Homework 2: Route Finding

Please keep the title of each section and delete examples. Note that please kee p the questions listed in Part III.

Part I. Implementation (6%):

- **Part 1: Breadth-first Search:**

```python
def bfs(start, end):
    # Begin your code (Part 1)
    # raise NotImplementedError("To be implemented")
    """
    Load the csv data and store in a nested map.
    :edgeFile: The csv data in the format of "start, end, distance, speed limit".
    :graph: A nested map, storing the data of edgeFile in the format of
            { start { end : ( distance, speed limit ) } }.
    """
    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()

            data[3] = data[3].split('\n')[0]
            data[3] = str(float(data[3])*1000/3600)  # convert km/hr to m/s

            graph[data[0]][data[1]] = (float(data[2]), float(data[3]))

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

    """
    Init the containers.
    :queue: a list of string, storing the nodes while implementing BFS.
    :visited: a list of string, storing the nodes that have visited.
    :parent: a dictionary that records the parent node of each node in BFS,
            will use it to trace back the shortest path from the start to the 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()
    visited.add(str(start))

    parent = {str(start): None}
    nodes = dict()
    found = False
```

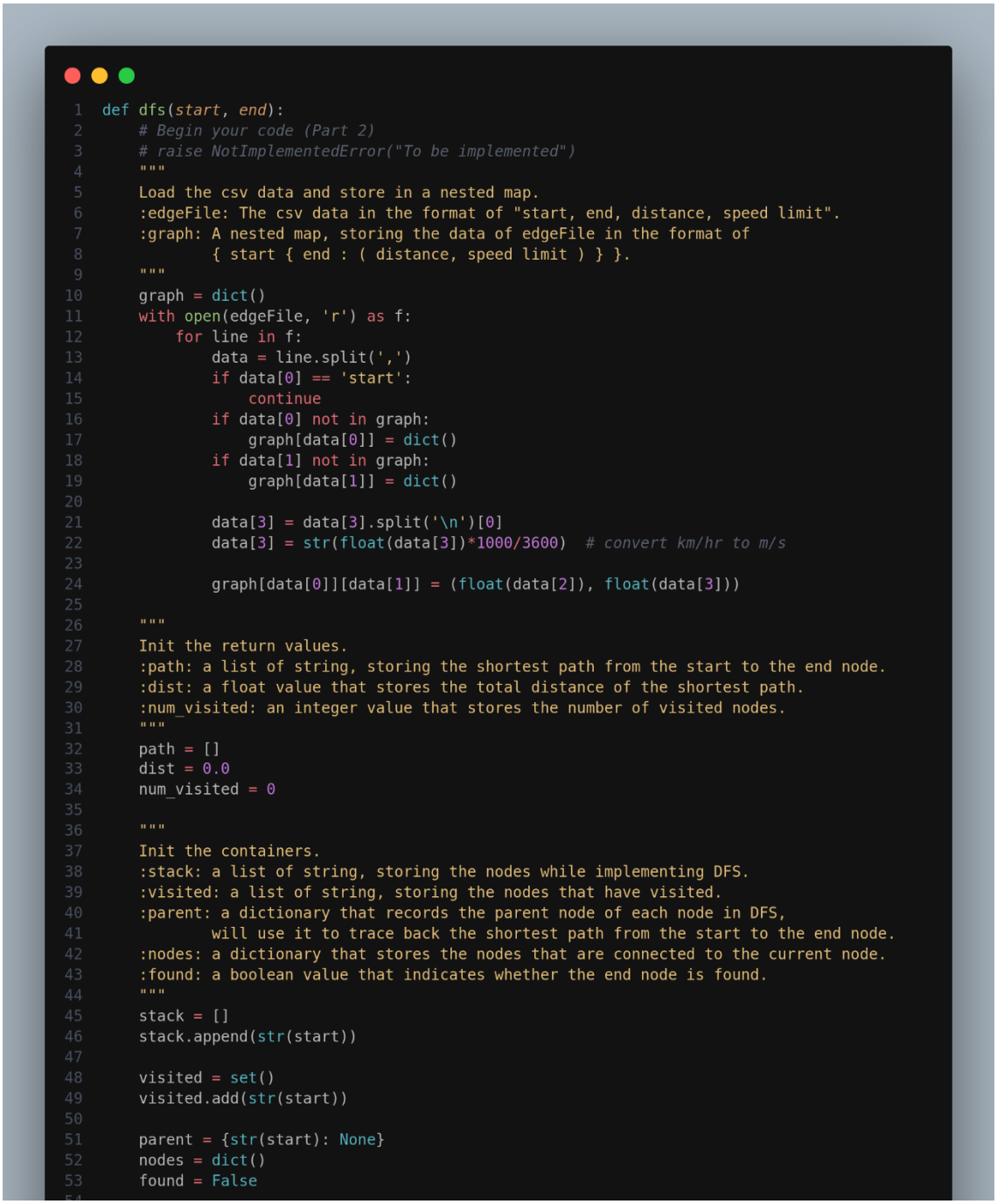
```
    """
    Implement BFS.
    - End the loop if the end node is found or the queue is empty.
    - In the loop:
        - Take out the first element in the queue.
        - Add the current node to the visited list.
        - Add the current node to the parent dictionary.
        - Check if it is the end node.
        - If not, add its neighbors to the queue.
    """
    while len(queue) > 0 and found == False:
        vertex = queue.pop(0)
        num_visited += 1
        nodes = graph[vertex]
        for node in nodes:
            if node not in visited:
                visited.add(node)
                parent[node] = vertex
                if node == str(end):
                    found = True
                    break
                queue.append(node)

    """
    Trace back from end to start.
    - End the loop if the start node is found or the parent dictionary is empty.
    - In the loop:
        - Add the current node to the path list.
        - Add the distance between the current node and its parent to the dist value.
        - Update the current node to its parent.
    """
    point = str(end)
    while point != str(start):
        path.append(int(point))
        if parent[point] == None:
            break
        dist = dist + float(graph[parent[point]][point][0])
        point = parent[point]

    path.append(start)
    path.reverse()

    return path, dist, num_visited
    # End your code (Part 1)
```

