Homework 2: Route Finding (109550171)

Part I. Implementation (6%):

I used comments to explain my implementation.

Part 1

```python
def bfs(start, end):
    # Begin your code (Part 1)
    adjacency={} # create an adjacency list
    with open(edgeFile, newline='') as csvfile: # open the csvfile
        rows = csv.DictReader(csvfile) # open with "dictionary" data structure
        for row in rows:
            key = int(row['start']) # "key" represents the start node
            value = [int(row['end']),float(row['distance'])] # "value" represents the list of end node and distance
            if key not in adjacency.keys():
                adjacency[key] = [] # encouter the key at the first time -> create a new list for the key
            adjacency[key].append(value) # update the adjacency list
    # setup initial values
    totaldist = 0.0
    visited = []
    parent = {}
    dist = {}
    queue = []
    queue.append(start)
    visited.append(start)
    while queue:
        top = queue.pop(0) # get the first one of the queue
        if top == end: # if 'top' equal to 'end', we started to evaluate the result.
            path = [end]
            while path[-1] != start:
                totaldist += dist[path[-1]] # calculate the total distance
                path.append(parent[path[-1]]) # find the path from 'end' to 'start'
            path.reverse()
            return path, totaldist, len(visited)
        if top in adjacency:
            for ad in adjacency[top]: # a loop for every adjacency node
                if ad[0] not in visited: # only handle the node which hasn't visited
                    queue.append(ad[0]) # add into the queue
                    parent[ ad[0] ] = top # record the parent of this adjacency node
                    dist[ ad[0] ] = ad[1] #record the distance from 'top' to this adjacency node
                    visited.append(ad[0])
    # End your code (Part 1)
```

Part 2

```python
def dfs(start, end):
    # Begin your code (Part 2)
    adjacency={} # create an adjacency list
    with open(edgeFile, newline='') as csvfile: # open the csvfile
        rows = csv.DictReader(csvfile) # open with "dictionary" data structure
        for row in rows:
            key = int(row['start']) # "key" represents the start node
            value = [int(row['end']),float(row['distance'])] # "value" represents the list of end node and distance
            if key not in adjacency.keys():
                adjacency[key] = [] # encouter the key at the first time -> create a new list for the key
            adjacency[key].append(value) # update the adjacency list
    # setup initial values
    totaldist = 0.0
    visited = []
    parent = {}
    dist = {}
    stack = []
    stack.append(start)
    while stack:
        top = stack.pop() # get the top one of the stack
        if top == end: # if 'top' equal to 'end', we started to evaluate the result.
            path = [end]
            while path[-1] != start:
                totaldist += dist[path[-1]] # calculate the total distance
                path.append(parent[path[-1]]) # find the path from 'end' to 'start'
            path.reverse()
            return path, totaldist, len(visited)
        if top in adjacency:
            for ad in adjacency[top]: # a loop for every adjacency node
                if ad[0] not in visited: # only handle the node which hasn't visited
                    stack.append(ad[0]) # add into the stack
                    parent[ ad[0] ] = top # record the parent of this adjacency node
                    dist[ ad[0] ] = ad[1] #record the distance from 'top' to this adjacency node
                    visited.append(ad[0])
    # End your code (Part 2)
```

Part 3

```python
def ucs(start, end):
    # Begin your code (Part 3)
    adjacency={} # create an adjacency list
    with open(edgeFile, newline='') as csvfile: # open the csvfile
        rows = csv.DictReader(csvfile) # open with "dictionary" data structure
        for row in rows:
            key = int(row['start']) # "key" represents the start node
            value = [int(row['end']),float(row['distance'])] # "value" represents the list of end node and distance
            if key not in adjacency.keys():
                adjacency[key] = [] # encouter the key at the first time -> create a new list for the key
            adjacency[key].append(value) # update the adjacency list
    # setup initial values
    parent = {}
    queue = []
    visited =[]
    dist = 0.0
    now =  start
    while now != end: # if 'now' not equal to 'end', we continue the process
        if (now in adjacency) and (now not in visited):
            visited.append(now)
            for ad in adjacency[now]: # a loop for every adjacency node
                if ad[0] not in visited: # ad[0]: the adjacency node, ad[1]: the distance from 'now' to ad[0]
                    # dist(recorded at last iteration) + dist(from 'now' to ad[0]) = total distance from 'start' to ad[0]
                    queue.append([ dist + ad[1], ad[0] ])
                    parent[ ad[0] ] = now # record the parent of this adjacency node
        queue = sorted(queue) # sort to find the minimum distance
        now =  queue[0][1] # record the node, and expend road using its adjacency nodes at next iteration
        dist = queue[0][0] # record the distance, it will use when next iteration
        queue.pop(0)
    path = [end] # finish the while loop, and evaluate the result
    while path[-1] != start:
        path.append(parent[path[-1]]) # find the path from 'end' to 'start'
    path.reverse()
    return path, dist, len(visited)
    # End your code (Part 3)
```

