HW2 Report

109550206 陳品劭 Self link

Part I. Implementation

Part 1

```
#
import queue
#
# Begin your code (Part 1)
#raise NotImplementedError("To be implemented")
"""
    1. Load all data in edges.csv with
       { start node id : { end node id : (distance, speed limit) } }
    2. init node_visted as -10
"""
graph = dict()
node_visted = dict()
with open(edgeFile) as f:
    for line in f:
        data = line.split(',')
        if data[0] == 'start':
            continue
        data[3] = data[3].split('\n')[0]
        node_visted[data[0]] = int(-10)
        node_visted[data[1]] = int(-10)
        if data[0] not in graph:
            graph[data[0]] = dict()
        if data[1] not in graph:
            graph[data[1]] = dict()
        graph [data[0]] [data[1]] = (float(data[2]), float(data[3]))
"""
    3. init return values
"""
path = list()
dist = float(0.0)
num_visted = int(0)
"""
    4. BFS init
        - found, a bool value means we found end node or not
        - create a FIFO datastructure, q
        - put the start node int to q
        - mark start node in node_visted table as START
"""
found = bool(0)
q = queue.Queue()
q.put( str(start) )
node_visted[str(start)] = 'START'
"""
    5. BFS
        - termination condition: found end point or no other point can touch
        - put every node adjacent with the top node in queue
        - mark the node adjacent with the top node
          in node_visted table as the top node
"""
while q.empty() == 0 and found == 0:
```

```
    now = q.get()
    num_visted += 1
    for new in graph[now]:
        if node_visted[new] == -10:
            node_visted[new] = now
            if new == str(end):
                found = 1
                break
            q.put(new)
"""
    6. get the path from end node
        - the mark of every node in node_visted is
          the prev node of the path from start node to end node
        - sum the distance of these paths
        - append these nodes to list
        - when we see the mark of start is finish
"""
now = str(end)
if found == 1:
    while node_visted[now] != 'START':
        path.append(int(now))
        dist += graph[node_visted[now]][now][0]
        now = node_visted[now]
"""
    7. append the start node
    8. reverse the list
    9. return
"""
path.append(start)
path.reverse()
return path, dist, num_visted
# End your code (Part 1)
```

Part 2

```
#
import queue
#
# Begin your code (Part 2)
#raise NotImplementedError("To be implemented")
"""
    1. Load all data in edges.csv with
       { start node id : { end node id : (distance, speed limit) } }
    2. init node_visted as -10
"""
graph = dict()
node_visted = dict()
with open(edgeFile) as f:
    for line in f:
        data = line.split(',')
        if data[0] == 'start':
            continue
        data[3] = data[3].split('\n')[0]
        node_visted[data[0]] = int(-10)
        node_visted[data[1]] = int(-10)
        if data[0] not in graph:
            graph[data[0]] = dict()
        if data[1] not in graph:
            graph[data[1]] = dict()
        graph [data[0]] [data[1]] = (float(data[2]), float(data[3]))
"""
```

```
    3. init return values
"""
path = list()
dist = float(0.0)
num_visted = int(0)
"""
    4. DFS init
        - found, a bool value means we found end node or not
        - create a LIFO datastructure, q
        - put the start node int to q
        - mark start node in node_visted table as START
"""
found = bool(0)
q = queue.LifoQueue()
q.put( str(start) )
node_visted[str(start)] = 'START'
"""
    5. DFS
        - termination condition: found end point or no other point can touch
        - put every node adjacent with the top node in stack (q)
        - mark the node adjacent with the top node
          in node_visted table as the top node
"""
while q.empty() == 0 and found == 0:
    now = q.get()
    num_visted += 1
    for new in graph[now]:
        if node_visted[new] == -10:
            node_visted[new] = now
            if new == str(end):
                found = 1
                break
            q.put(new)
"""
    6. get the path from end node
        - the mark of every node in node_visted is
          the prev node of the path from start node to end node
        - sum the distance of these paths
        - append these nodes to list
        - when we see the mark of start is finish
"""
now = str(end)
if found == 1:
    while node_visted[now] != 'START':
        path.append(int(now))
        dist += graph[node_visted[now]][now][0]
        now = node_visted[now]
"""
    7. append the start node
    8. reverse the list
    9. return
"""
path.append(start)
path.reverse()
return path, dist, num_visted
# End your code (Part 2)
```

Part3

```
#
import queue
#
# Begin your code (Part 3)
#raise NotImplementedError("To be implemented")
"""
    1. Load all data in edges.csv with
       { start node id : { end node id : (distance, speed limit) } }
    2. init node_visted as -10
"""
graph = dict()
node_visted = dict()
with open(edgeFile) as f:
    for line in f:
        data = line.split(',')
        if data[0] == 'start':
            continue
        data[3] = data[3].split('\n')[0]
        node_visted[data[0]] = int(-10)
        node_visted[data[1]] = int(-10)
        if data[0] not in graph:
            graph[data[0]] = dict()
        if data[1] not in graph:
            graph[data[1]] = dict()
        graph[data[0]][data[1]] = (float(data[2]), float(data[3]))
"""
    3. init return values
"""
path = list()
dist = float(0.0)
num_visted = int(0)
"""
    4. UCS init
        - found, a bool value means we found end node or not
        - create a prioity queue (q) to let smallest distance be the top
        - put the start node int to q
        - mark start node in node_visted table as (START, 0)
"""
found = bool(0)
q = queue.PriorityQueue()
q.put( (0, str(start)) )
node_visted[str(start)] = ('START', 0.0)
"""
    5. UCS
        - termination condition: found end point or no other point can touch
        - put every node adjacent with the top node in
          prioity queue (q) as (the distance have walked, node id)
        - mark the node adjacent with the top node
          in node_visted table as (the top node, the distance have walked)
        - if the node is marked, we need to check
          the distance have walked is smaller or not
"""
while q.empty() == 0 and found == 0:
    now = q.get()
    num_visted += 1
    if now[1] == str(end):
        found = 1
        break
    for new in graph[now[1]]:
```

```
        weight =  now[0] + graph[now[1]][new][0]
        if node_visted[new] == -10:
            node_visted[new] = (now[1], weight)
            q.put( (weight, new) )
        elif weight < node_visted[new][1]:
            node_visted[new] = (now[1], weight)
            q.put( (weight, new) )
"""
    6. get the path from end node
        - the mark of every node in node_visted is
          the prev node of the path from start node to end node
        - the distance of node_visted[end] will equal to
          the sum of the distance of these paths
        - append these nodes to list
        - when we see the mark of start is finish
"""
now = str(end)
dist = node_visted[now][1]
if found == 1:
    while node_visted[now][0] != 'START':
        path.append(int(now))
        now = node_visted[now][0]
"""
    7. append the start node
    8. reverse the list
    9. return
"""
path.append(start)
path.reverse()
return path, dist, num_visted
# End your code (Part 3)
```

