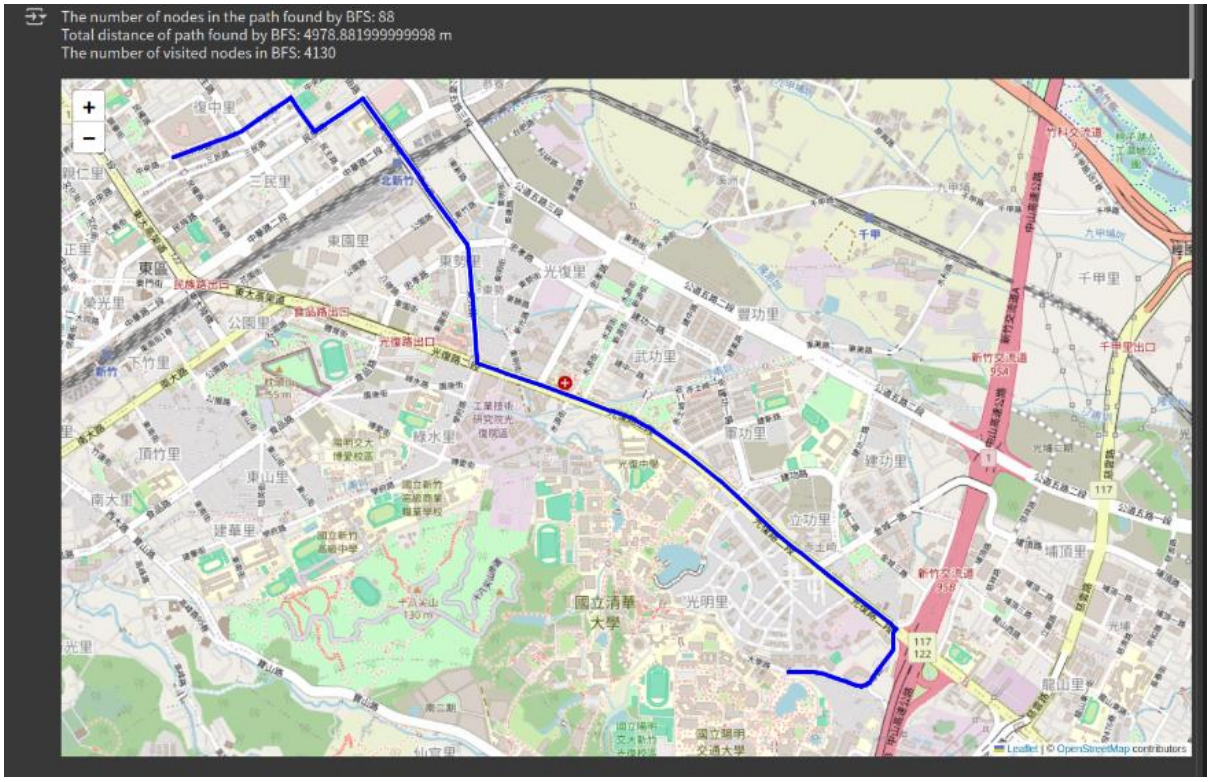
Test 1 :

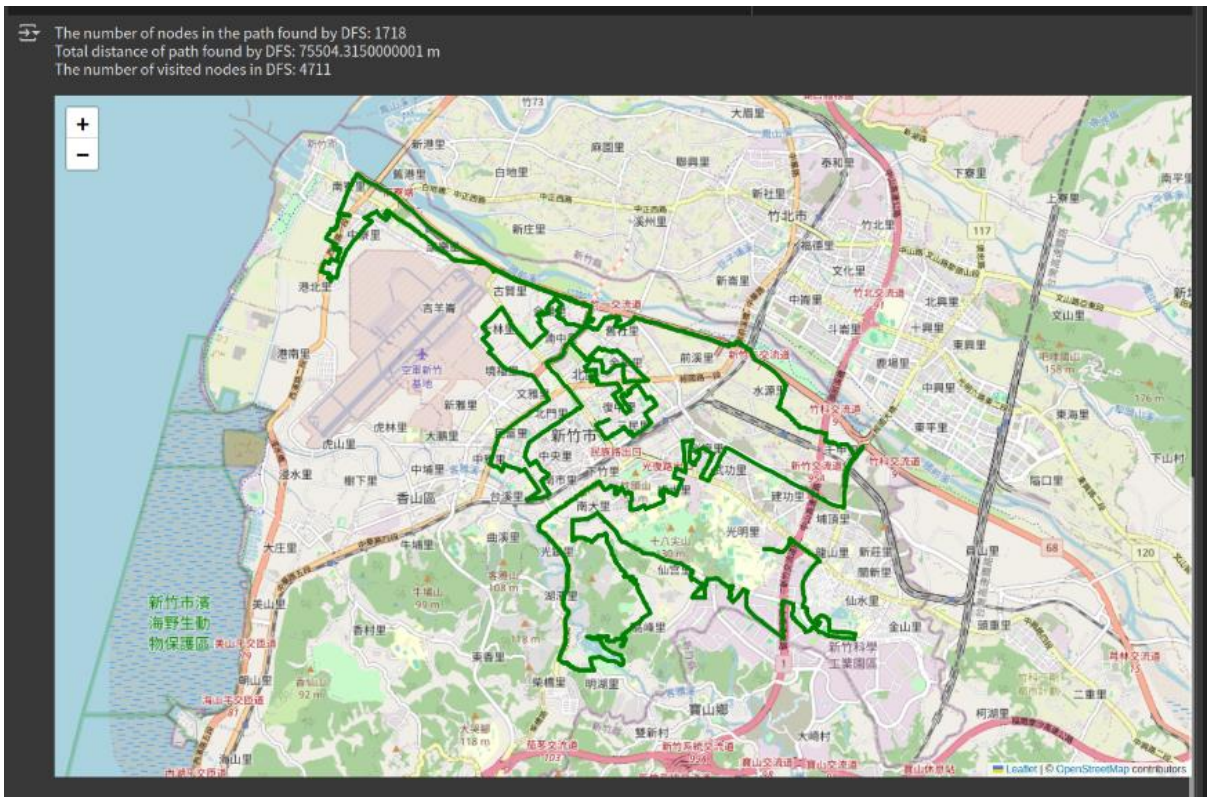
from National Yang Ming Chiao Tung University (ID: 2270143902)

to Big City Shopping Mall (ID: 1079387396)

