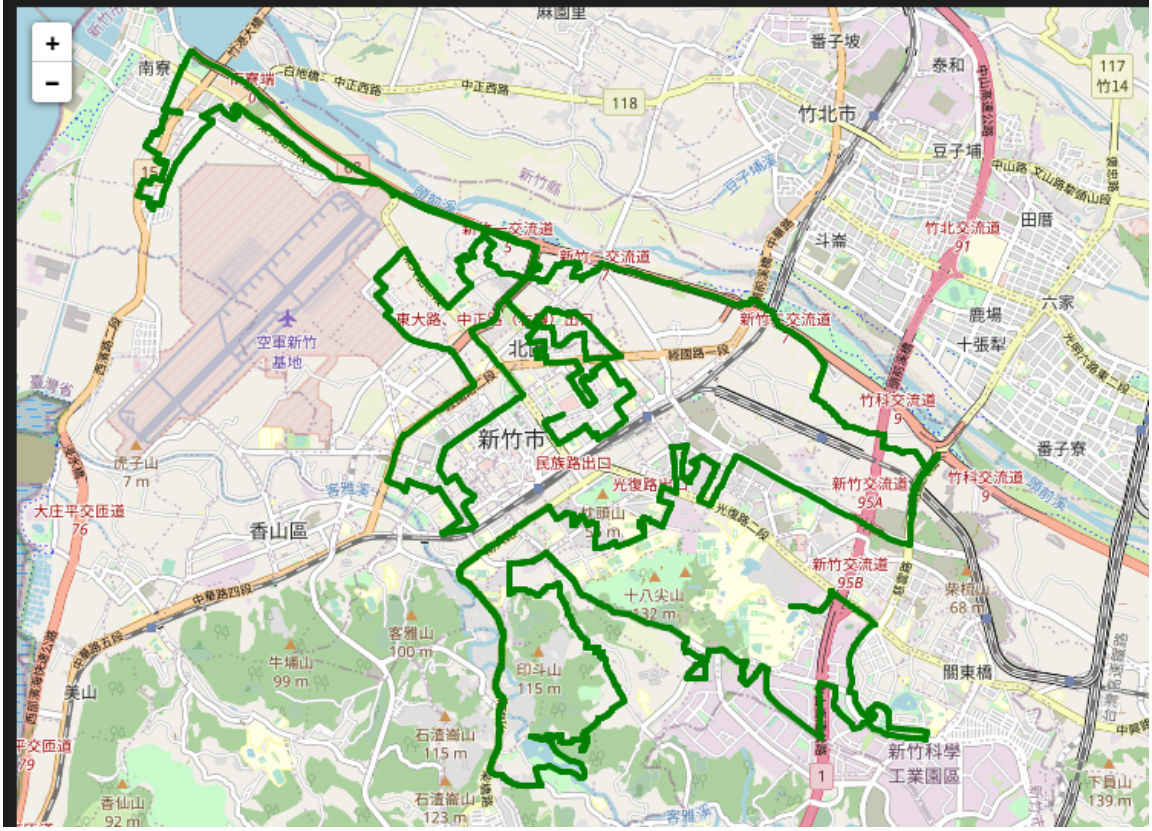
Test1

▼ 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
```

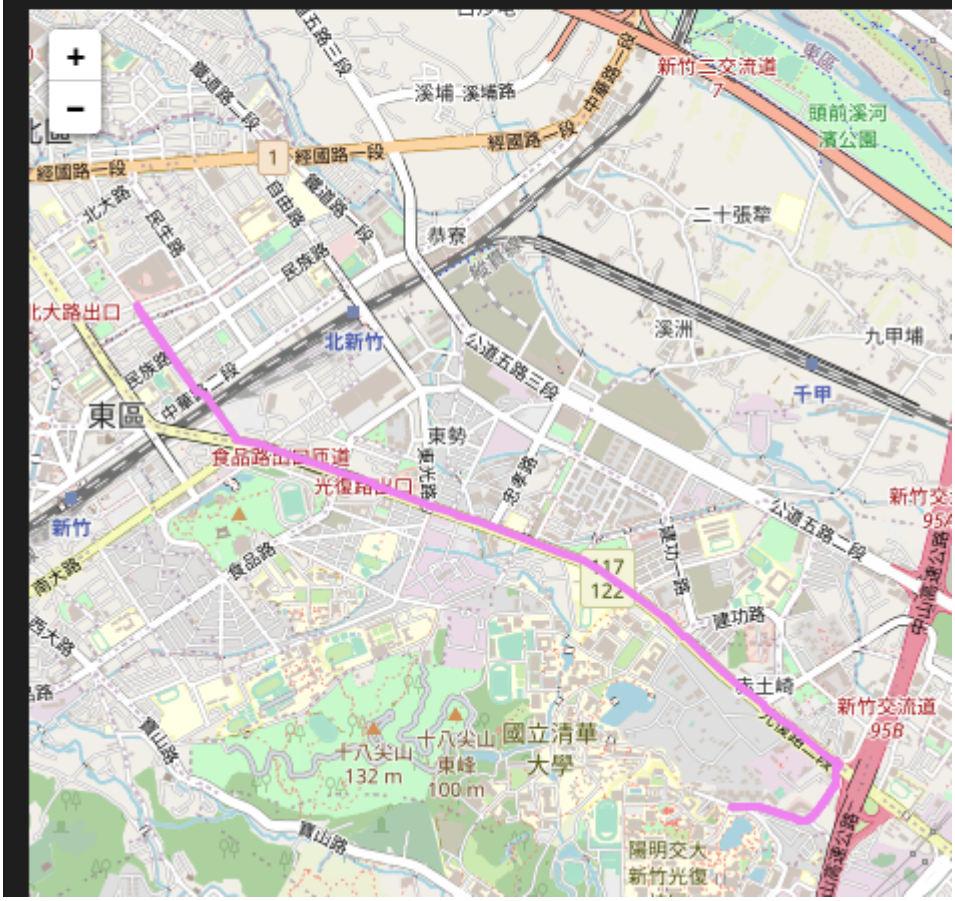
▼ DFS

```
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: 5235
```



▼ 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: 5307
```



▼ 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: 318
```

▼ A* time

```