Part 4

```
#
import queue
#
# Begin your code (Part 4)
#raise NotImplementedError("To be implemented")
"""
    1. Load all data in edges.csv with
       { start node id : { end node id : (distance, speed limit) } }
    2. init node_visted as -10
"""
graph = dict()
node_visted = dict()
with open(edgeFile) as f:
    for line in f:
        data = line.split(',')
        if data[0] == 'start':
            continue
        data[3] = data[3].split('\n')[0]
        node_visted[data[0]] = int(-10)
        node_visted[data[1]] = int(-10)
        if data[0] not in graph:
            graph[data[0]] = dict()
        if data[1] not in graph:
            graph[data[1]] = dict()
        graph [data[0]] [data[1]] = (float(data[2]), float(data[3]))
"""
    3. check the end node to choose the data we need
```

```
    4. Load all data in heuristic.csv
       { node id : straight line distance with end node}
"""
stra_dist = dict()
case = 0
with open(heuristicFile) as f:
    for line in f:
        data = line.split(',')
        data[3] = data[3].split('\n')[0]
        if data[0] == 'node':
            for i in range(1, 4):
                if data[i] == str(end):
                    case = i
                    break
            continue
        stra_dist[ data[0] ] = float(data[case])
"""
    5. init return values
"""
path = list()
dist = float(0.0)
num_visted = int(0)
"""
    6. A* init
        - found, a bool value means we found end node or not
        - create a prioity queue (q) to let smallest g() + h() be the top
              - g(): the distance which have walked
              - h(): straight line distance between that node and end node
        - put the start node int to q
        - mark start node in node_visted table as (START, 0)
"""
found = bool(0)
q = queue.PriorityQueue()
q.put( (0 + stra_dist[str(start)], str(start)) )
node_visted[str(start)] = ('START', 0.0)
"""
    7. A*
        - termination condition: found end point or no other point can touch
        - put every node adjacent with the top node
          in prioity queue (q) as (g() + h(), node id)
        - mark the node adjacent with the top node
          in node_visted table as (the top node, g() + h())
        - if the node is marked, we need to check the g() + h() is smaller or not
"""
while q.empty() == 0 and found == 0:
    now = q.get()
    num_visted += 1
    if now[1] == str(end):
        found = 1
        break
    for new in graph[now[1]]:
        weight = graph[now[1]][new][0] + now[0]
                 + stra_dist[new] - stra_dist[now[1]]
        if node_visted[new] == -10:
            node_visted[new] = (now[1], weight)
            q.put( (weight, new) )
        elif weight < node_visted[new][1]:
            node_visted[new] = (now[1], weight)
            q.put( (weight, new) )
"""
    8. get the path from end node
        - the mark of every node in node_visted is
```

```
          the prev node of the path from start node to end node
        - g() + h() of node_visted[end] will equal to
          the sum of the distance of these paths
        - append these nodes to list
        - when we see the mark of start is finish
"""
now = str(end)
dist = node_visted[now][1]
if found == 1:
    while node_visted[now][0] != 'START':
        path.append(int(now))
        now = node_visted[now][0]
"""
    9. append the start node
    10. reverse the list
    11. return
"""
path.append(start)
path.reverse()
return path, dist, num_visted
# End your code (Part 4)
```

Part 6

```
#
import queue
#
# Begin your code (Part 6)
#raise NotImplementedError("To be implemented")
"""
    different thing:
    1. To find the largest speed limit
    2. change the speed limit from km/hr to m/s
    others same with A*
"""
graph = dict()
node_visted = dict()
Max = 0
with open(edgeFile) as f:
    for line in f:
        data = line.split(',')
        if data[0] == 'start':
            continue
        data[3] = data[3].split('\n')[0]
        node_visted[data[0]] = int(-10)
        node_visted[data[1]] = int(-10)
        if data[0] not in graph:
            graph[data[0]] = dict()
        if data[1] not in graph:
            graph[data[1]] = dict()
        graph [data[0]] [data[1]] = (float(data[2]),
                                     float(data[3]) / 60 / 60 * 1000)
        Max = max(Max, float(data[3]) / 60 / 60 * 1000 )
"""
    same with A*
"""
stra_dist = dict()
case = 0
with open(heuristicFile) as f:
    for line in f:
        data = line.split(',')
```

```
        data[3] = data[3].split('\n')[0]
        if data[0] == 'node':
            for i in range(1, 4):
                if data[i] == str(end):
                    case = i
                    break
            continue
        stra_dist[ data[0] ] = float(data[case])
"""
    same with A*
"""
path = list()
time = float(0.0)
num_visted = int(0)
"""
    3. weight
        - g(): the fatest time we need to go to this node (distance / speed limit)
        - h(): straight line distance between
          that node and end node / the max speed limit of this graph
    same with A*
"""
found = bool(0)
q = queue.PriorityQueue()
q.put( (0 + stra_dist[str(start)] / Max, str(start)) )
node_visted[str(start)] = ('START', 0.0)
"""
    same with A*
"""
while q.empty() == 0 and found == 0:
    now = q.get()
    num_visted += 1
    if now[1] == str(end):
        found = 1
        break
    for new in graph[now[1]]:
        weight = graph[now[1]][new][0] / graph[now[1]][new][1] + now[0]
                 + stra_dist[new] / Max - stra_dist[now[1]] / Max
        if node_visted[new] == -10:
            node_visted[new] = (now[1], weight)
            q.put( (weight, new) )
        elif weight < node_visted[new][1]:
            node_visted[new] = (now[1], weight)
            q.put( (weight, new) )
"""
    same with A*
"""
now = str(end)
time = node_visted[now][1]
if found == 1:
    while node_visted[now][0] != 'START':
        path.append(int(now))
        now = node_visted[now][0]
"""
    same with A*
"""
path.append(start)
path.reverse()
return path, time, num_visted
# End your code (Part 6)
```

