109550069 林宇柔

HW2 Report

I. Implementation (Part 1 – Part 4)

```
class n:
    def __init__(self, End, distance, limit):  # define a class to store the data of an edge
        self.End = End  # including the end, the distance, and the speed limit
        self.distance = distance
        self.limit = limit
```

1. BFS

```
def bfs(start, end):
    # Begin your code (Part 1)

    # read the data
    EdgeFile = open(edgeFile)  # open 'edges.csv'
    lines = EdgeFile.readlines()  # read them as separate lines
    # create a dictionary to store the edges from the same start
    graph = defaultdict(list)
    # create a dictionary to see which has been visited
    visited = defaultdict(str)

    for line in lines:
        data = line.split(",")  # store the data from csv file
        Node = n(data[1], data[2], data[3].replace("\n", ""))
        graph[data[0]].append(Node)
        visited[data[0]] = False  # set all elements to not visited
    # bfs
    q = []  # create a queue for bfs
    q.append(start)  # push the start node inside
    visited[str(start)] = True  # mark the start as visited
    parent = defaultdict(str)  # create a dictionary to store the previous node
    parent[str(start)] = None  # start with no previous node
    num_visited = 0
    while len(q) != 0:  # while the queue is not empty
        # set the current node to the top element of the queue (FIFO)
        current = str(q.pop(0))
        # for every edge starting from the current node
        for i in range(len(graph[current])):
            next = graph[current][i].End  # set the next node
            if visited[next] == False:  # if the next node is not visited
                # set the previous of the next to current
                parent[next] = current
                visited[str(next)] = True  # visited
                num_visited += 1  # so the number of visited nodes increases
                q.append(next)  # push the next node into the queue
                if next == str(end):  # if this edge reaches the end
                    del q[:]  # break the loop
                    break
```

```
    path = []  # create a queue for storing the path
    path.append(str(end))  # push the end node inside
    dist = 0  # initialize the distance to 0
    # now = end  # set the current node to 'end'
    turn = 0  # the first iteration
    while parent[path[turn]] != None:  # while the current node is not 'start'
        # print(parent[path[turn]])
        # for edges starting from the previous node
        for i in range(len(graph[parent[path[turn]]])):
            if graph[parent[path[turn]]][i].End == path[turn]:  # if the End is the current node
                # add up the distance
                dist += float(graph[parent[path[turn]]][i].distance)
        turn += 1  # next iteration
        path.append(parent[path[turn-1]])  # add the previous node to the path
        # now = parent[now]  # set the current node to the previous

    path.reverse()  # reverse because it was stored bakwards
    for i in range(len(path)):  # change all the elements in the path to integers
        path[i] = int(path[i])

    return path, dist, num_visited  # return

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

2. DFS_stack

```
def dfs(start, end):
    # Begin your code (Part 2)

    # read the data
    EdgeFile = open(edgeFile)  # open 'edges.csv'
    lines = EdgeFile.readlines()  # read them as separate lines
    # create a dictionary to store the edges from the same start
    graph = defaultdict(list)
    # create a dictionary to see which has been visited
    visited = defaultdict(str)

    for line in lines:
        data = line.split(",")  # store the data from csv file
        Node = n(data[1], data[2], data[3].replace("\n", ""))
        graph[data[0]].append(Node)
        visited[data[0]] = False  # set all elements to not visited
    # dfs
    q = []  # create a queue for dfs
    q.append(start)  # push the start node inside
    visited[str(start)] = True  # mark the start as visited
    parent = defaultdict(str)  # create a dictionary to store the previous node
    parent[str(start)] = None  # start with no previous node
    num_visited = 0
```