- **Part 2: Depth-first Search(stack)**



```python
def dfs(start, end):
    # Begin your code (Part 2)
    # raise NotImplementedError("To be implemented")
    """
    Load the csv data and store in a nested map.
    :edgeFile: The csv data in the format of "start, end, distance, speed limit".
    :graph: A nested map, storing the data of edgeFile in the format of
            { start { end : ( distance, speed limit ) } }.
    """
    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()

            data[3] = data[3].split('\n')[0]
            data[3] = str(float(data[3])*1000/3600)  # convert km/hr to m/s

            graph[data[0]][data[1]] = (float(data[2]), float(data[3]))

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

    """
    Init the containers.
    :stack: a list of string, storing the nodes while implementing DFS.
    :visited: a list of string, storing the nodes that have visited.
    :parent: a dictionary that records the parent node of each node in DFS,
            will use it to trace back the shortest path from the start to the 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.
    """
    stack = []
    stack.append(str(start))

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

    parent = {str(start): None}
    nodes = dict()
    found = False
```

```
    """
    Implement DFS.
    - End the loop if the stack is empty or the end node is found.
    - In the loop:
        - Take out the last element in the stack.
        - Add the current node to the visited list.
        - Add the current node to the parent list.
        - If the end node is found, break the loop.
        - If the current node has not been visited, add it to the stack.
    """
    while len(stack) > 0 and found == False:
        vertex = stack.pop()
        num_visited += 1
        nodes = graph[vertex]
        for node in nodes:
            if node not in visited:
                visited.add(node)
                parent[node] = vertex
                if node == str(end):
                    found = True
                    break
                stack.append(node)

    """
    Trace back from end to start.
    - End the loop if the start node is found.
    - In the loop:
        - Add the current node to the path list.
        - If the current node is the start node, break the loop.
        - Add the distance between the current node and its parent node to the dist value.
        - Update the current node to its parent node.
    """
    point = str(end)
    while point != str(start):
        path.append(int(point))
        if parent[point] == None:
            break
        dist = dist + float(graph[parent[point]][point][0])
        point = parent[point]

    path.append(start)
    path.reverse()

    return path, dist, num_visited
    # End your code (Part 2)
```