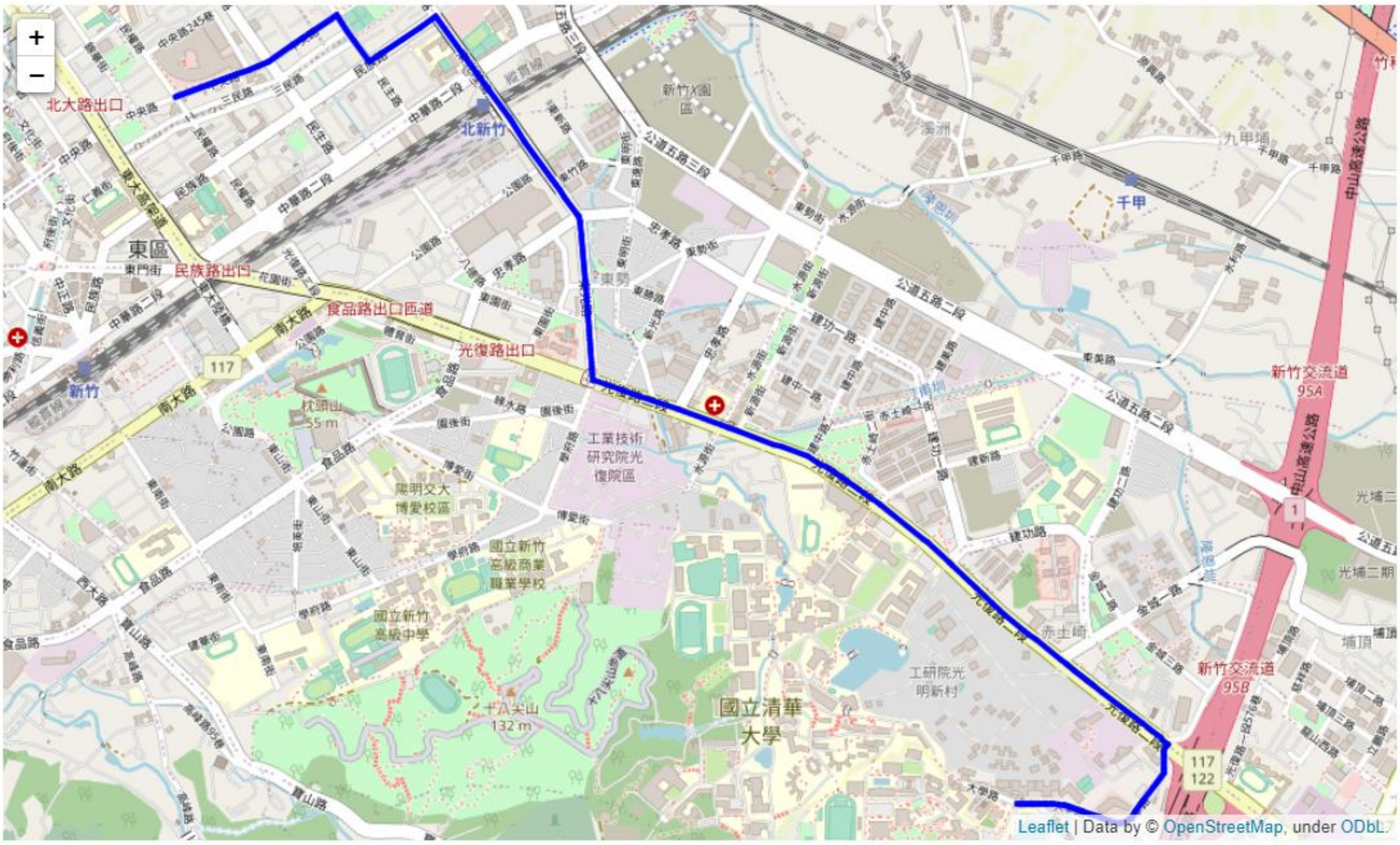
Part II. Results & Analysis

Test 1：

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

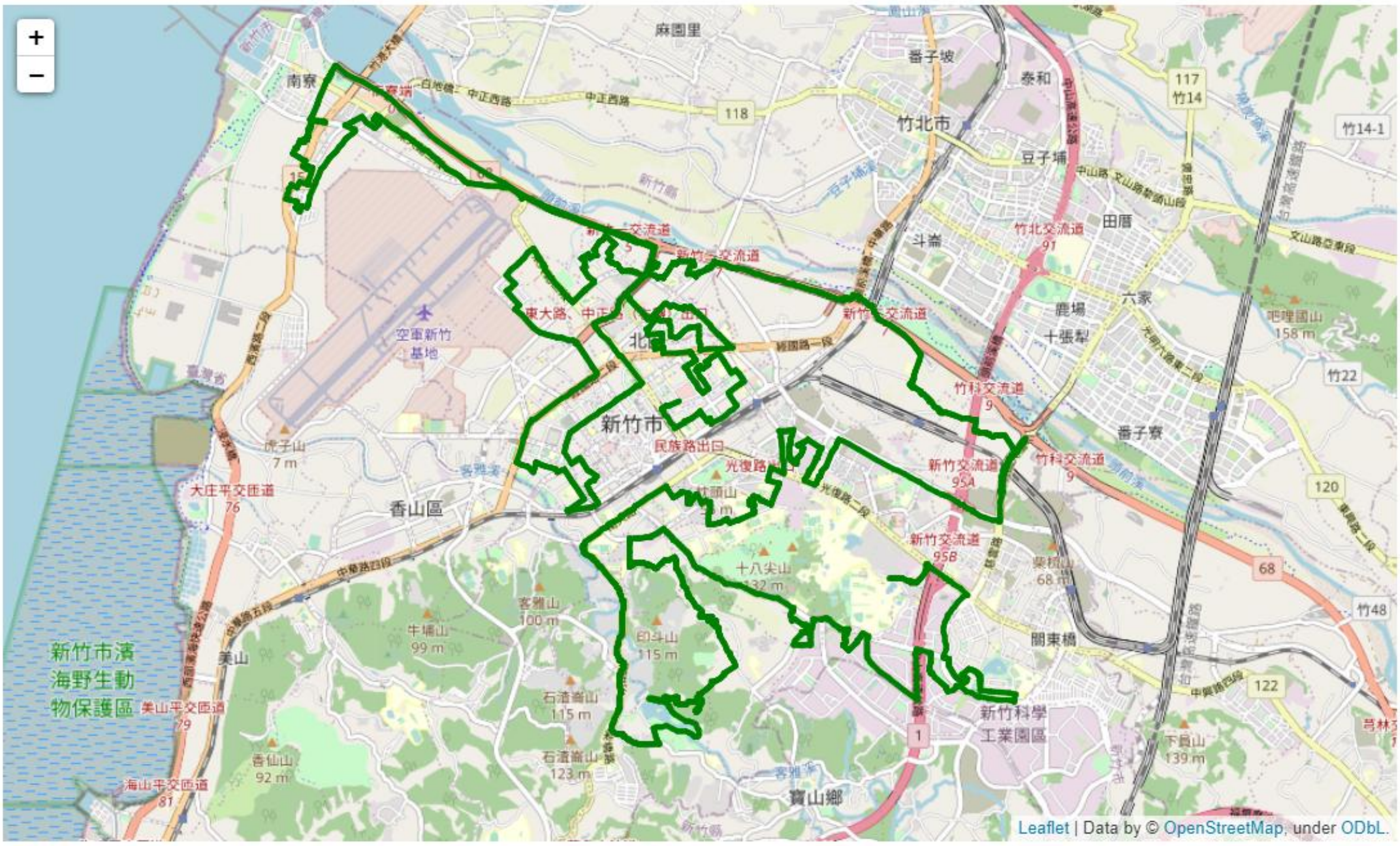
BFS

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: 4130



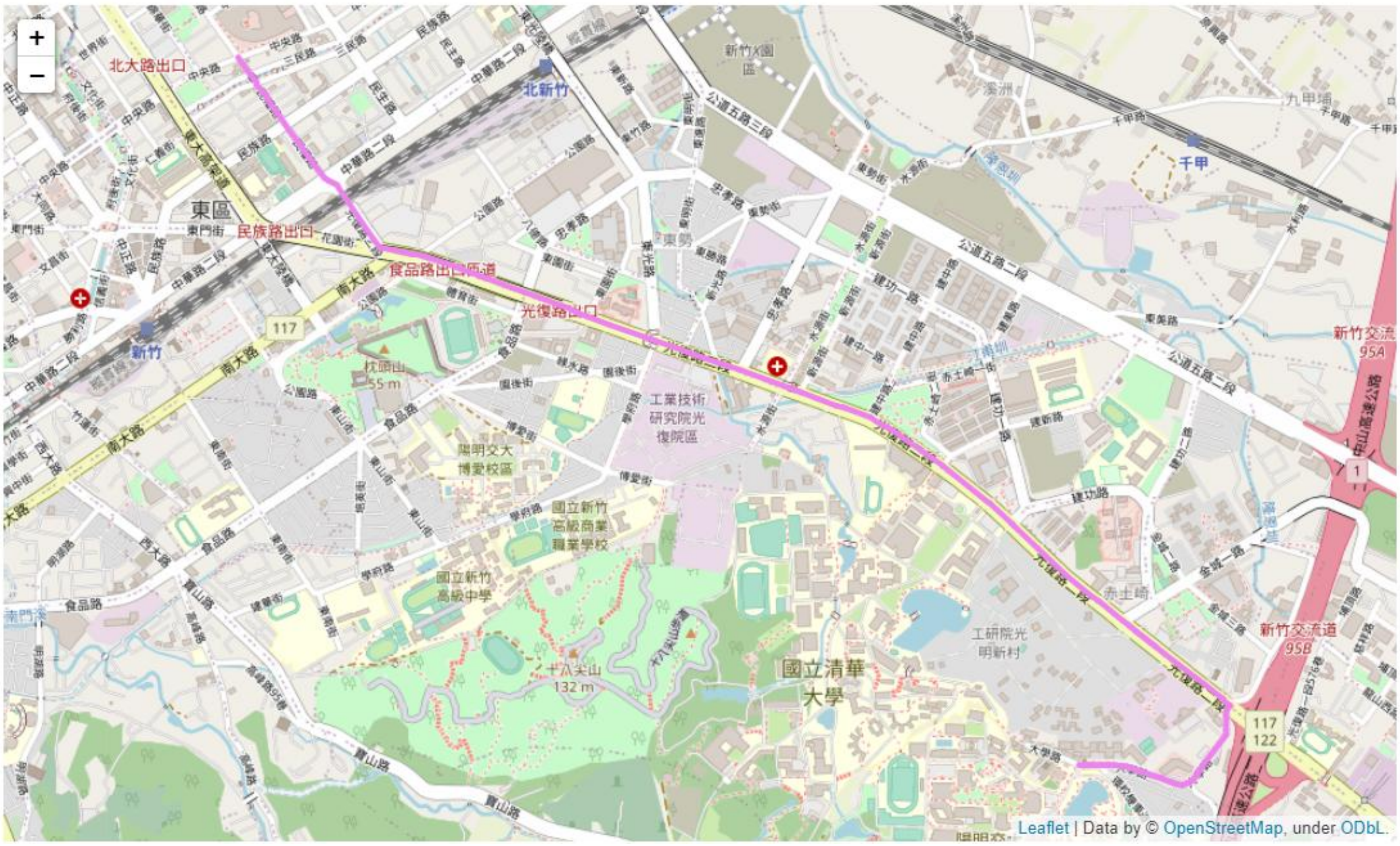
DFS (stack)

The number of nodes in the path found by DFS: 1718
Total distance of path found by DFS: 75504.3150000001 m
The number of visited nodes in DFS: 4711