Part 4

```python
def astar(start, end):
    # Begin your code (Part 4)
    adjacency={} # create an adjacency list
    with open(edgeFile, newline='') as csvfile: # open the csvfile
        rows = csv.DictReader(csvfile) # open with "dictionary" data structure
        for row in rows:
            key = int(row['start']) # "key" represents the start node
            value = [int(row['end']),float(row['distance'])] # "value" represents the list of end node and distance
            if key not in adjacency.keys():
                adjacency[key] = [] # encouter the key at the first time -> create a new list for the key
            adjacency[key].append(value) # update the adjacency list
    heuristic = {}
    with open(heuristicFile, newline='') as hfile:
        rows = csv.DictReader(hfile) # open with "dictionary" data structure
        for row in rows:
            key = int(row['node']) # "key" represents the node
            value = float(row[str(end)]) #"value" represents the distance from this node to 'end'
            heuristic[key] = value
    # setup initial values
    parent = {}
    queue = []
    visited =[]
    visited.append(start)
    for ad in adjacency[start]:
        queue.append([ad[1]+heuristic[ad[0]], ad[1], start, ad[0]])
    while 1:
        queue = sorted(queue) # sort to find the minimum (distance + heuristic function value)
        now =  queue[0][3] # record the node, and expend road using its adjacency nodes at next iteration
        dist = queue[0][1] # record the distance from 'start' to 'now'
        par = queue[0][2] # record the parent of 'now'
        queue.pop(0)
        if now in adjacency and now not in visited:
            visited.append(now)
            parent[now] = par
            for ad in adjacency[now]:
                if ad[0] not in visited:
                    # it's like ucs search, but considering the heuristic function value
                    # record distance + heuristic,  distance only, parent, adjacency node at the same time
                        queue.append([ dist + ad[1] + heuristic[ad[0]] ,dist + ad[1], now, ad[0] ])
        if now == end:  # if 'now' not equal to 'end', we continue the process
            break
    path = [end] # finish the while loop, and evaluate the result
    while path[-1] != start:
        path.append(parent[path[-1]]) # find the path from 'end' to 'start'
    path.reverse()
    return path, dist, len(visited)
    # End your code (Part 4)
```

Part II. Results & Analysis (12%):

BFS:

This algorithm doesn't consider the distance between two nodes, so it needs longer path than UCS and A* (can't find shortest path).

DFS:

This algorithm has more visited nodes and longer path because it visits the whole path even it doesn't contain the target.

UCS:

This algorithm takes distance between two nodes into consider, so it can find the shortest path. But it doesn't consider the distance from target, it must visit more nodes than A*.

A*:

This algorithm considers distance between two nodes and distance from target at the same time. We can use it to find the shortest path with less visited nodes than UCS.