```
    while len(q) != 0:  # while the queue is not empty
        # set the current node to the last element of the queue (FILO)
        current = str(q.pop())
        # for every edge starting from the current node
        for i in range(len(graph[current])):
            next = graph[current][i].End  # set the next node
            if visited[next] == False:  # if the next node is not visited
                # set the previous of the next to current
                parent[next] = current
                visited[str(next)] = True  # visited
                num_visited += 1  # so the number of visited nodes increases
                q.append(next)  # push the next node into the queue
                if next == str(end):  # if this edge reaches the end
                    del q[:]  # break the loop
                    break

    path = []  # create a queue for storing the path
    path.append(str(end))  # push the end node inside
    dist = 0  # initialize the distance to 0
    # now = end  # set the current node to 'end'
    turn = 0  # the first iteration
    while parent[path[turn]] != None:  # while the current node is not 'start'
        # print(parent[path[turn]])
        # for edges starting from the previous node
        for i in range(len(graph[parent[path[turn]]])):
            if graph[parent[path[turn]]][i].End == path[turn]:  # if the End is the current
                # add up the distance
                dist += float(graph[parent[path[turn]]][i].distance)
        turn += 1  # next iteration
        path.append(parent[path[turn-1]])  # add the previous node to the path
        # now = parent[now]  # set the current node to the previous

    path.reverse()  # reverse because it was stored bakwards
    for i in range(len(path)):  # change all the elements in the path to integers
        path[i] = int(path[i])

    return path, dist, num_visited  # return

    raise NotImplementedError("To be implemented")
    # End your code (Part 2)
```

3. UCS

```
def ucs(start, end):
    # Begin your code (Part 3)

    # read data
    EdgeFile = open('edges.csv')  # open 'edges.csv'
    lines = EdgeFile.readlines()  # read them as separate lines
    # create a dictionary to store the edges from the same start
    graph = defaultdict(list)
    # create a dictionary to see which has been visited
    visited = defaultdict(str)
```

```
    for line in lines:
        data = line.split(",")  # store the data from csv file
        Node = n(data[1], data[2], data[3].replace("\n", ""))
        graph[data[0]].append(Node)
        graph[data[0]].sort(key=lambda Node: Node.End)
        visited[data[0]] = False  # set all elements to not visited

    # ucs
    q = []  # create a list for ucs
    q.append(start)  # push the start node inside
    visited[str(start)] = True  # mark the start as visited
    # create a dictionary to store the distance from 'start'
    distance = defaultdict(float)
    distance[start] = 0  # the distace from 'start' to 'start' is 0
    parent = defaultdict(str)  # create a dictionary to store the previous node
    parent[start] = None  # start with no previous node
    num_visited = 0
    while len(q) != 0:  # while the queue is not empty
        min = float("inf")  # set the minimum to infinity
        for i in range(len(q)):  # for the distances to the ends in current queue
            if distance[q[i]] < min:  # if it is smaller than the minimum
                min = distance[q[i]]  # update the minimum
                m_index = i  # record the index
        # set current node to the one with minimum distance
        current = str(q.pop(m_index))
        visited[current] = True  # mark as visited
        num_visited += 1  # so the number of visited nodes increases
        if current == str(end):  # if the node is 'end'
            del q[:]  # break the loop
            # set the total distance to the distance to 'end'
            dist = distance[str(end)]
            break
        # for every edge starting from current node
        for i in range(len(graph[current])):
            # set the end of this edge to the next node
            next = graph[current][i].End
            if visited[next] == False:  # if the next is not visited
                if next not in q:  # and if it is not in the queue yet
                    q.append(next)  # add it to the queue
                if next in distance:  # if it already has a distance
                    if distance[next] > distance[current] + float(graph[current][i].distance):
                        # and if there exists smaller distance, update
                        parent[next] = current
                        distance[next] = distance[current] + \
                            float(graph[current][i].distance)
                else:  # if next does not have a recorded distance yet
                    parent[next] = current  # do the same as above
                    distance[next] = distance[current] + \
                        float(graph[current][i].distance)
        # set the used distance back to infinity
        distance[m_index] = float("inf")
```

```
    path = []  # create a queue for storing the path
    path.append(str(end))  # push the end node inside
    turn = 0  # the first iteration
    while path[turn] != str(start):  # while the current node is not 'start'
        path.append(parent[path[turn]])  # add the previous node to the path
        turn += 1  # next iteration

    path.reverse()  # reverse because it was stored bakwards
    for i in range(len(path)):  # change all the elements in the path to integers
        path[i] = int(path[i])

    return path, dist, num_visited  # return

    raise NotImplementedError("To be implemented")
    # End your code (Part 3)
```

4. A*

```
def astar(start, end):
    # Begin your code (Part 4)