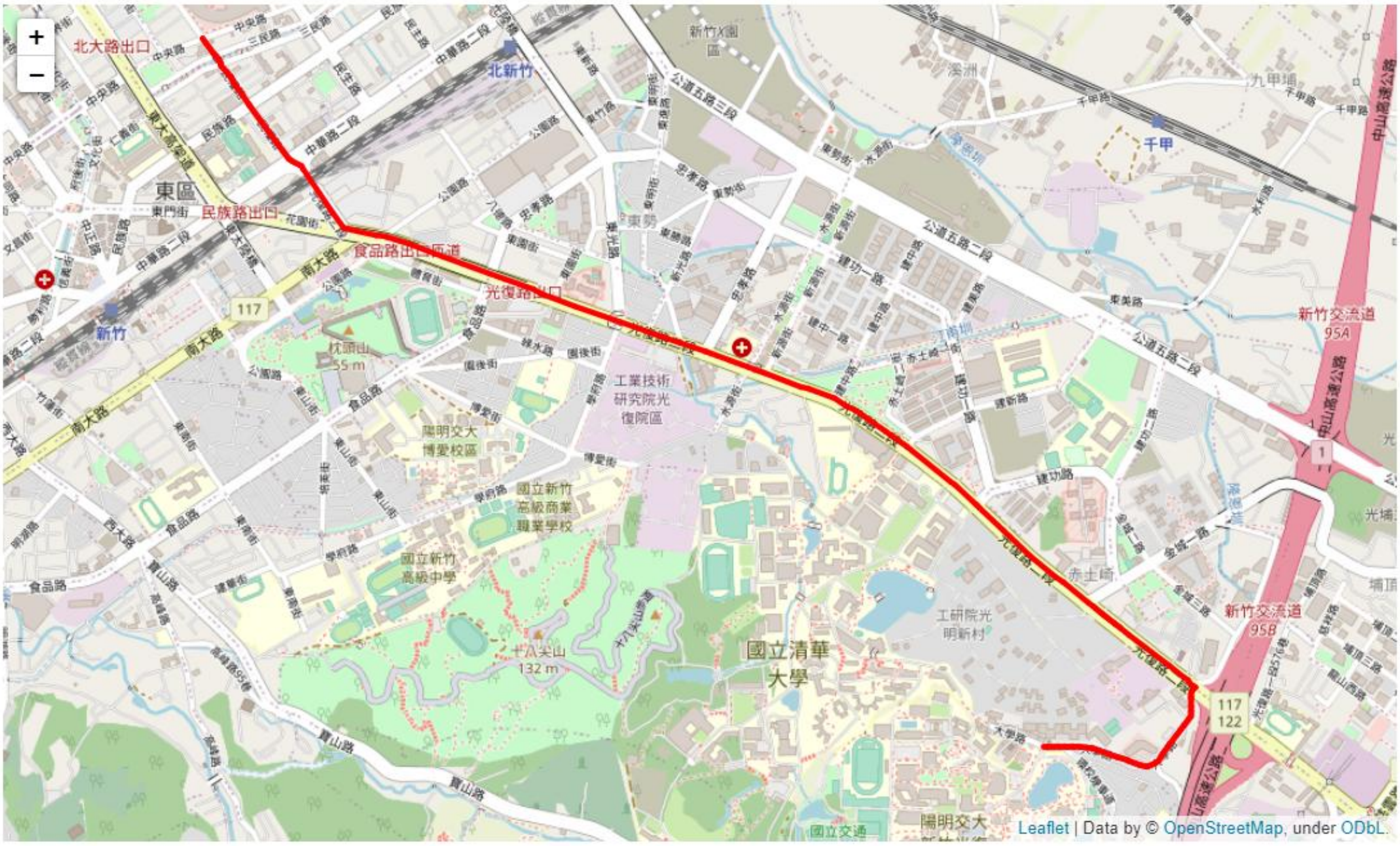
UCS

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: 5232



A*

The number of nodes in the path found by A* search: 89
Total distance of path found by A* search: 4367.880999999995 m
The number of visited nodes in A* search: 262



A* (Bonus)

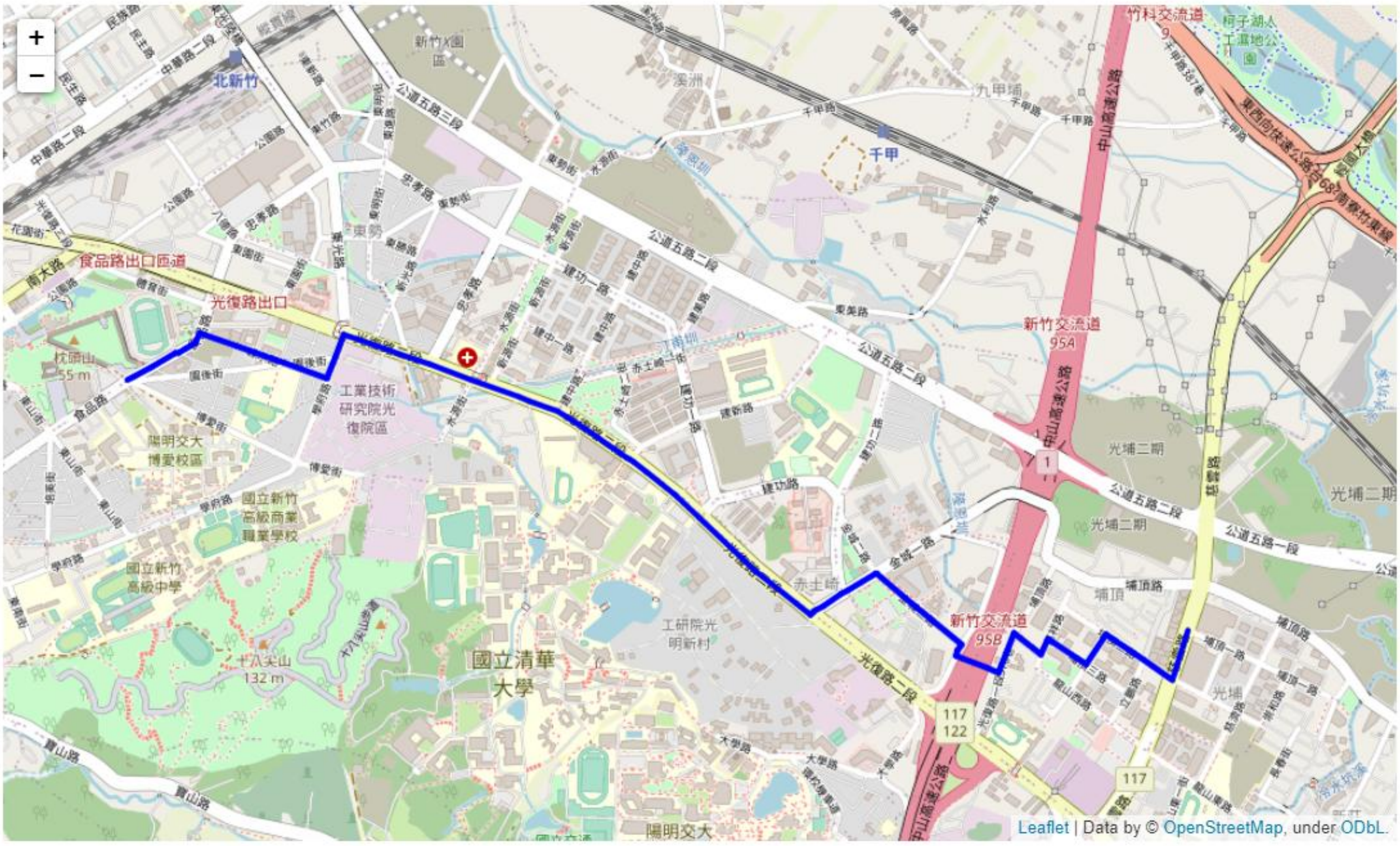
The number of nodes in the path found by A* search: 89
Total second of path found by A* search: 320.87823163083146 s
The number of visited nodes in A* search: 2016