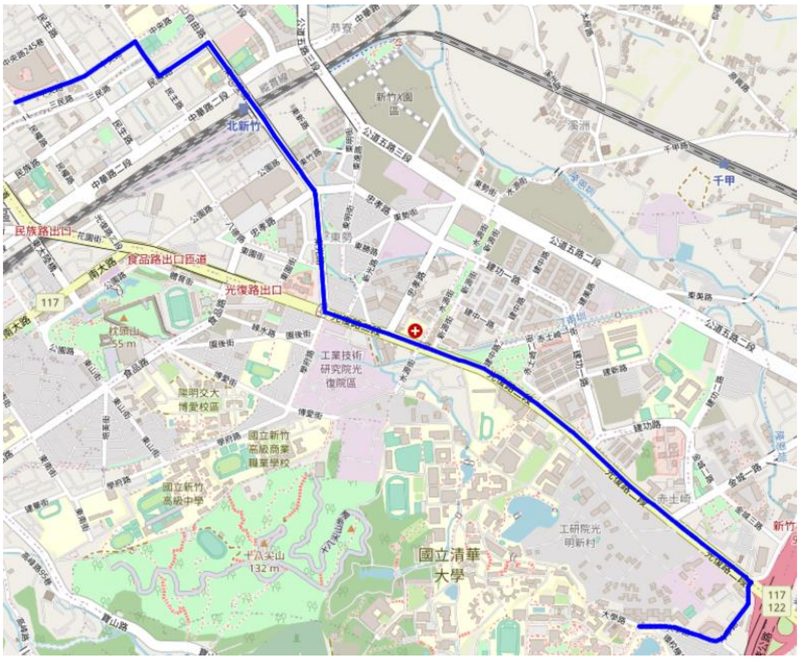
A*_time:

Besides distance between two nodes and distance from target, this algorithm considers the speed limit, and it is closer to our reality world.

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: 4978.881999999998 m
The number of visited nodes in BFS: 4403

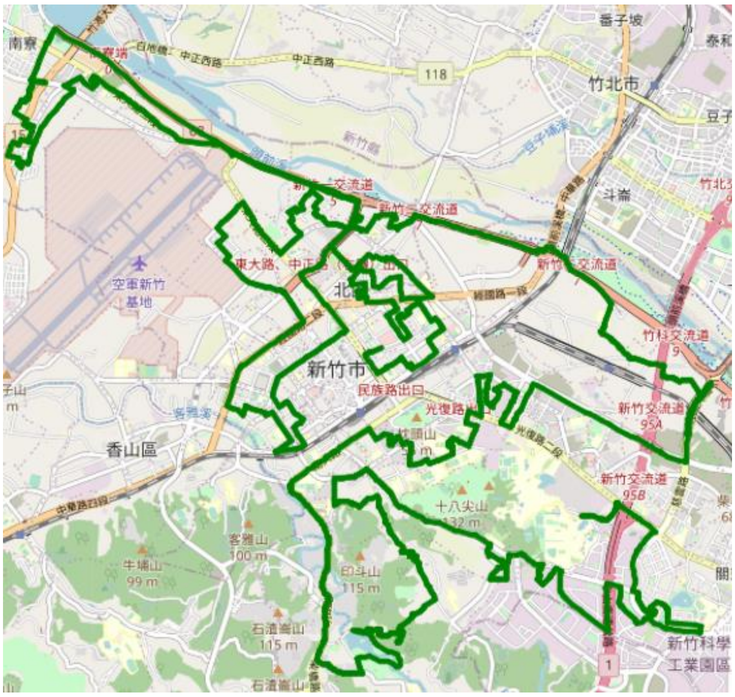


DFS (stack):