    # read data
    EdgeFile = open('edges.csv')  # open 'edges.csv'
    lines = EdgeFile.readlines()  # read them as separate lines
    # create a dictionary to store the edges from the same start
    graph = defaultdict(list)
    # create a dictionary to see which has been visited
    visited = defaultdict(str)

    for line in lines:
        data = line.split(",")  # store the data from csv file
        Node = n(data[1], data[2], data[3].replace("\n", ""))
        graph[data[0]].append(Node)
        graph[data[0]].sort(key=lambda Node: Node.End)
        visited[data[0]] = False  # set all elements to not visited

    # read heuristic
    Hfile = open(heuristicFile)  # open 'heuristic.csv'
    Lines = Hfile.readlines()  # read them as separate lines
    # create three dictionaries for three possible values
    val1 = defaultdict(float)
    val2 = defaultdict(float)
    val3 = defaultdict(float)

    for line in Lines:
        data = line.split(",")  # store the data from csv file
        val1[data[0]] = data[1]
        val2[data[0]] = data[2]
        val3[data[0]] = data[3].replace("\n", "")
```

```
    # A*
    q = []  # create a list for A*
    q.append(start)  # push the start node inside
    visited[str(start)] = True  # mark the start as visited
    # create a dictionary to store the distance from 'start'
    distance = defaultdict(float)
    distance[start] = 0  # the distace from 'start' to 'start' is 0
    parent = defaultdict(str)  # create a dictionary to store the previous node
    parent[start] = None  # start with no previous node
    add = defaultdict(float)  # create a dictionary to store the addition
    num_visited = 0

    if str(end) == '1079387396':  # check which value should be used
        value = val1
    elif str(end) == '1737223506':
        value = val2
    elif str(end) == '8513026827':
        value = val3
```

```
    while len(q) != 0:  # while the queue is not empty
        min = float("inf")  # set the minimum to infinity
        for i in range(len(q)):  # for the additions to the ends in current queue
            if add[q[i]] < min:  # if it is smaller than the minimum
                min = add[q[i]]  # update the minimum
                m_index = i  # record the index
        # set current node to the one with minimum distance
        current = str(q.pop(m_index))
        visited[current] = True  # mark as visited
        num_visited += 1  # so the number of visited nodes increases
        if current == str(end):  # if the node is 'end'
            del q[:]  # break the loop
            # set the total distance to the addition to 'end'
            dist = add[str(end)]
            break
        # for every edge starting from current node
        for i in range(len(graph[current])):
            # set the end of this edge to the next node
            next = graph[current][i].End
            if visited[next] == False:  # if the next is not visited
                if next not in q:  # and if it is not in the queue yet
                    q.append(next)  # add it to the queue
                if next in distance:  # if already exits
                    if distance[next] > distance[current] + float(graph[current][i].distance):
                        # and if there exists smaller distance, update
                        parent[next] = current
                        distance[next] = distance[current] + \
                            float(graph[current][i].distance)
                else:  # if next does not have a recorded distance yet
                    parent[next] = current  # do the same as above
                    distance[next] = distance[current] + \
                        float(graph[current][i].distance)
            add[next] = distance[next] + float(value[int(next)]) # add up the value and the distance
        add[m_index] = float("inf")  # set the used addition back to infinity
        distance[m_index] = float("inf") # set the used distance back to infinity
```

```
    path = []  # create a queue for storing the path
    path.append(str(end))  # push the end node inside
    turn = 0  # the first iteration
    while path[turn] != str(start):  # while the current node is not 'start'
        path.append(parent[path[turn]])  # add the previous node to the path
        turn += 1  # next iteration

    path.reverse()  # reverse because it was stored bakwards
    for i in range(len(path)):  # change all the elements in the path to integers
        path[i] = int(path[i])

    return path, dist, num_visited  # return

    raise NotImplementedError("To be implemented")
    # End your code (Part 4)
```

II. Results and Analysis (Part 5)

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: 4266
```

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: 5227
```

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: 5077
```