- **Part 3: Uniform Cost Search**

```python
def ucs(start, end):
    # Begin your code (Part 3)
    # raise NotImplementedError("To be implemented")
    """
    Load the csv data and store in a nested map.
    :edgeFile: The csv data in the format of "start, end, distance, speed limit".
    :graph: A nested map, storing the data of edgeFile in the format of
            { start { end : ( distance, speed limit ) } }.
    """
    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()

            data[3] = data[3].split('\n')[0]
            data[3] = str(float(data[3])*1000/3600)  # convert km/hr to m/s

            graph[data[0]][data[1]] = (float(data[2]), float(data[3]))

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

    """
    Init the containers.
    :que: a list of string, storing the nodes while implementing UCS.
    :visited: a list of string, storing the nodes that have visited.
    :parent: a dictionary that records the parent node of each node in UCS,
            will use it to trace back the shortest path from the start to the 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.
    """
    que = queue.PriorityQueue()
    que.put((0.0, str(start)))

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

    parent = {str(start): (None, 0.0)}
    nodes = dict()
    found = False
```

```
    """
    Implement UCS.
    - End the loop if the queue is empty or the end node is found.
    - In the loop:
        - Take out the node with the smallest distance from the queue.
        - If the node is the end node, end the loop.
        - Else add it to the visited set.
        - Get the nodes that are connected to the current node.
        - For each node, calculate the distance from the start node to the node.
        - If the node is not in the visited set or the distance is smaller than the distance
            recorded in the parent dictionary, update the parent dictionary and put the node
            into the queue.
    """
    while que.empty() == 0 and found == False:
        vertex = que.get()
        num_visited += 1
        if vertex[1] == str(end):
            found = True
            break
        nodes = graph[vertex[1]]
        for node in nodes:
            weight = vertex[0] + graph[vertex[1]][node][0]
            if node not in visited or weight < parent[node][1]:
                visited.add(node)
                parent[node] = (vertex[1], weight)  # (point, distance)
                que.put((weight, node))

    """
    Trace back from end to start.
    - End the loop if the start node is found.
    - In the loop:
        - Add the current node to the path.
        - If the current node is the start node, end the loop.
        - Add the distance from the current node to its parent node to the total distance.
        - Update the current node to its parent node.
    """
    point = str(end)
    while point != str(start):
        path.append(int(point))
        if parent[point][0] == None:
            break
        dist = dist + float(graph[parent[point][0]][point][0])
        point = parent[point][0]

    path.append(start)
    path.reverse()

    return path, dist, num_visited
    # End your code (Part 3)
```

- **Part 4: A* Search**

```python
def astar(start, end):
    # Begin your code (Part 4)
    # raise NotImplementedError("To be implemented")
    """
    Load the csv data and store in a nested map.
    :edgeFile: The csv data in the format of "start, end, distance, speed limit".
    :graph: A nested map, storing the data of edgeFile in the format of
            { start { end : ( distance, speed limit ) } }.
    """
    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()

            data[3] = data[3].split('\n')[0]
            data[3] = str(float(data[3])*1000/3600)  # convert km/hr to m/s

            graph[data[0]][data[1]] = (float(data[2]), float(data[3]))

    """
    Load the heuristic data and store in a map.
    :heuristicFile: The csv data in the format of "node, distance to ID1, ..., distance to ID3".
    :straight_dist: A map, storing the data of heuristicFile in the format of
                    { node : h }.
    """
    straight_dist = dict()
    with open(heuristicFile, 'r') as f:
        for line in f:
            data = line.split(',')
            data[3] = data[3].split('\n')[0]
            if data[0] == 'node':
                for i, element in enumerate(data):
                    if element == str(end):
                        case = i
                        break
                continue
            straight_dist[data[0]] = float(data[case])

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

    """
    Init the containers.
    :que: a priority queue of string, storing the nodes while implementing UCS.
    :visited: a list of string, storing the nodes that have visited.
    :parent: a dictionary that records the parent node of each node in UCS,
                in the format of { node : ( parent, distance ) }.
    :nodes: a dictionary that stores the neighbors of the current node.
    :found: a boolean value that indicates whether the end node is found.
    """
    que = queue.PriorityQueue()
    que.put((0 + straight_dist[str(start)], str(start)))

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

    parent = {str(start): (None, 0.0)}
    nodes = dict()
    found = False