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: 5236

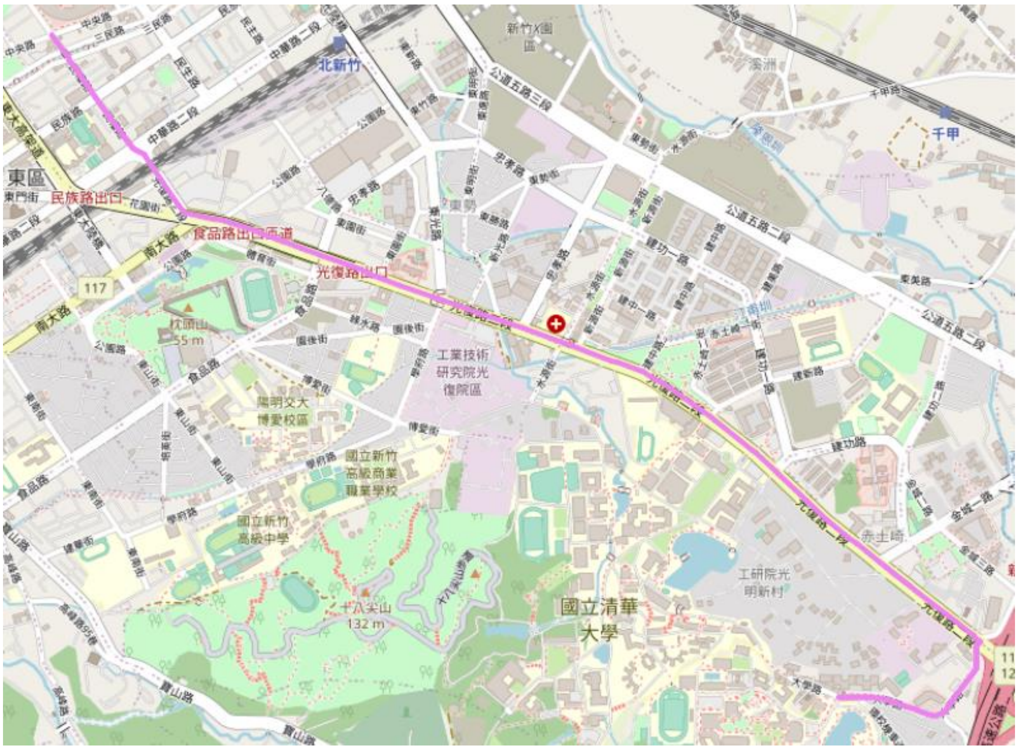


UCS:

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: 5076

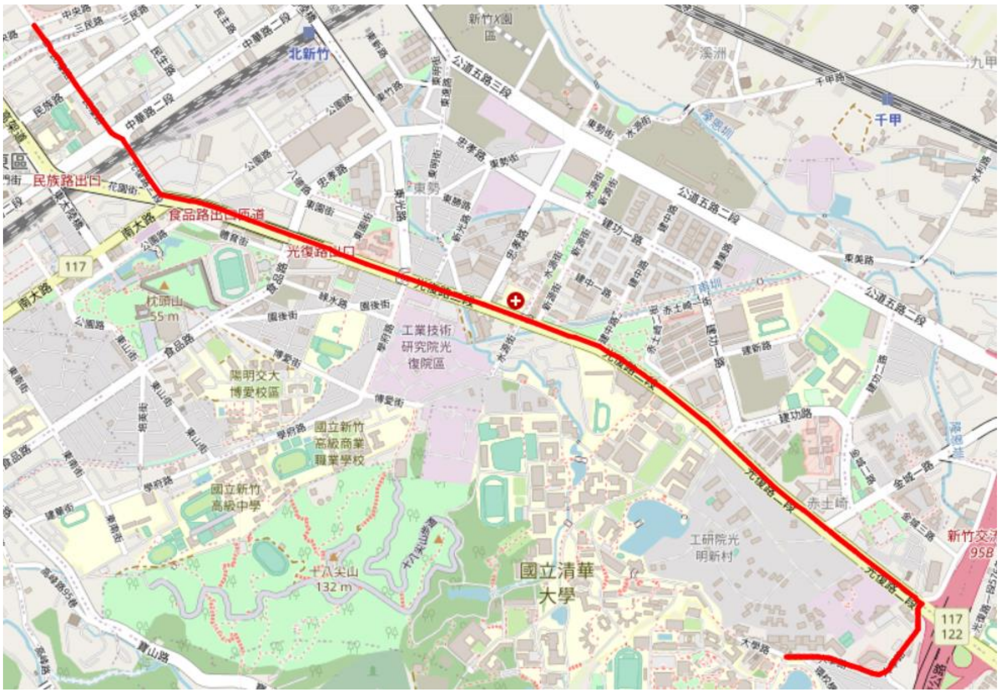


A* Search:

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



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: 807

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: 4215.521000000001 m

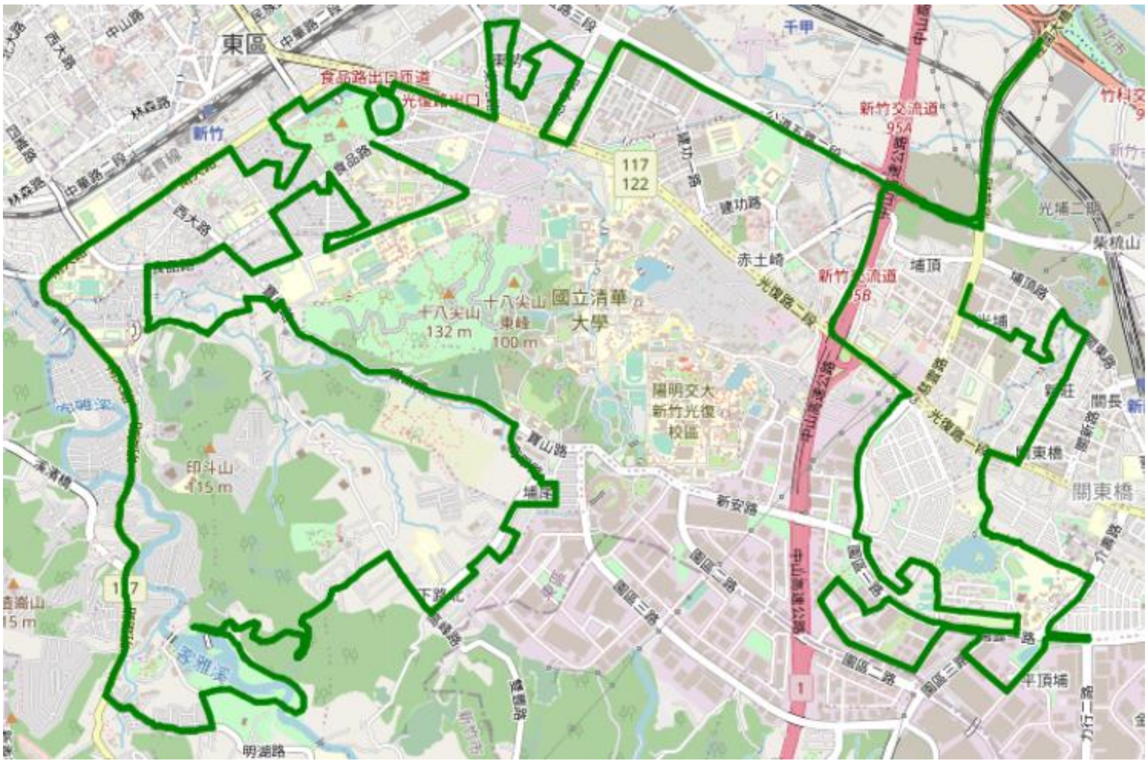
The number of visited nodes in BFS: 4752

DFS:

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: 9616



UCS:

The number of nodes in the path found by UCS: 72

Total distance of path found by UCS: 4101.84 m

The number of visited nodes in UCS: 7206

A* Search:

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: 1171

A*_time:

The number of nodes in the path found by A* search: 63

Total second of path found by A* search: 310.83006307091864 s

The number of visited nodes in A* search: 1678

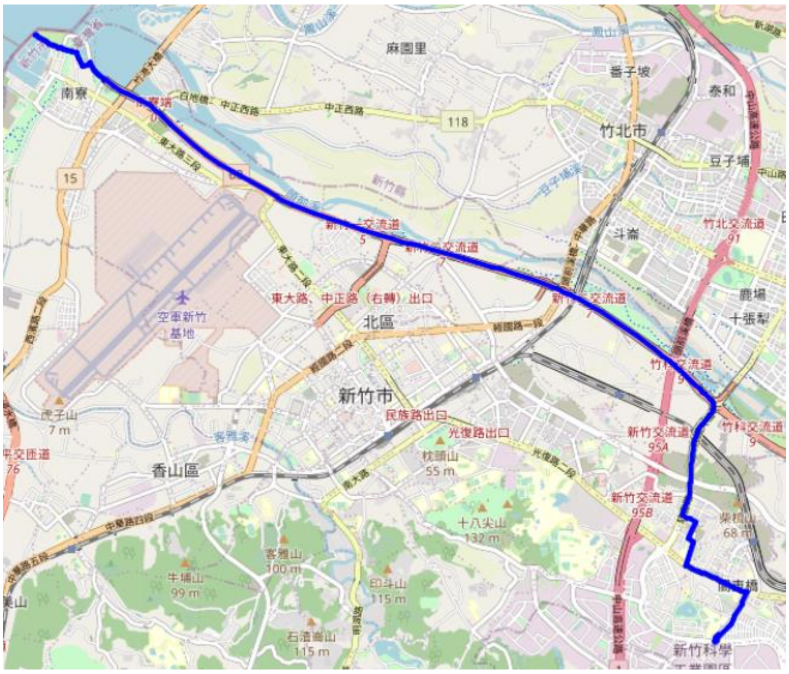
Test3: from National Experimental High School At Hsinchu Science Park (ID: 1718165260) to Nanliao Fighing Port (ID: 8513026827)