A*

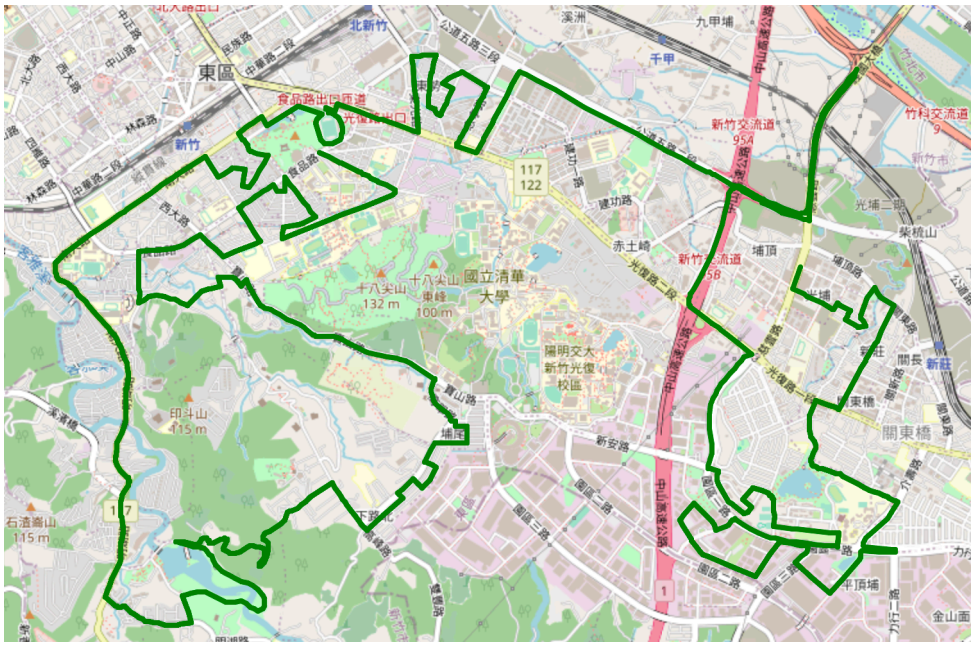

```
The number of nodes in the path found by A* search: 89
Total distance of path found by A* search: 4367.881 m
The number of visited nodes in A* search: 5077
```

2. From Hsinchu Zoo (ID: 426882161) to COSTCO Hsinchu Store (ID: 1737223506)

BFS

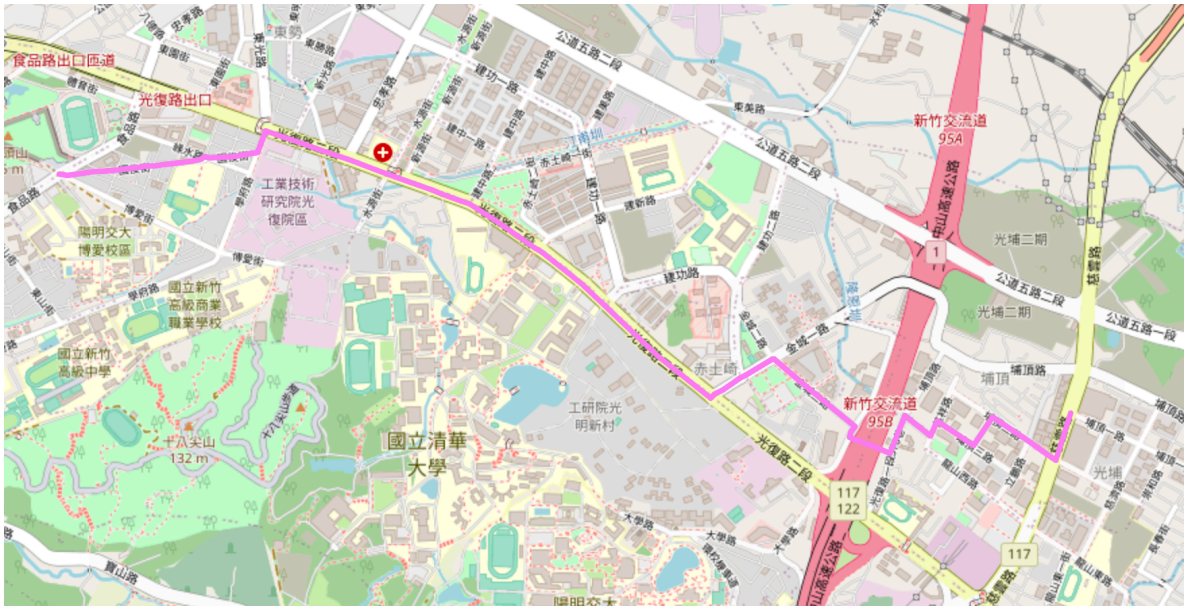

```
The number of nodes in the path found by BFS: 60
Total distance of path found by BFS: 4215.521000000001 m
The number of visited nodes in BFS: 4603
```