Test 2：

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

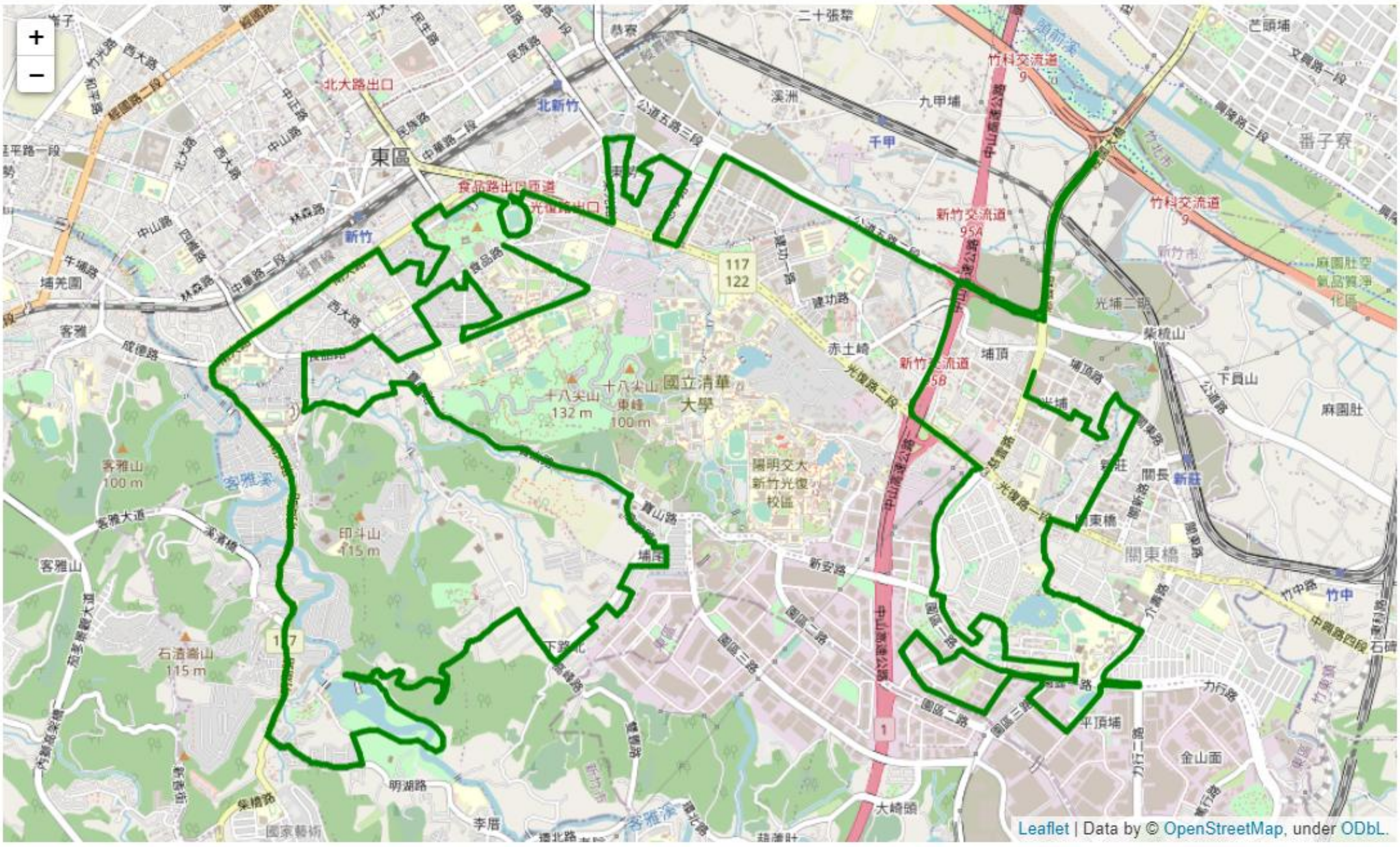
BFS

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: 4468



DFS (stack)

The number of nodes in the path found by DFS: 930
Total distance of path found by DFS: 38752.307999999895 m
The number of visited nodes in DFS: 9365