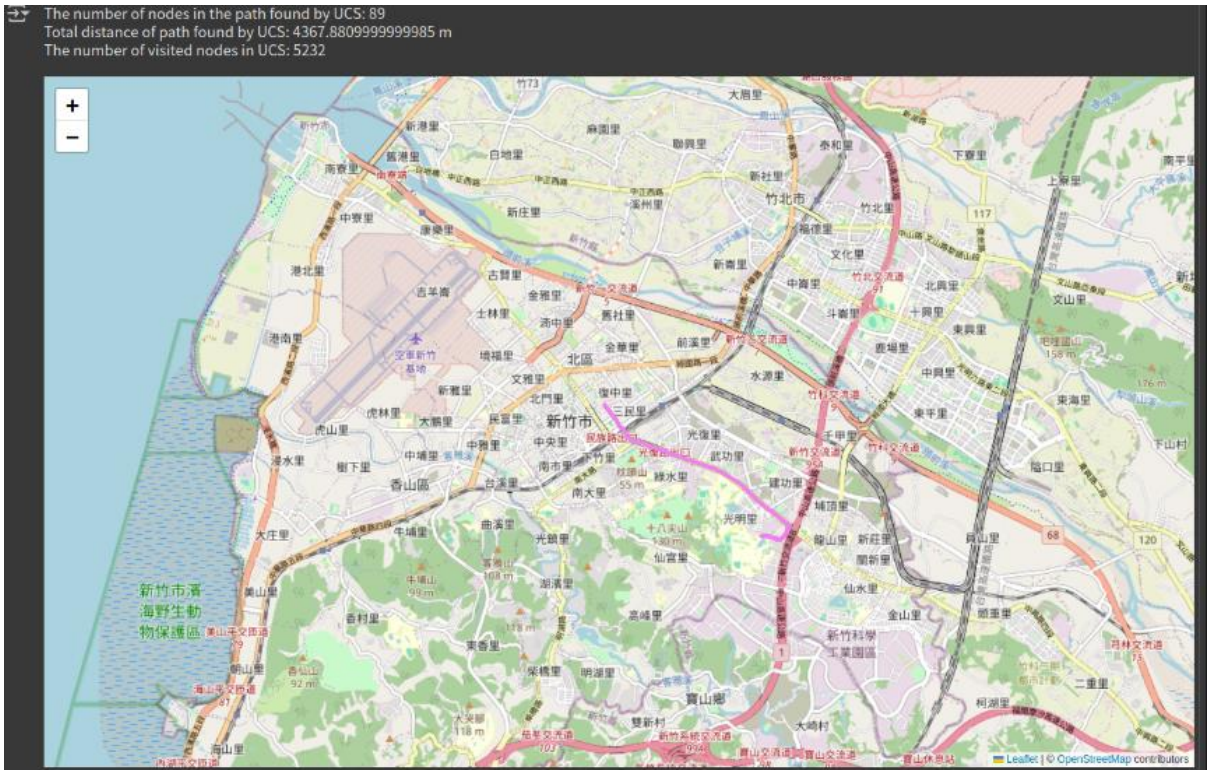
BFS:



DFS stack:

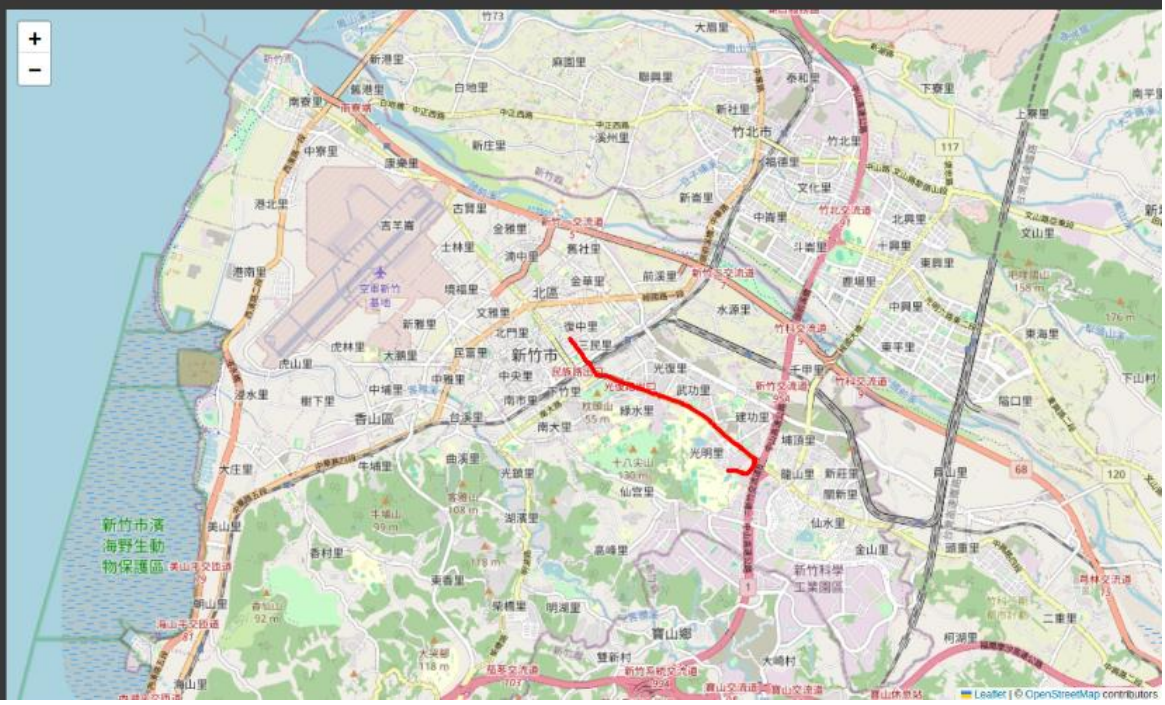

The number of nodes in the path found by DFS: 1718
Total distance of path found by DFS: 75504.3150000001 m
The number of visited nodes in DFS: 4711
新竹市
香山區
寶山鄉
新竹市濱海野生動物保護區
Leaflet | © OpenStreetMap contributors

UCS:


The number of nodes in the path found by UCS: 89
Total distance of path found by UCS: 4367.8809999999985 m
The number of visited nodes in UCS: 5232
新竹市
香山區
寶山鄉
新竹市濱海野生動物保護區
Leaflet | © OpenStreetMap contributors

A_star:

The number of nodes in the path found by A* search: 89
Total distance of path found by A* search: 4367.8809999999985 m
The number of visited nodes in A* search: 262