DFS_Stack



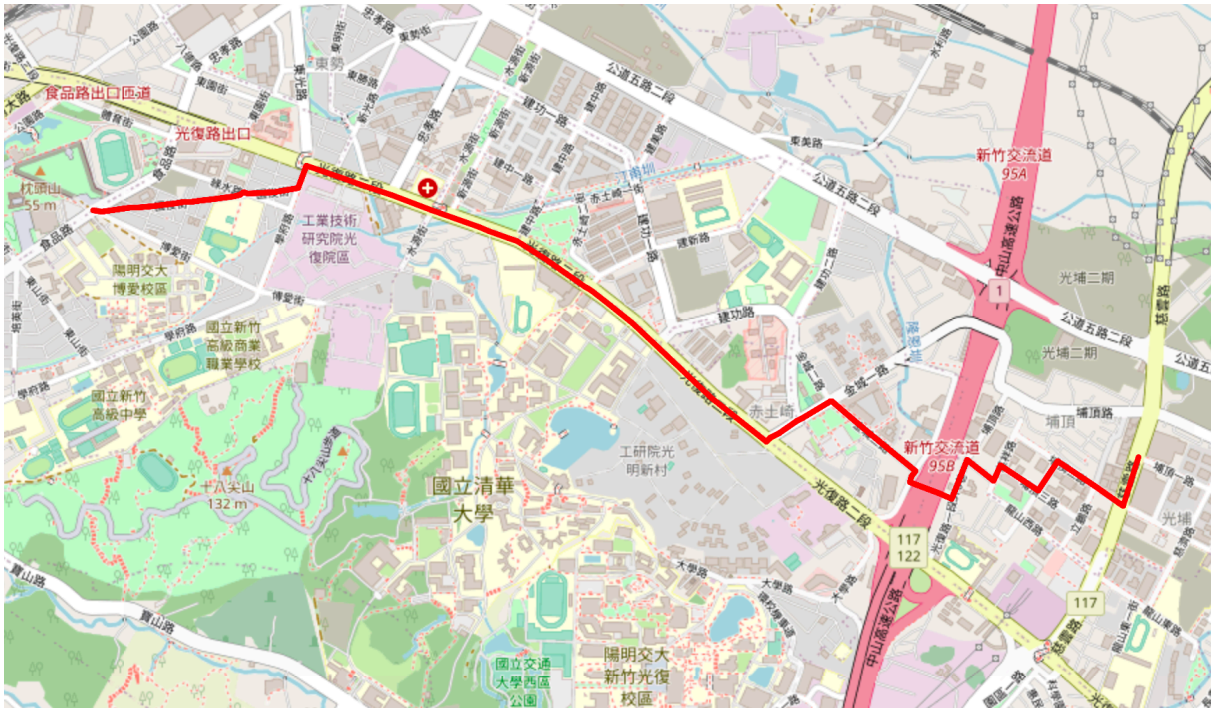

```
The number of nodes in the path found by DFS: 930
Total distance of path found by DFS: 38752.307999999895 m
The number of visited nodes in DFS: 9599
```