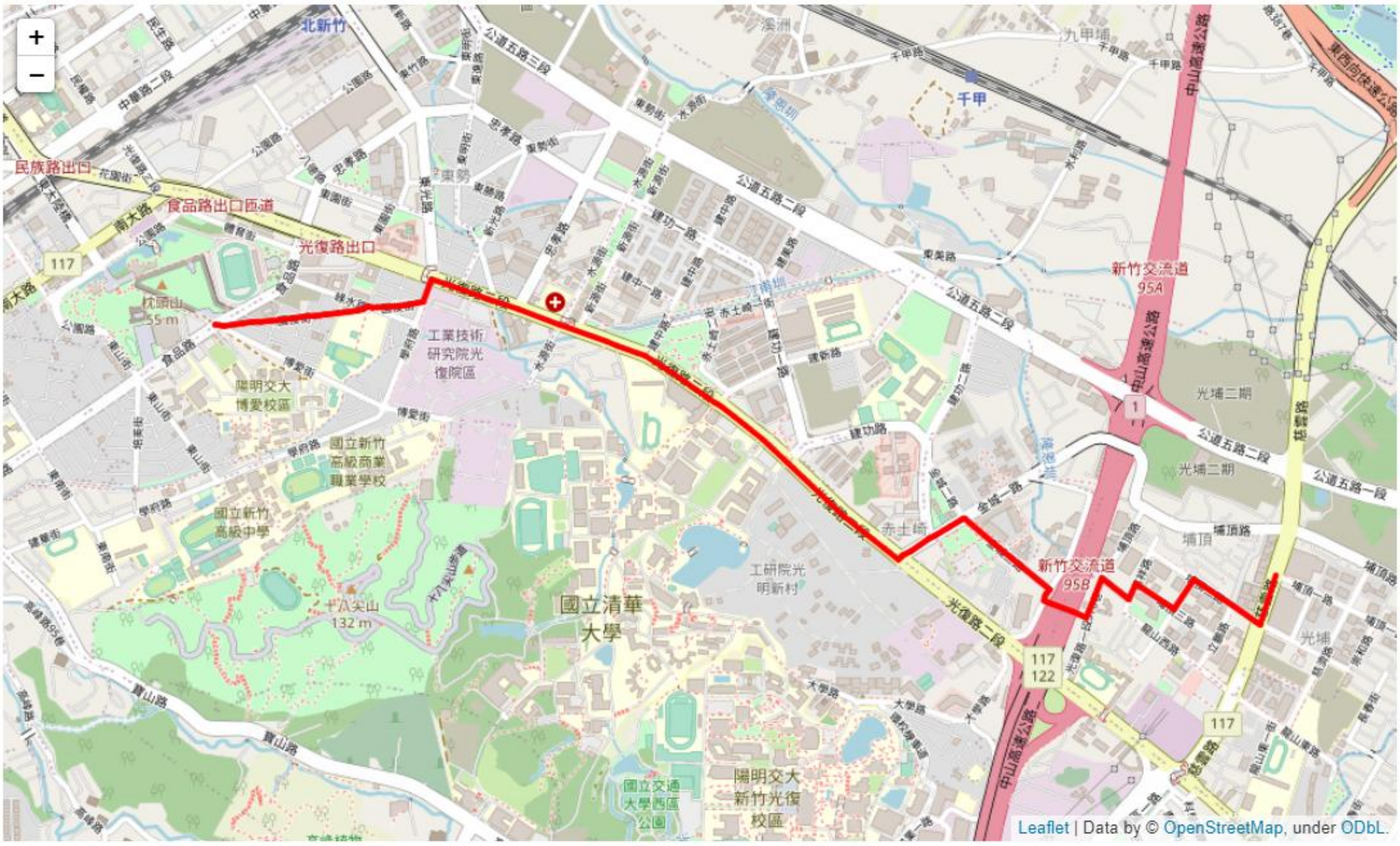


UCS

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: 7454

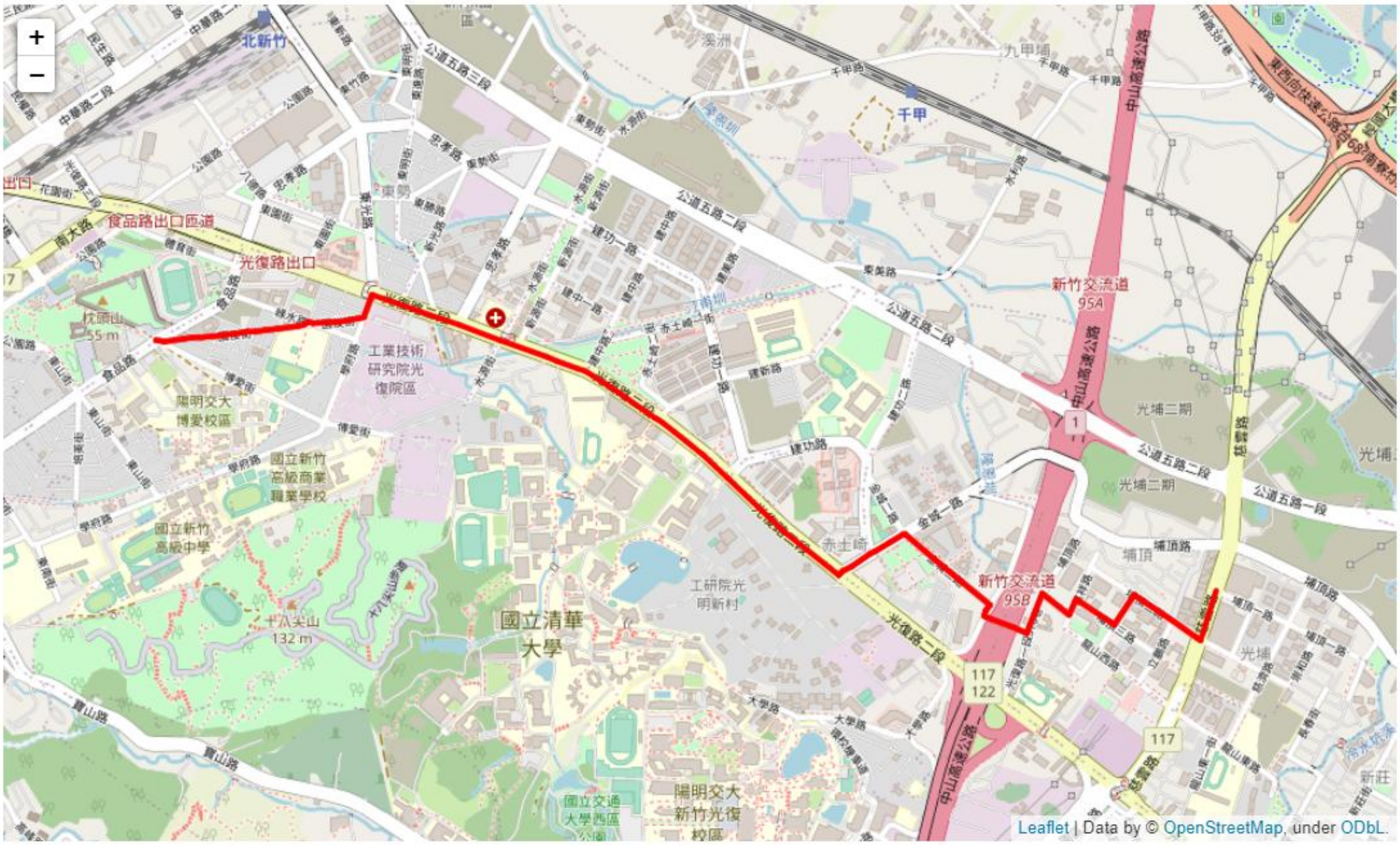
A*

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: 1245



A* (Bonus)

The number of nodes in the path found by A* search: 63