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: 269
```

Test2

▼ 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

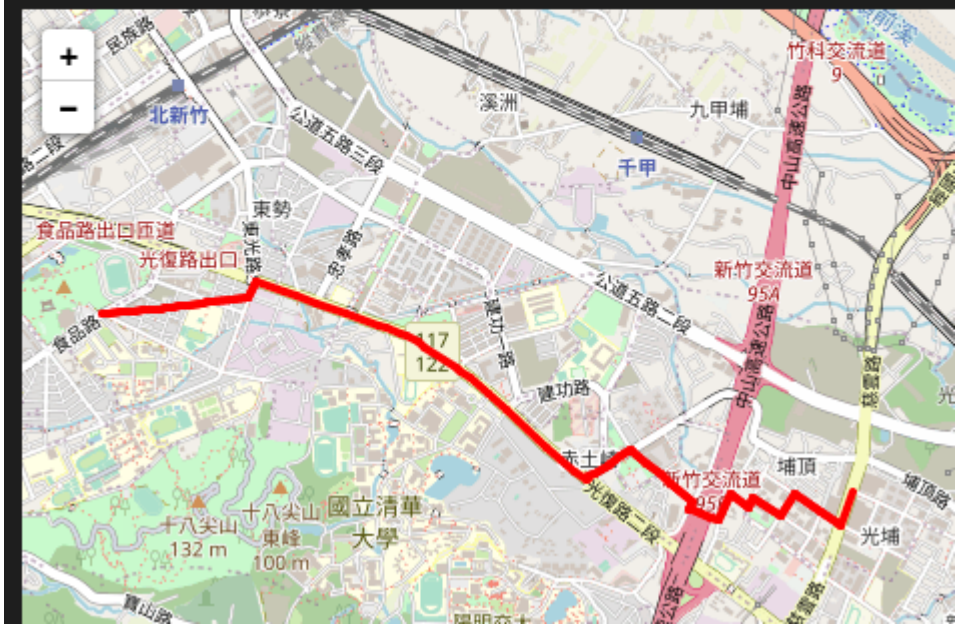
```
The number of nodes in the path found by DFS: 930
Total distance of path found by DFS: 38752.30799999996 m
The number of visited nodes in DFS: 9615
```

▼ 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: 7566
```