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: 7207
```

A*



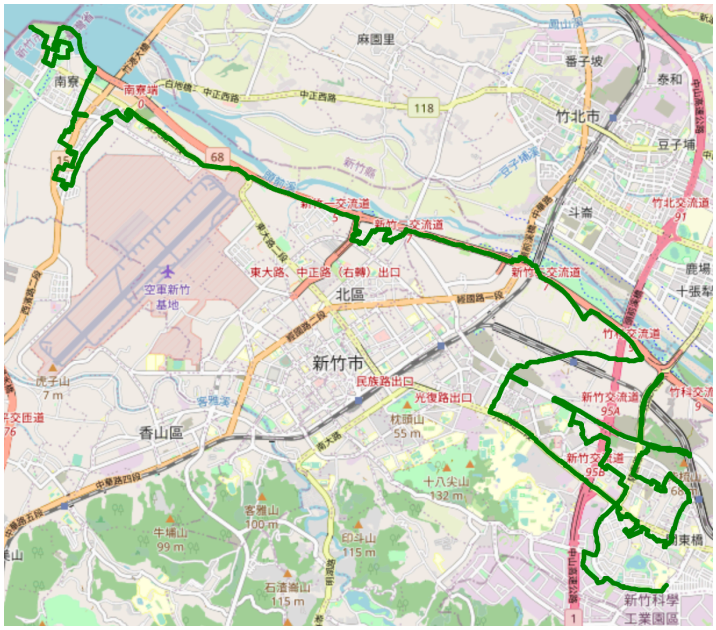

```
The number of nodes in the path found by A* search: 63
Total distance of path found by A* search: 4101.84 m
The number of visited nodes in A* search: 7207
```

3. From National Experimental High School At Hsinchu Science Park (ID: 1718165260) to Nanliao Fishing Port (ID: 8513026827)

BFS

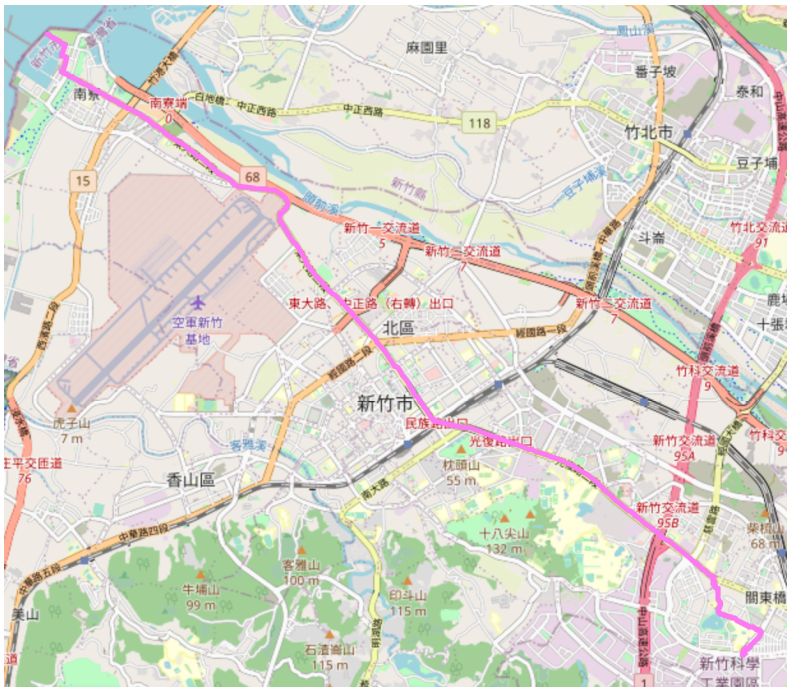