Total second of path found by A* search: 304.4436634360303 s
The number of visited nodes in A* search: 2955



Test 3：

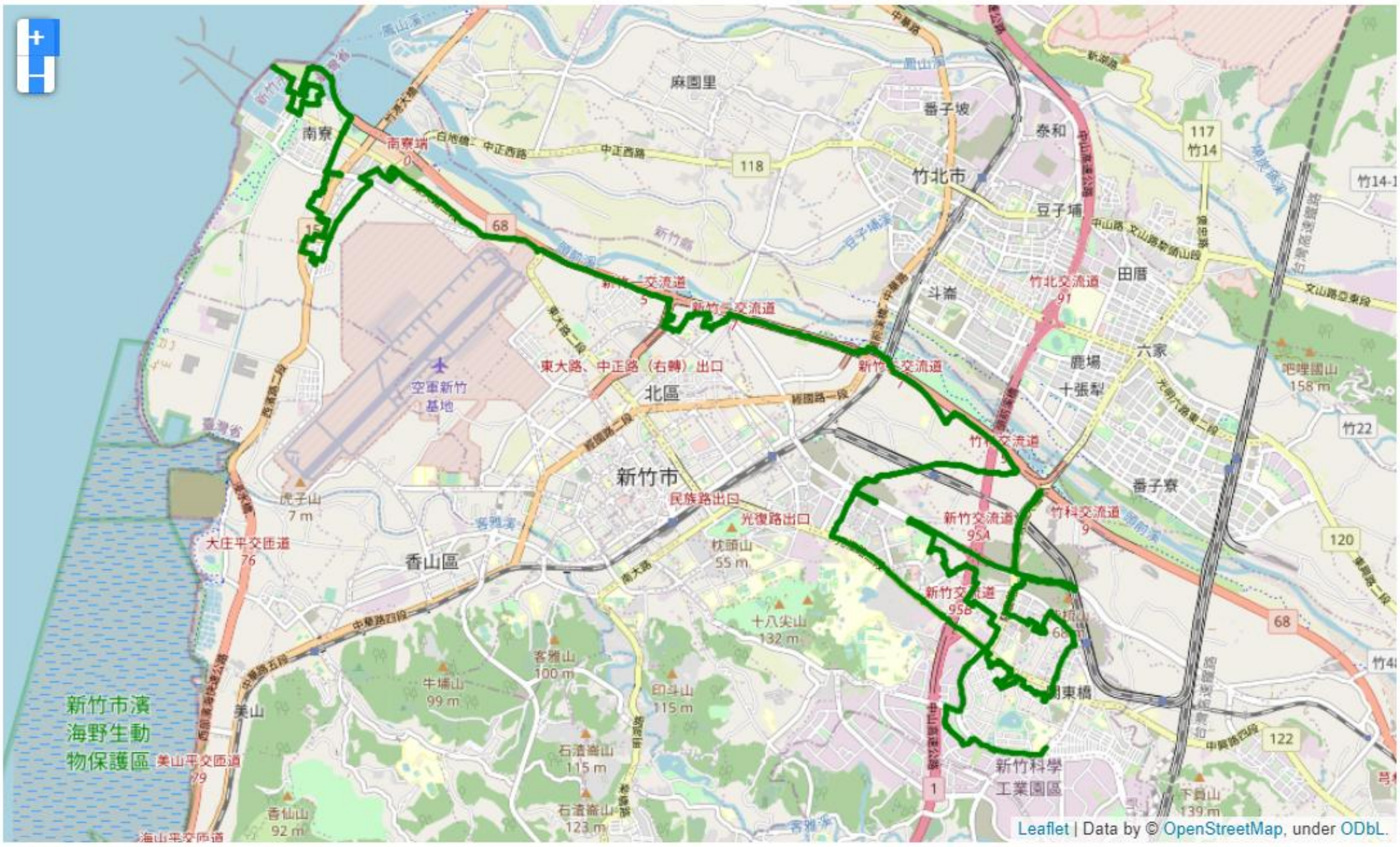
from National Experimental High School At Hsinchu Science Park (ID: 1718165260) to Nanliao Fighing Port (ID: 8513026827)