BFS:

The number of nodes in the path found by BFS: 183

Total distance of path found by BFS: 15442.394999999995 m

The number of visited nodes in BFS: 11266

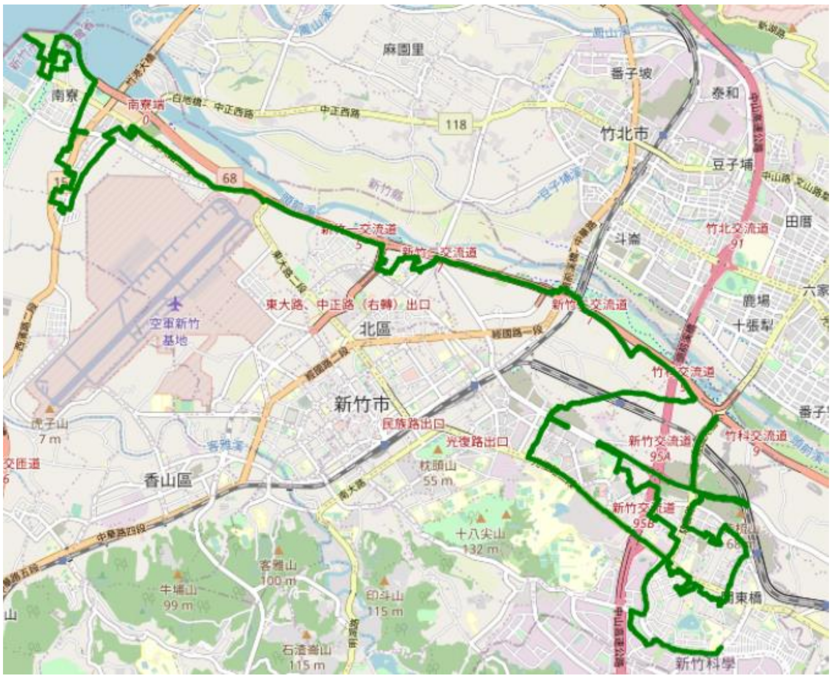


DFS:

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: 2493

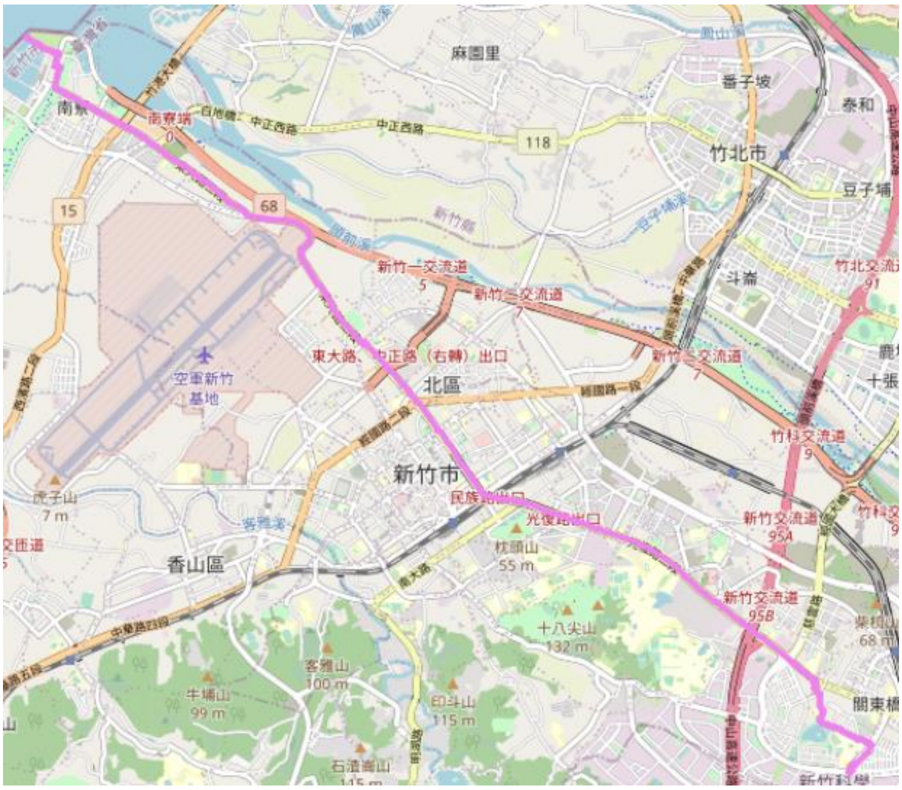


UCS:

The number of nodes in the path found by UCS: 303

Total distance of path found by UCS: 14212.412999999997 m

The number of visited nodes in UCS: 11908

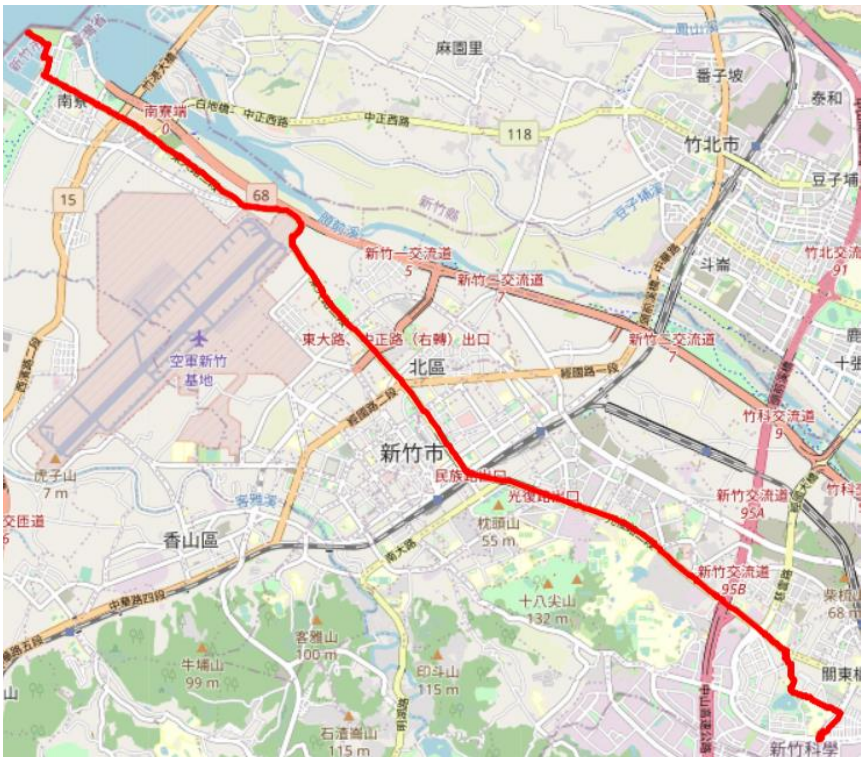


A* Search:

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: 7067



A*_time:

The number of nodes in the path found by A* search: 207

Total second of path found by A* search: 781.217356759769 s

The number of visited nodes in A* search: 7787

Part III. Question Answering (12%):

1. Please describe a problem you encountered and how you solved it.
   When implementing BFS and DFS in the beginning, I didn't know what situation I had to mark the node as visited, resulting in infinite loop.
   I drew the process of the algorithms and determined the situation to mark visited.

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.

   How traffic is jammed.
   If there is a shortest road but this road is very busy during the rush hour, we might choose another road which is not busy .

3. As mentioned in the introduction, a navigation system involves mapping, localization, and route finding. Please suggest possible solutions for mapping and localization components?

   - **Topological maps:** focusing on the connectivity of the environment rather than creating a geometrically accurate map.
   - **Satellite navigation device:** it can receive information from GNSS satellites and calculate the device's geographical position.

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
   = (the straight-line distance from current node to target) divides (speed*)
   The speed* considers the current speed of the driver and how the traffic is jammed.
   If the future path is busy, the speed* decrease.
   If the future path is light, the speed* increase.
   The speed* updates every 5 seconds.
   I define this function because it can adjust the speed depending on the traffic is busy or not.