```
The number of nodes in the path found by BFS: 183
Total distance of path found by BFS: 15442.394999999995 m
The number of visited nodes in BFS: 11226
```

DFS_stack



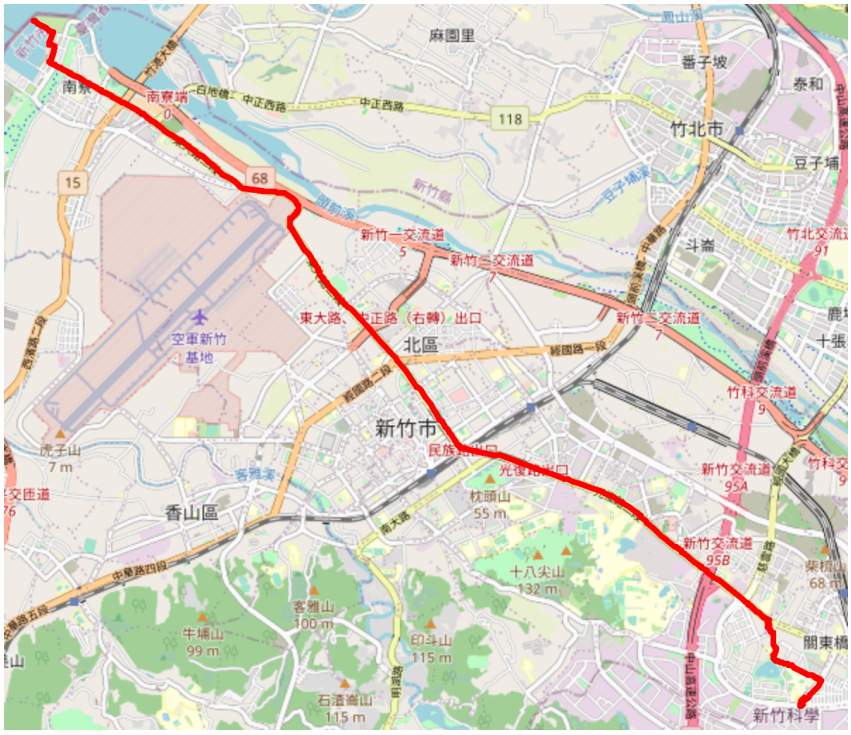
The number of nodes in the path found by DFS: 900
Total distance of path found by DFS: 39219.993000000024 m
The number of visited nodes in DFS: 2488

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: 11909

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: 11909
```

Comparison

Route 1	BFS	DFS_stack	UCS	A*
Number of nodes in the path	88	1718	89	89
Total distance	4978.88	75504.31	4367.881	4367.881
Number of visited nodes	4266	5667	5077	5077
Route 2	**BFS**	**DFS_stack**	**UCS**	**A***
Number of nodes in the path	60	930	63	63
Total distance	4215.52	38752.30	4101.84	4101.84
Number of visited nodes	4603	9599	7207	7207

Route 3	BFS	DFS_stack	UCS	A*
Number of nodes in the path	183	900	288	288
Total distance	15442.39	39219.99	14212.41	14212.41
Number of visited nodes	11226	2488	11909	11909

Discussion:

1. For each route, BFS takes the least number of node in the final path, and visits the least nodes during traversal.
2. For each route, UCS and A* share the same final path and therefore has the same number of nodes in the path and the same total distance. For all 4 algorithms, they take the shortest total distance. However, UCS should visit way more nodes than A* does during traversal, which I didn't successfully implement.
3. For each route, DFS using stack takes the most number of node in the final path, the largest total distance. For the first and the second route, it also visits the most nodes during traversal.

III. Question Answering

1. Problem Encountered:

   (1) For the previous lab, I used Anaconda as my programming environment, so this is my first time using Google Colab, and I don't really enjoy coding with it, you have to upload all the files to your drive and run the block for saving it every time or else it doesn't record your progress, also, the terminal wasn't really understandable for me. So I switched my environment to Visual Studio Code, then I encountered another problem that the version of Python in my VSCode was Python2 instead of Python3, so it couldn't compile several functions in my code. I tried to update it but it was helpless. At last, I tried to enter commands like "python3 bfs.py" in the terminal but not simply pressing the run button to ensure that it runs in python3, and it worked.

   (2) I think that after the first lab, I wasn't so sure about how Python works, however after this assignment, I feel like I'm starting to embrace the advantages of Python. For example, when I write BFS and DFS in C++, I always

have to include the header files of queue and stack separately, while in Python, I need neither of them because I can simply use a list (which is in a format of a queue) and pop the thing I want to decide if it is first–in–first–out or first–in–last–out. Also, I learned how to use the sorting function and the delete function, which I believe would be really helpful in the future.

2. Besides speed limit and distance, I think the other attribute that is essential for route finding is the traffic flow of each road.

3. For mapping, it means to construct a graph with all the places as nodes and the roads as edges. For each road, record its length and the start and end node of it. For localization, you record the starting point, then estimate the possible speed of the vehicle by knowing the speed limit of each road, the traffic flow of it, and the traffic signs located there.

4. h(x) = the total of all roads divided by its own speed limit –> This gives the fastest estimated time by expecting all the drivers would drive fast enough to meet the speed limit.