BFS

The number of nodes in the path found by BFS: 183
Total distance of path found by BFS: 15442.394999999995 m
The number of visited nodes in BFS: 11216

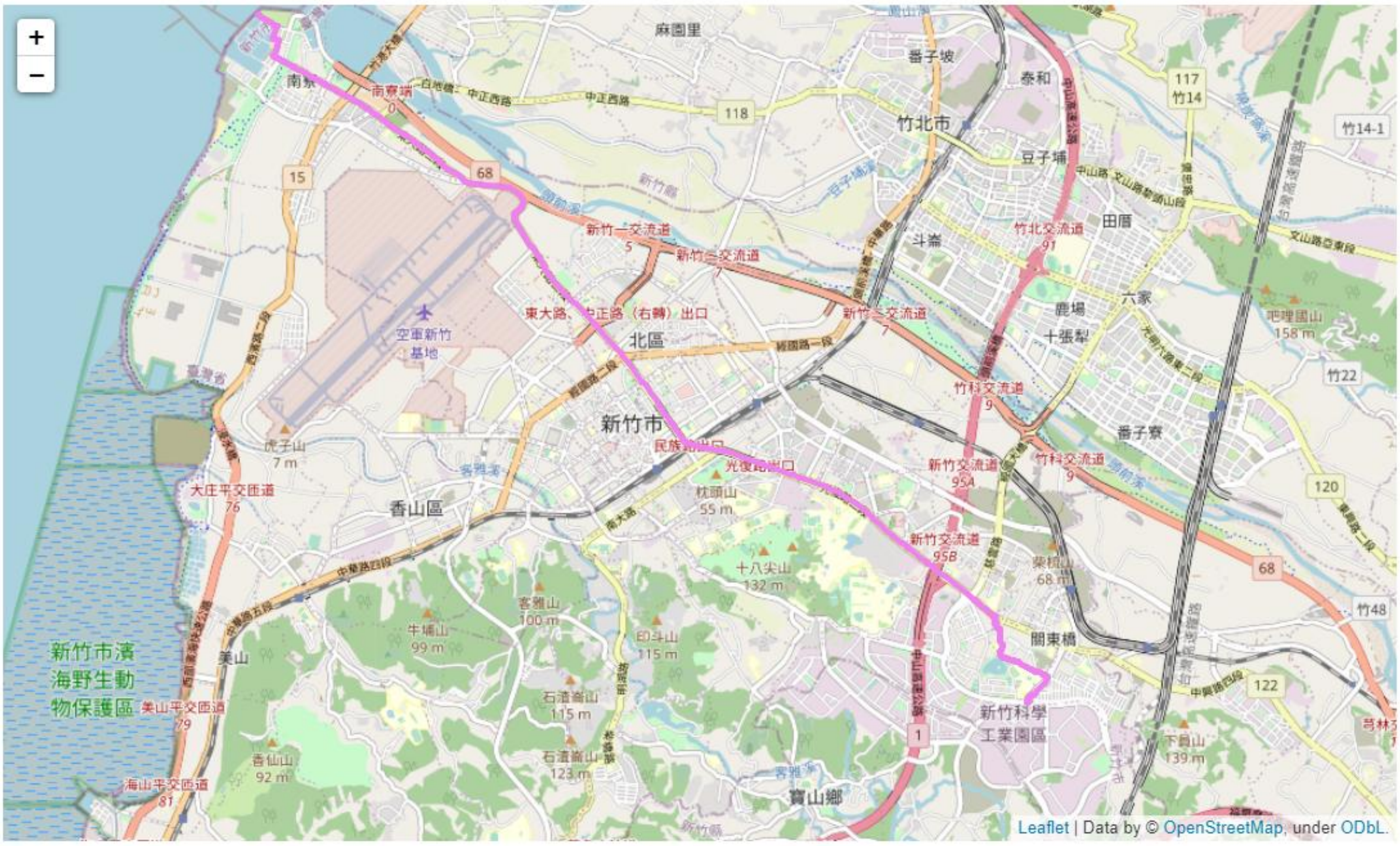
DFS (stack)

The number of nodes in the path found by DFS: 900
Total distance of path found by DFS: 39219.993000000024 m
The number of visited nodes in DFS: 2247



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: 12312



A*

The number of nodes in the path found by A* search: 288
Total distance of path found by A* search: 14212.41299999997 m
The number of visited nodes in A* search: 7571

A* (Bonus)

The number of nodes in the path found by A* search: 209
Total second of path found by A* search: 779.5279228368471 s
The number of visited nodes in A* search: 8727

With these results we know some things. DFS is very bad way to deal this problem since we will take a detour. BFS may be a acceptable way, at least it looks like it keeps getting closer to the target. The better way is use UCS which can find shortest path. That is enough for not far target. But when the target is far away, UCS will takes too much time to calculate the shortest path. So this time we need to use A* that can calculate the shortest path in shorter time.

In bonus, we take the staight line distance / the maximun speed limit in this graph to be heuristic function which is admissible. Since there is no faster way to touch the target than walk on staight line with highest speed. That means we will not overestimate it to let we miss the shortest path.

Part III. Question Answering

1. Please describe a problem you encountered and how you solved it.
    - Test 3 has a strange result.
      We did not remove newlines symbol for the latest data of every line. Remove it.
    - UCS has a result with longest path
      We did not update the shortest distance for the nodes walked. Mark the shortest path for every node at that time.
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
      If this road has more trffic light or the time of traffic light is longer, this path may be a slower path.
    - Road congestion
      If this road has a lot of cars, we need to take a lower speed to drive.
      For the both attribute, we need to stop and wait for some time that will let us touch the target be slower.
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
      By the photo which taked by satellite in the space can see the roads, buildings, ... Then we can read, analyze and redraw the data of these pictures, then mark some attribute for roads, buildings .... Done.
    - localization components
      For our smartphones can provide GPS positioning services by sending signals to satellites. Then we can knoe where am I for that map. Done.
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
   Use same way in my A* bonus to get the time between the delivery man and the target. Let the staight line distance / the maximun speed limit in this graph to be heuristic function.

   A: the time of above way when target is restaurant
   B: the prep time provided by the restaurant
   C: the time of above way when target is user's place

   If the delivery man does not take meals, ETA will be max (A + B) + C

   If the delivery man have taken meals, ETA will be C.