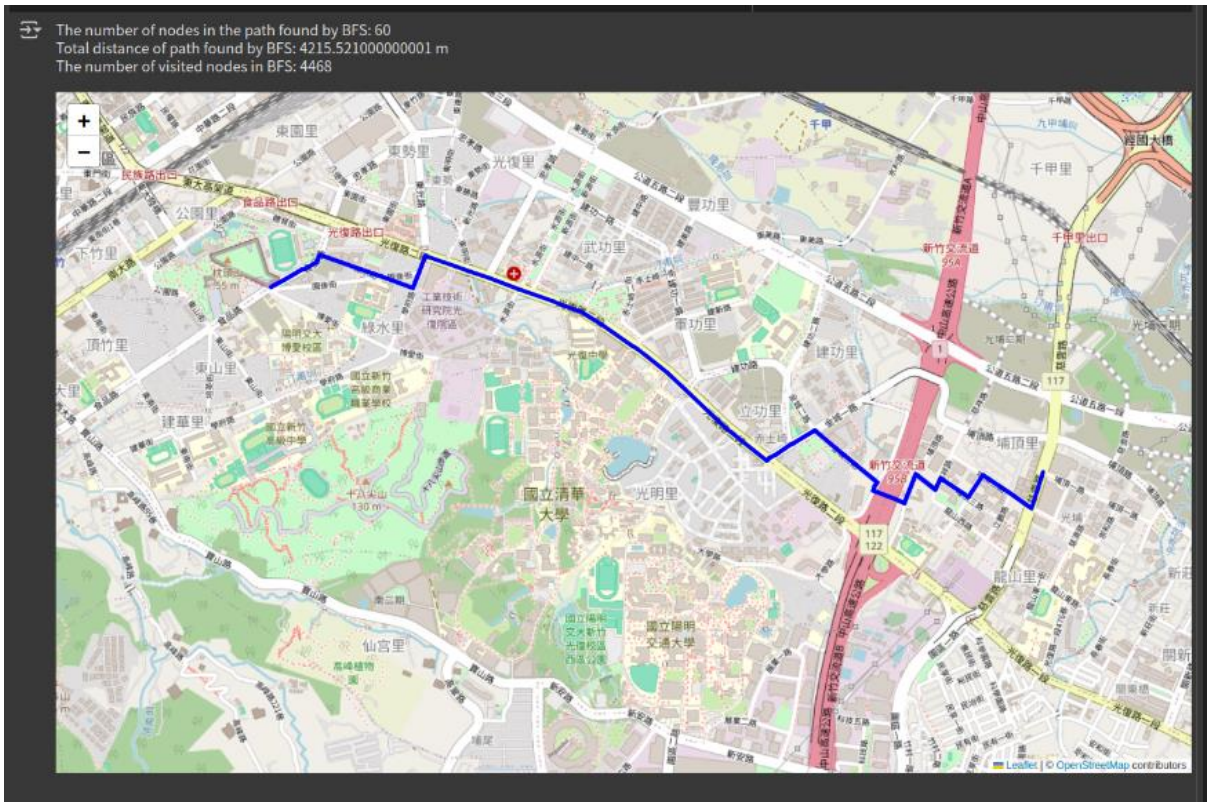


Test 2 ：

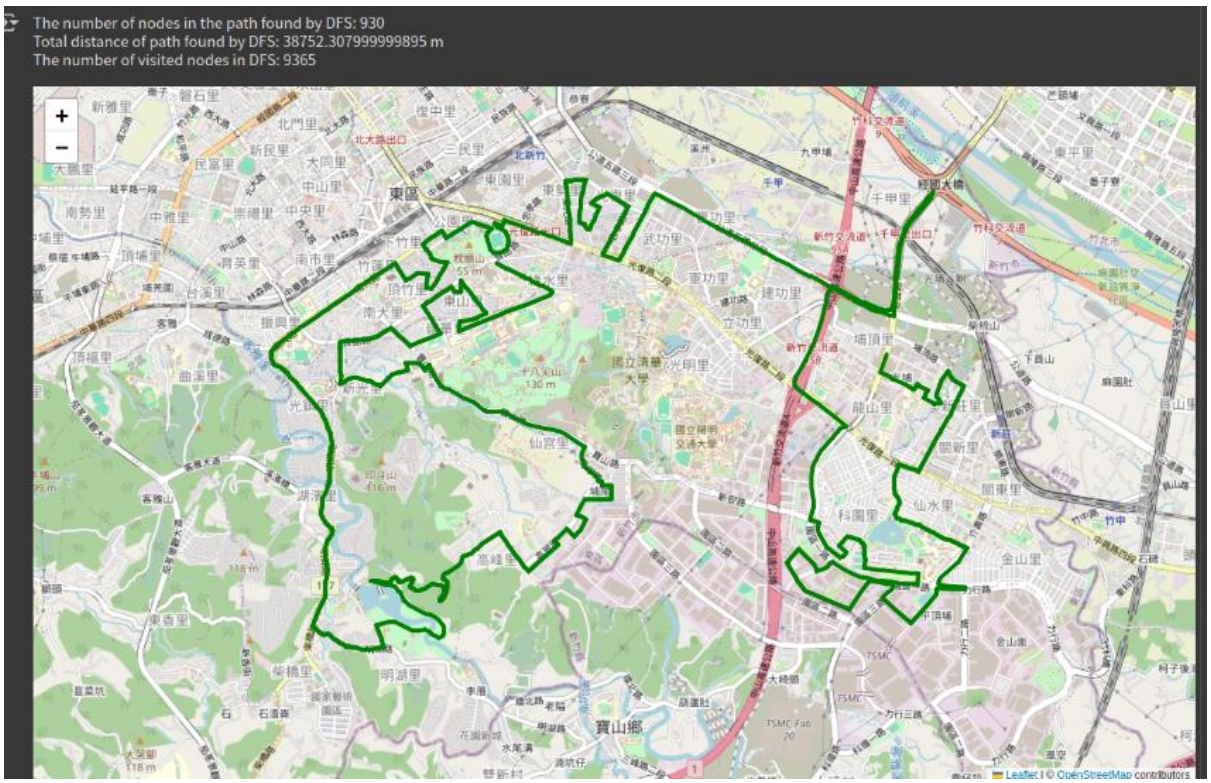
from Hsinchu Zoo (ID: 426882161)

to COSTCO Hsinchu Store (ID: 1737223506)