```

```
    """
    Implement A* algorithm.
    :weight: g() + h()
        - g(): the distance which have walked
        - h(): straight line distance between that node and end node

    - End the loop when the queue is empty or the end node is found.
    - In the loop:
        - Get the node with the smallest distance from the queue.
        - If the node is the end node, end the loop.
        - Get the neighbors of the node.
        - For each neighbor, Calculate the weight from the start node to the neighbor.
        - If the neighbor is not visited or the weight is smaller than the previous one,
        - Add the neighbor to the visited set.
        - Record the parent node and the distance in the parent dictionary.
        - Add the neighbor to the queue.
    """
    while que.empty() == 0 and found == False:
        vertex = que.get()
        num_visited += 1
        if vertex[1] == str(end):
            found = True
            break
        nodes = graph[vertex[1]]
        for node in nodes:
            weight = vertex[0] + graph[vertex[1]][node][0] + \
                straight_dist[node] - straight_dist[vertex[1]]
            if node not in visited or weight < parent[node][1]:
                visited.add(node)
                parent[node] = (vertex[1], weight)  # (point, distance)
                que.put((weight, node))

    """
    Trace back from end to start.
    - End the loop when the start node is found.
    - In the loop:
        - Get the parent node of the current node.
        - If the parent node is not found, end the loop.
        - Calculate the distance from the parent node to the current node.
        - Add the current node to the path.
    """
    point = str(end)
    while point != str(start):
        path.append(int(point))
        if parent[point][0] == None:
            break
        dist = dist + float(graph[parent[point][0]][point][0])
        point = parent[point][0]

    path.append(start)
    path.reverse()

    return path, dist, num_visited
    # End your code (Part 4)
```

- **Part 6: Search with a different heuristic (time)**

```python
def astar_time(start, end):
    # Begin your code (Part 6)
    # raise NotImplementedError("To be implemented")
    """
    Load the csv data and store in a nested map.
    :edgeFile: The csv data in the format of "start, end, distance, speed limit".
    :graph: A nested map, storing the data of edgeFile in the format of
            { start { end : ( distance, speed limit ) } }.
    """
    graph = dict()
    max_speed = 0
    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()

            data[3] = data[3].split('\n')[0]
            data[3] = str(float(data[3])*1000/3600)  # convert km/hr to m/s

            max_speed = max(max_speed, float(data[3]))
            graph[data[0]][data[1]] = (float(data[2]), float(data[3]))

    """
    Load the heuristic data and store in a map.
    :heuristicFile: The csv data in the format of "node, distance to ID1, ..., distance to ID3".
    :straight_dist: A map, storing the data of heuristicFile in the format of
                    { node : h }.
    """
    straight_dist = dict()
    with open(heuristicFile, 'r') as f:
        for line in f:
            data = line.split(',')
            data[3] = data[3].split('\n')[0]
            if data[0] == 'node':
                for i, element in enumerate(data):
                    if element == str(end):
                        case = i
                        break
                continue
            straight_dist[data[0]] = float(data[case])

    """
    Init the return values.
    :path: a list of string, storing the shortest path from the start to the end node.
    :time: a float value that stores the total time of the shortest path.
    :num_visited: an integer that stores the number of nodes that have been visited.
    """
    path = []
    time = 0.0
    num_visited = 0

    """
    Init the containers.
    :que: a priority queue, storing the nodes to be visited.
    :visited: a set, storing the nodes that have been visited.
    :parent: a map, storing the parent of each node.
    :nodes: a map, storing the neighbors of each node.
    :found: a boolean, indicating whether the end node has been found.
    """
    que = queue.PriorityQueue()
    que.put((0 + straight_dist[str(start)] / max_speed, str(start)))

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

    parent = {str(start): (None, 0.0)}
    nodes = dict()
    found = False
```

```
    """
    Implement the A* algorithm.
    :weight: g() + h()
        - g() = the time cost from start to current node
        - h() = the time cost of the straight line distance from current node to end node
    - The steps is the same as the A* algorithm.
    """
    while que.empty() == 0 and found == False:
        vertex = que.get()
        num_visited += 1
        if vertex[1] == str(end):
            found = True
            break
        nodes = graph[vertex[1]]
        for node in nodes:
            weight = vertex[0] + graph[vertex[1]][node][0] / graph[vertex[1]][node][1] + \
                straight_dist[node] / max_speed - \
                straight_dist[vertex[1]] / max_speed
            if node not in visited or weight < parent[node][1]:
                visited.add(node)
                parent[node] = (vertex[1], weight)  # (point, time)
                que.put((weight, node))

    """
    Trace back from end to start.
    - End the loop when the start node is reached.
    - Append the node to the path.
    - Update the time.
    """
    point = str(end)
    while point != str(start):
        path.append(int(point))
        if parent[point][0] == None:
            break
        time = time + graph[parent[point][0]][point][0] / \
            graph[parent[point][0]][point][1]
        point = parent[point][0]

    path.append(start)
    path.reverse()

    return path, time, num_visited
    # End your code (Part 6)
```