▼ 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: 1309
```



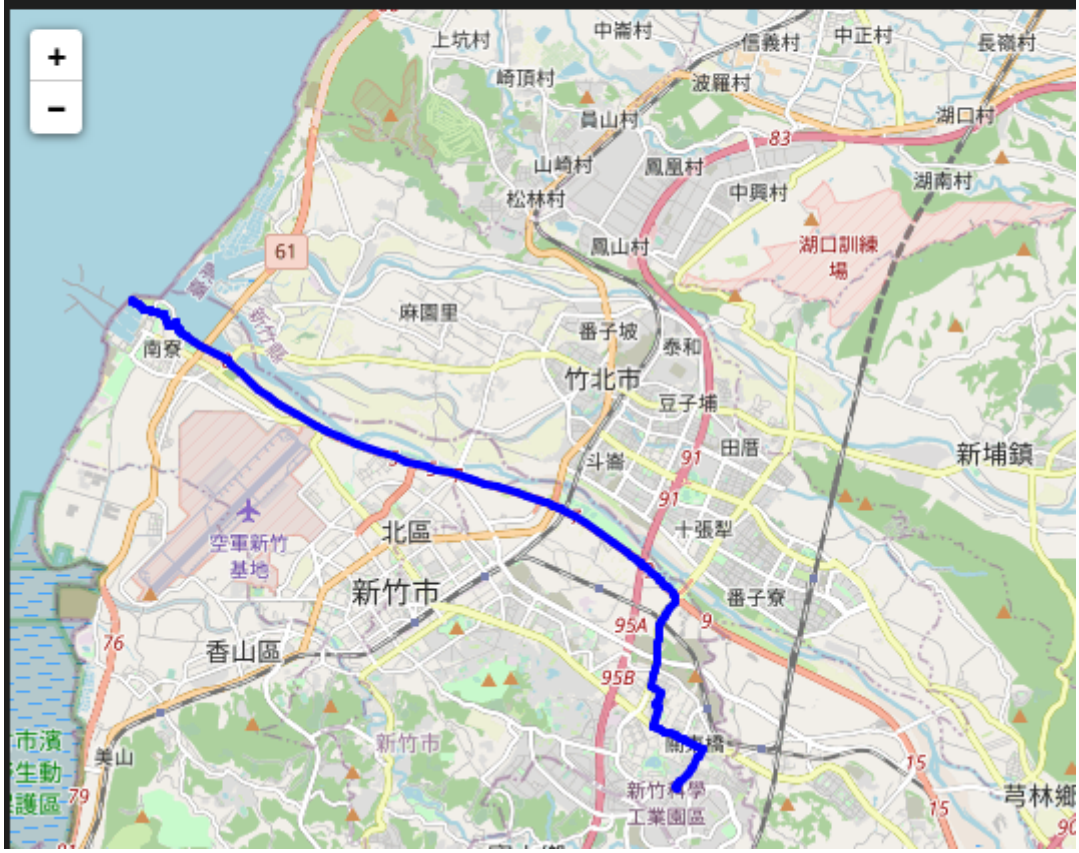
▼ A* time