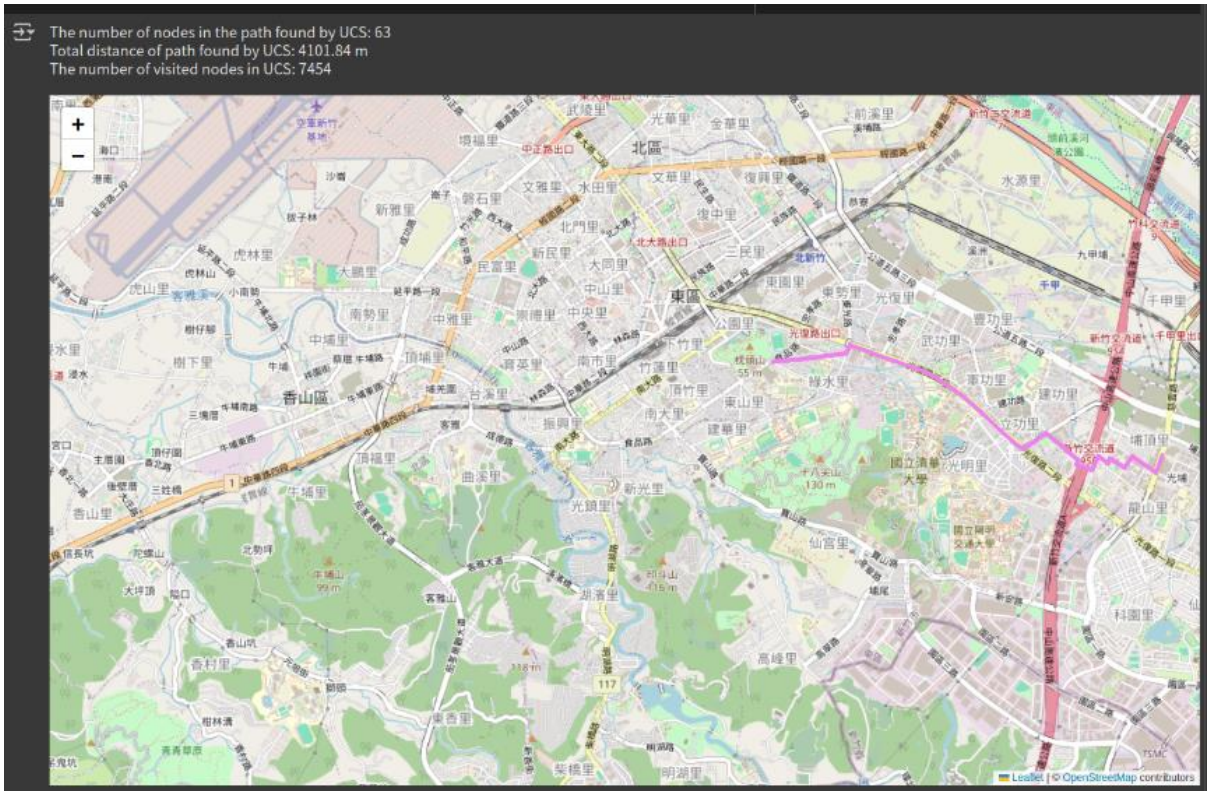
BFS:



DFS stack:

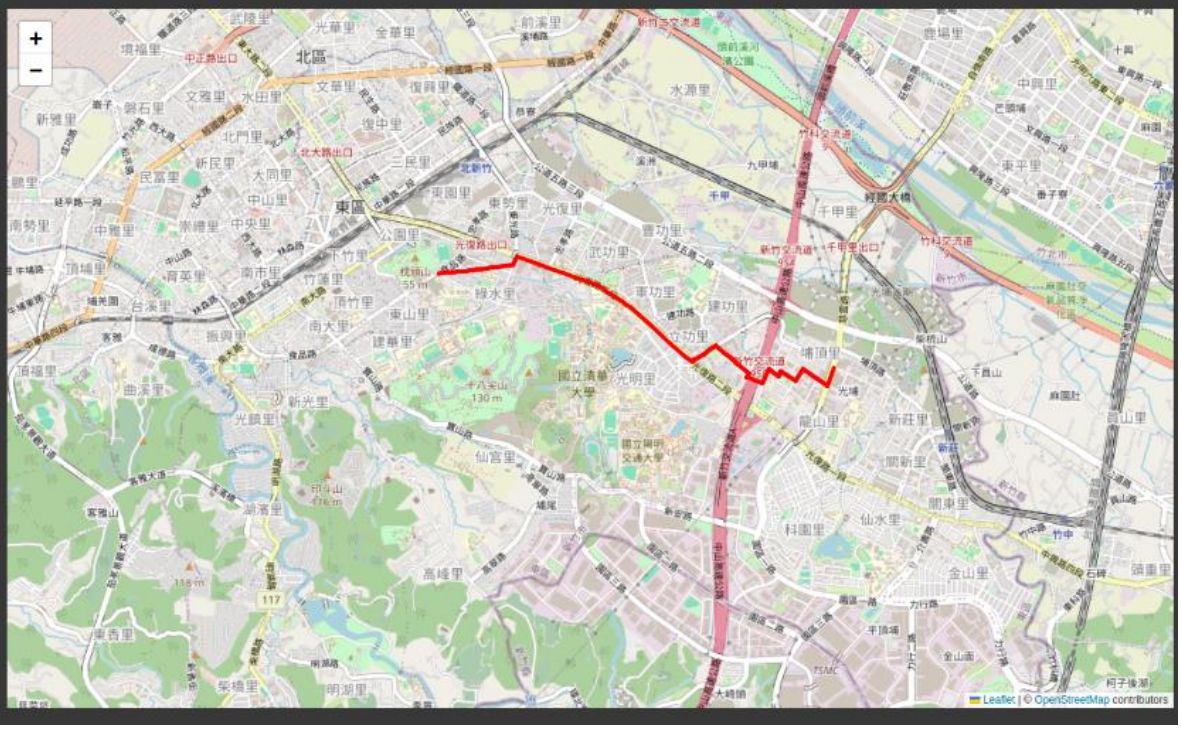

The number of nodes in the path found by DFS: 930
Total distance of path found by DFS: 38752.307999999895 m
The number of visited nodes in DFS: 9365
Leaflet | © OpenStreetMap contributors

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
Leaflet | © OpenStreetMap contributors

A_star:

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

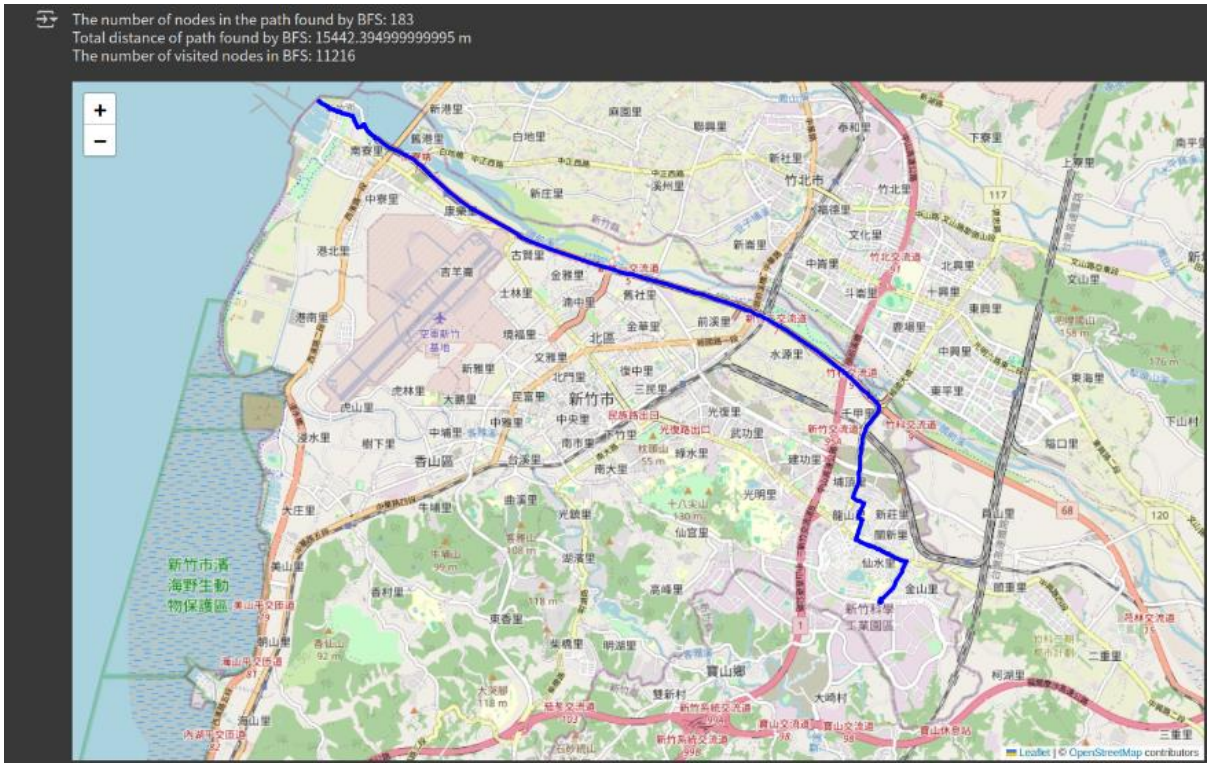


Test 3 :

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

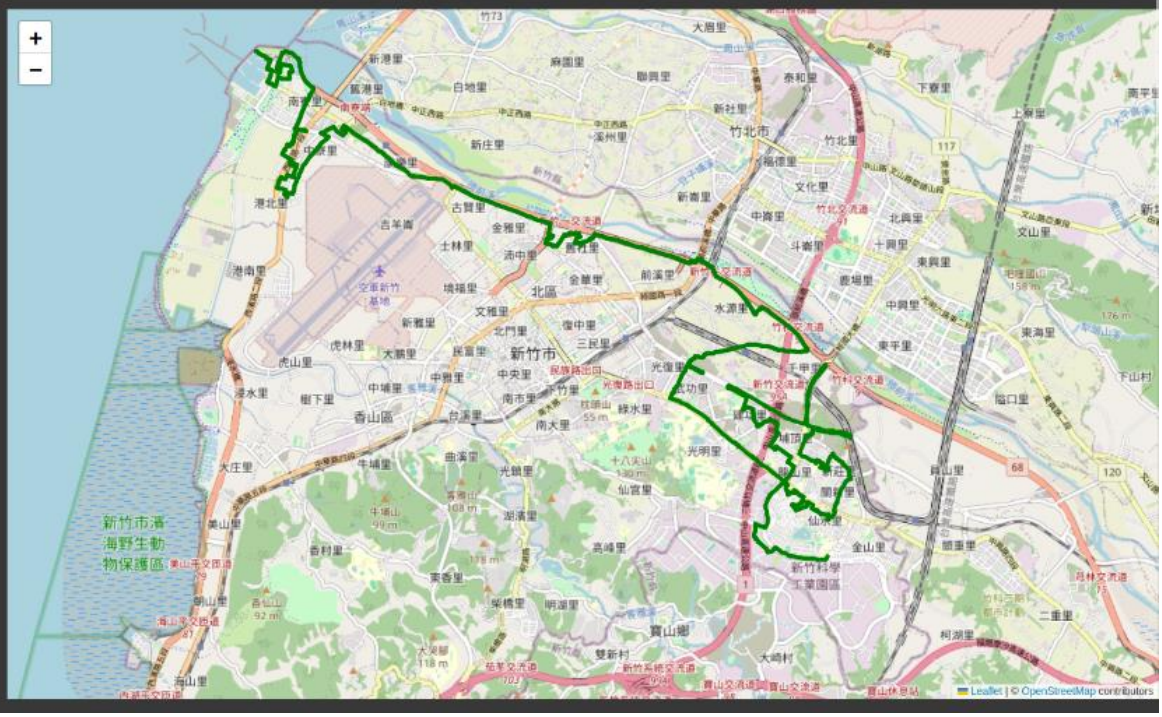
to Nanliao Fighing Port (ID: 8513026827)

BFS:

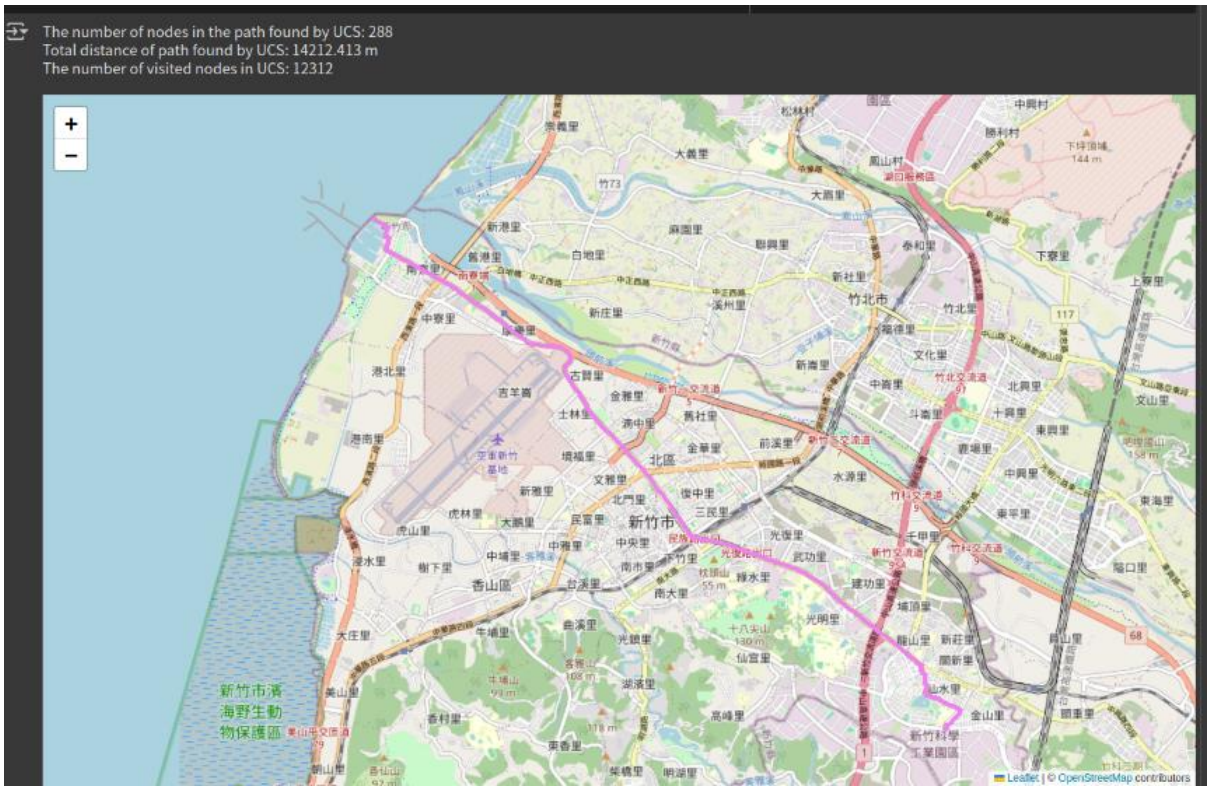


DFS stack:

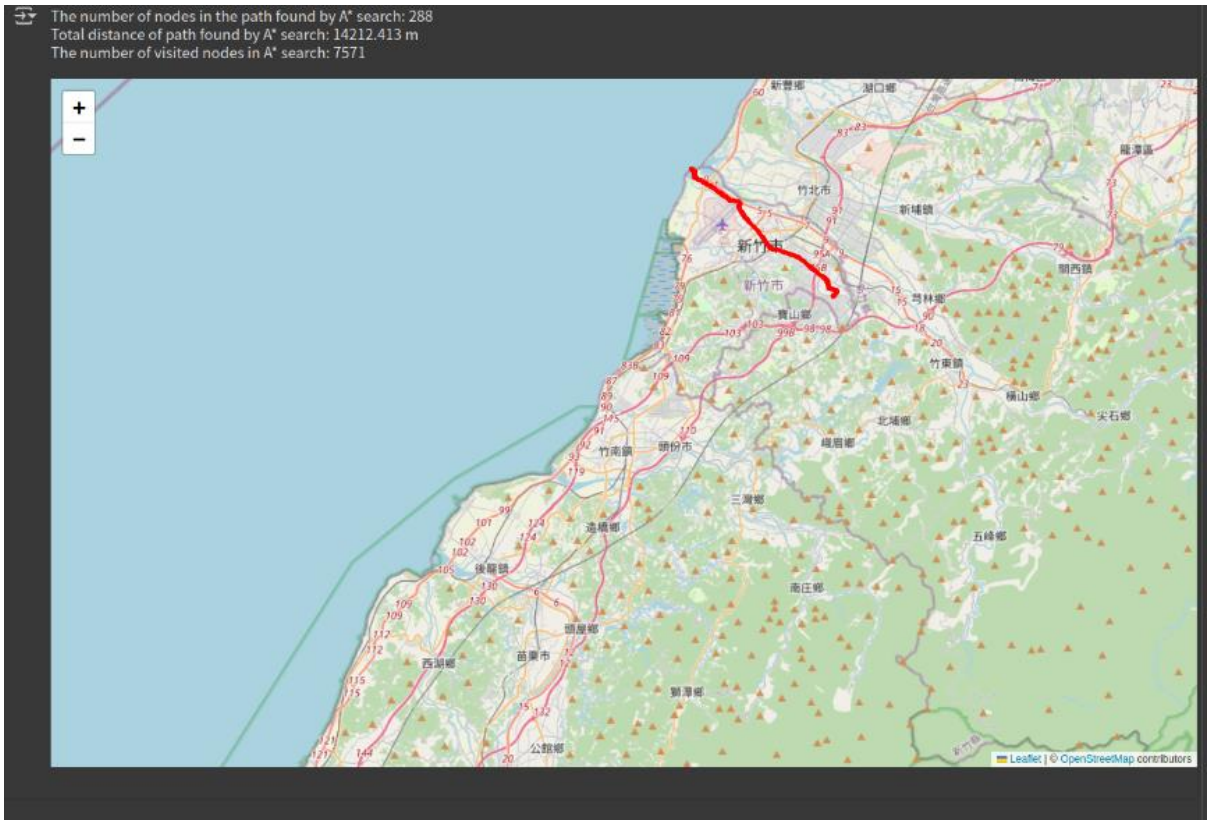
The number of nodes in the path found by DFS: 900
Total distance of path found by DFS: 39219.993000000024 m
The number of visited nodes in DFS: 2247



UCS:


The number of nodes in the path found by UCS: 288
Total distance of path found by UCS: 14212.413 m
The number of visited nodes in UCS: 12312
新竹市
新竹市濱海野生動物保護區
Leaflet | © OpenStreetMap contributors

A-star:


The number of nodes in the path found by A* search: 288
Total distance of path found by A* search: 14212.413 m
The number of visited nodes in A* search: 7571
新竹市
Leaflet | © OpenStreetMap contributors

Code:

BFS:

```python
import csv
edgeFile = 'edges.csv'


def bfs(start, end):
    # Begin your code (Part 1)
    """
    Load the csv file in and store it in nested map.
    edgefile: The csv data have the format of 'start, end, distance'.
    graph: The nested map that stores the graph. Store the data in the format of
    {start: {end: (distance)}}.

    For example, if the csv file is like:
    A_1, A_2, 1
    A_2, A_3, 2
    The graph should be like:
    {'A_1': {'A_2': 1}, 'A_2': {'A_3': 2}, 'A_3': {}}
    """
    graph = dict()
    with open(edgeFile, 'r') as f:
        for line in f:
            data = line.split(',') # split the data by ','
            if data[0] == 'start':
                continue
            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])

    """
    Init the retuen value.
    dist: A float value that stores the total distance of shortest the path.
    path: A list of string that stores the nodes of the shortest path from start to end.
    num_visited: An integer that stores the number of visited nodes.
    """
    dist = 0.0
    path = []
    num_visited = 0

    """
    Init the container for BFS.
    queue: A list of string that stores the nodes that need to be visited.
    visited: A set of string that stores the nodes that have been visited.
    parent: A map that stores the parent node of each node.
            We use this to trace back the shortest path from start node to end node
    nodes: A dictionary that stores the nodes that are connected to the current node.
    found: A boolean value that indicates whether the end node is found.
    """
    queue = []
    queue.append(str(start))

    visited = set() # Value in set is unique
    visited.add(str(start))

    parent = {str(start): None}# parent of start node is None
    nodes = dict()
    found = False

    """
    Implement the BFS algorithm.
    1. End the loop when the queue is empty or the end node is found.
    2. In the loop:
        a. Pop the first node in the queue.
        b. Increase the number of visited nodes by 1.
        c. If the current node is the end node, set found to True and break the loop.
        d. If the current node is not the end node, add the connected nodes to the queue.
    """
    while len(queue) > 0 and found == False:
        current = queue.pop(0)
        num_visited += 1
        nodes = graph[current]
        for node in nodes:
            if node not in visited:
                visited.add(node)
                parent[node] = current
                if node == str(end):
                    found = True
                    break
                queue.append(node)

    """
    Trace the path from the end node to the start node, to get the shortest path.
    To do this, we need to:
    1. End the loop if the start node is found or the parent is None.
    2. In the loop:
        a. Insert the current node to the path list.
        b. Add the distance between the current node and the parent node to the total distance.
        c. Update the current node to the parent node.
    """
    current = str(end)
    while current != str(start):
        path.append(int(current))
        if parent[current] == None:
            break
        dist = dist + float(graph[parent[current]][current])
        current = parent[current]

    path.append(str(start))
    path.reverse()

    return path, dist, num_visited

    # End your code (Part 1)


if __name__ == '__main__':
    path, dist, num_visited = bfs(2270143902, 1079387396)
    print(f'The number of path nodes: {len(path)}')
    print(f'Total distance of path: {dist}')
    print(f'The number of visited nodes: {num_visited}')
```

DFS:

```python
import csv
edgeFile = 'edges.csv'


def dfs(start, end):
    # Begin your code (Part 2)
    """
    Load the csv file in and store it in nested map.
    edgefile: The csv data that have the format of 'start' ,'end', 'distance'.
    graph: The nested map that store tje graph. Store the data in the format of
    {start: {end: (distance)}}.
    """
    graph = dict()
    with open(edgeFile, 'r') as f:
        for line in f:
            data = line.split(',')
            if data[0] == 'start':
                continue
            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])

    """
    Init the return value.
    dist: The float number that store the total distance of the shortest path.
    path: The list of the string that store the nodes of the shortest path from start to
end
    num_visited: The integer that store the number of visited nodes.
    """
    dist = 0.0
    path = []
    num_visited = 0

    """
    Init the container for DFS.
    stack: The list of the string that store the nodes that need to be visited.
    parent: The map that store the parent node of each node.
            We use this to trace back the shortest path from start node to end node.
    visited: The set of the string that store the nodes that have been visited.
    nodes: The dicitionary that store the nodes that are connected to the current node.
    found: The boolean value that indicate whetehr the end node is found or not.
    """
    stack = []
    stack.append(str(start))

    visited = set()
    visited.add(str(start))

    parent = {str(start) : None} # parent of the start node is None
    nodes = dict()
    found = False

    """
    Implemented the DFS alopgrithm.
    1.End the loop when stack is empty or the end node is foupath.insert(current)
    2. In the loop:
        a. Pop the current node from the stack.
        b. Check if the current node is visited or not.
        c. If the current node is the end node, set the found to True and break the loop.
        d. Increase the number of visited nodes by 1.
        e. If not, add the connected nodes into the stack.
    """
    while len(stack) > 0 and found == False:
        current = stack.pop()
        num_visited += 1
        nodes = graph[current]
        for node in nodes:
            if node not in visited:
                visited.add(node)
                parent[node] = current
                if node == str(end):
                    found = True
                    break
                stack.append(node)

    """
    Trace back the shortest path from the end node to the start node to achieve the
distace and path.
    To do this, we need to:
    1. End the loop when we find the start node or the parent of thwe current node is
None.
    2. In the loop:
        a. Add the distance vector between the current node and the parent node to the
total distance.
        b. Insert the current node to the path list.
        c. Update the current node to the parent node.
    """
    current = str(end)
    while current != str(start) and parent[current] != None:
        path.append(int(current))
        if parent[current] == None:
            break
        dist = dist + graph[parent[current]][current]
        current = parent[current]

    path.append(str(start))
    path.reverse()

    return path, dist, num_visited

    # End your code (Part 2)


if __name__ == '__main__':
    path, dist, num_visited = dfs(2270143902, 1079387396)
    print(f'The number of path nodes: {len(path)}')
    print(f'Total distance of path: {dist}')
    print(f'The number of visited nodes: {num_visited}')
```