```
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: 1208
```

Test3

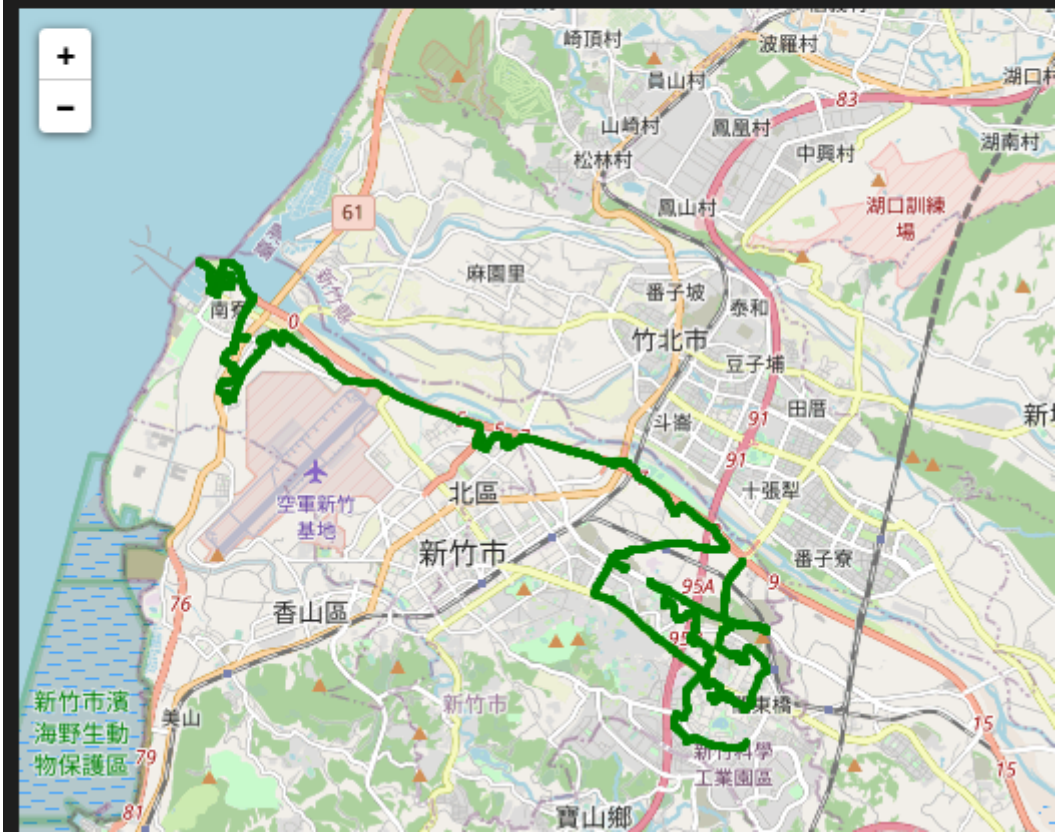
▼ BFS

```
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

```
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: 2493
```

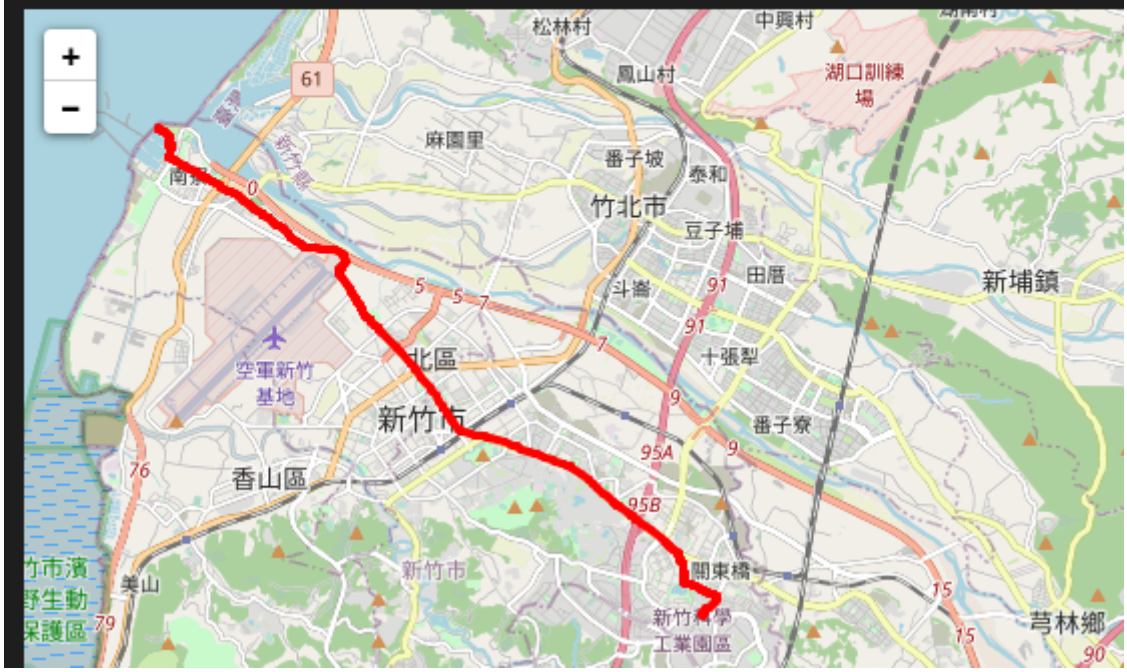


▼ UCS

```
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: 12320
```

▼ 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: 7772
```



▼ A* time

```
The number of nodes in the path found by A* search: 234
Total second of path found by A* search: 914.3259868458373 s
The number of visited nodes in A* search: 6304
```

Discussion

	BFS	DFS	UCS	A*	A* time
TEST1	node: 88 distance: 4978.882	node: 1718 distance: 75504	node: 89 distance: 4367.881	node: 89 distance: 4367.881	node: 89 distance: 320.88
TEST2	node: 60 distance: 4215.521	node: 930 distance: 38752	node: 63 distance: 4101.84	node: 63 distance: 4101.84	node: 63 distance: 304.44
TEST3	node:183 distance:15442.395	node: 900 distance: 39219	node: 288 distance: 14212.41	node: 288 distance: 14212.41	node:234 distance: 914.33

	UCS	A*
TEST1	visit node: 5307	visit node: 318
TEST2	visit node: 7566	visit node: 1309
TEST3	visit node: 12320	visit node: 7772

We can discover that A* is the best method in this three test dataset. A* can have the best distance and take less step to find the answer, since $h(x)$ give a great estimation.

UCS is the second, since its performance is as good as A*, but t**ake more step** to find the answer.

The BFS doesn't work bad, but **it only consider the depth instead of distance**. It is not the good method but not the worst.

The DFS is the worst method. It takes lots of time to calculate and get the worst performance, since it just give the first path it find.

Part III. Question Answering (12%):

1. Please describe a problem you encountered and how you solved it.
    - I take a lots of time to find the correct solution of BFS, since the BFS I learn before can work better than this one.
    - I check the slide to find the method I need to implement.
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.

- The number of cars driving on the road will affect the speed people actually drive. The more car there is, the slower driver may drive.
- The number of traffic light will affect too. Since when the traffic light turn red, it will force the driver to stop and wait for it.

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: Every intersection of road can be viewed as the node of the graph and the road is the edge. We can use this the build the map with node and edge.
- Localization: We choose the nearest intersection be the position. Though it may have some bias, it is still a useful method.

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.

    - The heuristic function can be the time the past delivery man take. Since the value may be weird if it is in the crowded time, we need to take time into consideration. Moreover, every delivery man may have different patterns, we can get better estimation if we did better personalization.