UCS:

```python
import csv
import queue
edgeFile = 'edges.csv'


def ucs(start, end):
    # Begin your code (Part 3)
    """
    Load the csv file in a nd store it in nested map.
    edgeFile: The csv data that have the format of 'start', 'end', 'distance'.
    graph: The nested map that store the graph. Store the data in the format of
    {start: {end: (distance)}}.
    """
    graph = dict()
    with open(edgeFile, 'r') as f:
        for line in f:
            data = line.split(',')
            if data[0] == 'start':
                continue
            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])

    """
    Init the return value
    Path: The list that store the path from start node to end node
    Dist: The float that store the total distance of the path
    Num_visited: The int that store the number of visited nodes
    """
    path = []
    dis = 0.0
    num_visited = 0

    """
    Init the container for the UCS algorithm
    heap: The list that store the nodes that need to be visited. The format of the node is
            (distance, node)
    visited: The set that store the nodes that have been visited
    parent: The map that store the parent node of each node. We use this to trace back the
shortest path
    found: The boolean value that indicate whether the end node is found or not
    """
    # Priority queue can arrange the nodes in the order of distance.
    heap = queue.PriorityQueue()
    heap.put((0.0, str(start)))

    visited = set()
    visited.add(str(start))

    parent = {str(start): (None, 0.0)} # parent of the start node is None, and the
distance is 0.0
    found = False
    # End your code (Part 3)
    """
    Implemented the UCS algorithm
    1. End the loop when the heap is empty or the node is found.
    2. In the loop:
        a.  Pop the node with the smallest distance from the heap
        b. If the node is the end node or the queue is empty, end the loop
        c. Else add it to the visited set.
        d. Get the connected nodes of the current node
        e. For each node, calculate the distance from start node to the node.
        f. If the node is not visited or we found the shorter path, update the parent node
and add it to the heap.

    For example:
    A --1-- B --2-- C
    |       |       |
    4       5       1
    |       |       |
    C ----1---- D
    The reason why we need to update the parent node is that we need to trace back the
shortest path.
    If we just see through a -> c, there are two paths: a -> b -> c and a -> c. If we
found the shorter path
    ,we first find a->c, so we set the parent of c is a. Then we find a->b->c, we need to
update the parent of c
    to b. So we can trace back the shortest path.
    """
    while heap.empty() == 0 and found == False:
        current = heap.get()
        num_visited += 1
        if str(current[1]) == str(end):
            found= True
            break
        nodes = graph[current[1]]
        for node in nodes:
            # Current[0] is the distance from start node to current node,
            # graph[current[1]][node] is the distance from current node to the connected
node
            start_to_node = current[0] + graph[current[1]][node]
            if node not in visited or start_to_node < parent[node][1]: # parent[node][1]
is the distance already found to reach the current node
                visited.add(node)
                parent[node] = (current[1], start_to_node)
                heap.put((start_to_node, str(node)))

    """
    Traceback the shortest path
    1. End the loop when the parentnode is None or the current node is start node
    2. In the loop:
        a. Add the distance to the total distance
        b. Insert the current node to the path list
        c. Update the current node to the parent node
    """
    current = str(end)
    while current != str(start) and parent[current][0] != None:
        dis = dis + float(graph[parent[current][0]][current])
        path.append(int(current))
        current = parent[current][0]

    path.append(str(start))
    path.reverse()

    # The error of dis might
    return path, dis, num_visited


if __name__ == '__main__':
    path, dist, num_visited = ucs(2270143902, 1079387396)
    print(f'The number of path nodes: {len(path)}')
    print(f'Total distance of path: {dist}')
    print(f'The number of visited nodes: {num_visited}')
```

A_star:

```
import csv
import queue
edgeFile = 'edges.csv'
heuristicFile = 'heuristic_values.csv'


def astar(start, end):
    # Begin your code (Part 4)
    """
    Load the csv file in and store it in nested map.
    edgefile; Is a csv file that stores the data it the format of 'start', 'end',
'distance'
    graph: A nested map that store the graph. Store the data in the format of
    {start: {end: (distance)}}
    """
    graph = dict()
    with open(edgeFile, 'r') as f:
        for line in f:
            data = line.split(',')
            if data[0] == 'start':
                continue
            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])

    """
    Load the heuristic csv file and store it in a map.
    heuristic: Is a csv file that stores the data in the format of 'node', 'distance of
node to destination 1', 'distance of node to destination 2', 'distance of node to
destination 3'
    graph: A map that store the heuristic. Store the data in the format of:
    {node: Destination}
    """
    straight_dis = dict()
    with open(heuristicFile, 'r') as f:
        for line in f:
            data = line.split(',') # split the data by ','
            # data[3] = 8461.89\n
            data[3] = data[3].split('\n')[0] # data[3]=8461.89, data[3].split('\n')[1]='\n'
            if data[0] == 'node':
                # Identify what the destination is
                for index, element in enumerate(data):# Enumerate relate the data with the
index.
                    if element == str(end):# Chooses which is the destination
                        case = index
                        break
                continue
            straight_dis[data[0]] = float(data[case])

    """
    Init the return value
    dist: a float number that stoe the total distance of the shortest path
    path: a list of string that store the shortest path from start to end
    num_visited: an integer that store the number of visited nodes
    """
    dist = 0.0
    path = []
    num_visited = 0

    """
    Init the container for A* algorithm
    heap: a priority queue that store the nodes that need to be visisted. The format of of
the node is
            (weight, node)
    visited" a set of nodes that already visited
    parent: a map that store trhe partent of each nodes, we use this to trace back the
shortest path
            , the format of the node is {node:(parent, distance)}
    nodes: a dictionary that store the nodes that are connected to the current node
    found: A boolean that indicate whether the end is found or not.
    """
    heap = queue.PriorityQueue()
    heap.put((0+straight_dis[str(start)], str(start)))

    visited = set()
    visited.add(str(start))

    parent = dict()
    parent[str(start)] = (None, 0.0) # The parent of the start node is None, and the
straight_line distance is 0.0
    nodes = dict()
    found = False
    # End your code (Part 4)

    """
    Impleemented the A* algorithm
    weight: g() + h()
        g(): The distance from the start node to the current node
        h(): The straight line distance from the current node to the end node.
    1. End the loop when the heap is  empty or found is True
    2. In the loop:
        a. Pop the the first node in the heap(the smallest distance)
        b. If the node is the end node, end the loop
        c. Find the connected nodes of the current node
        d. For each node, calculate the weight of the node
        e. If the node is not visited or the weight is smaller than the previous one,
update the parent node and add it to the heap
            - Add to the visited set
            - Record the parent node and the distance in the dict()
            - Add the connected nodes to the heap
    """
    while heap.empty() == False and found == False:
        current = heap.get()
        num_visited += 1
        if current[1] == str(end):
            found = True
            break
        nodes = graph[current[1]]
        for node in nodes:
            # Distance from start to current node + distance from current node to the node
+  straight_line distance from the node to the end
            weight = (current[0] - straight_dis[current[1]]) + graph[current[1]][node] +
straight_dis[node]
            if node not in visited or weight < parent[node][1]:
                visited.add(node)
                parent[node] = (current[1], weight)
                heap.put((weight, node))

    """
    Trace back the shortest path from the end node to the start node
    1. End the loop when the start node is found or the parent is None.
    2. In the loop:
        a. Insert the node in the path,
        b. Add the distance between the current node and the parent node to the total
distance
        c.Replace the current node with the parent node
    """
    current = str(end)
    while current != str(start):
        if parent[current] == None:
            break
        path.append(int(current))
        dist = dist + float(graph[parent[current][0]][current])
        current = parent[current][0]

    path.append(str(start))
    path.reverse()
    return path, dist, num_visited

if __name__ == '__main__':
    path, dist, num_visited = astar(2270143902, 1079387396)
    print(f'The number of path nodes: {len(path)}')
    print(f'Total distance of path: {dist}')
    print(f'The number of visited nodes: {num_visited}')
```