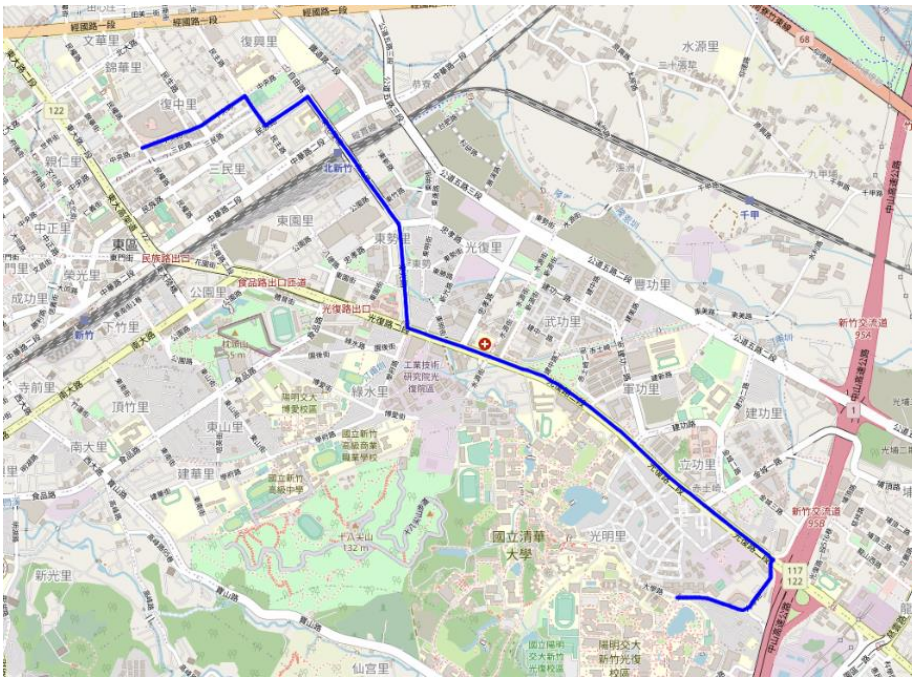
Part II. Results & Analysis (12%):

Please screenshot the results.

Test 1: from National Yang Ming Chiao Tung University (ID: 2270143902)to Big City Shopping Mall (ID: 1079387396)

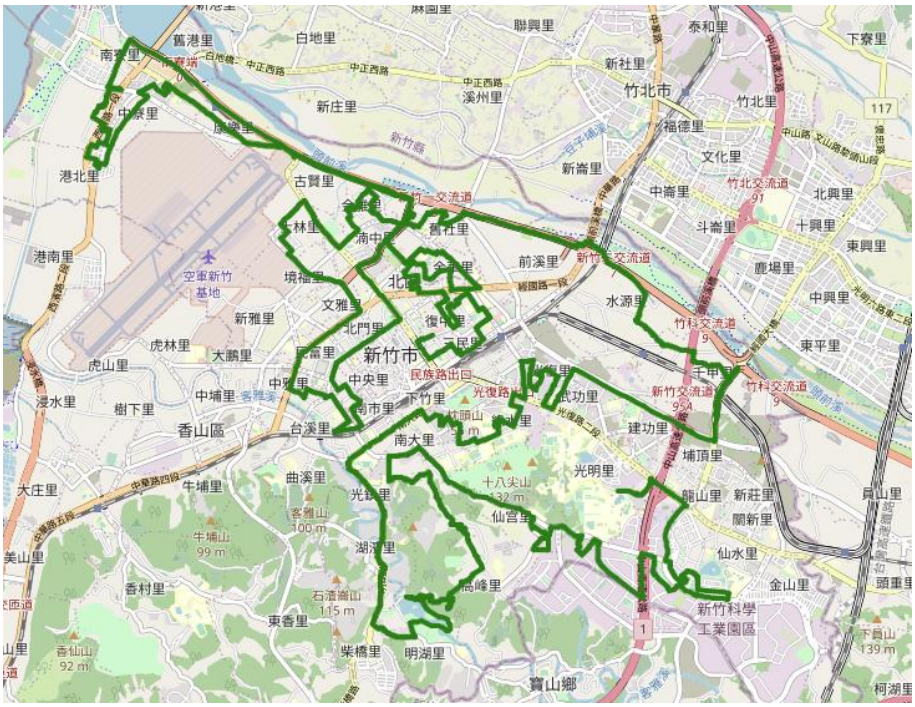
- BFS

```
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
```



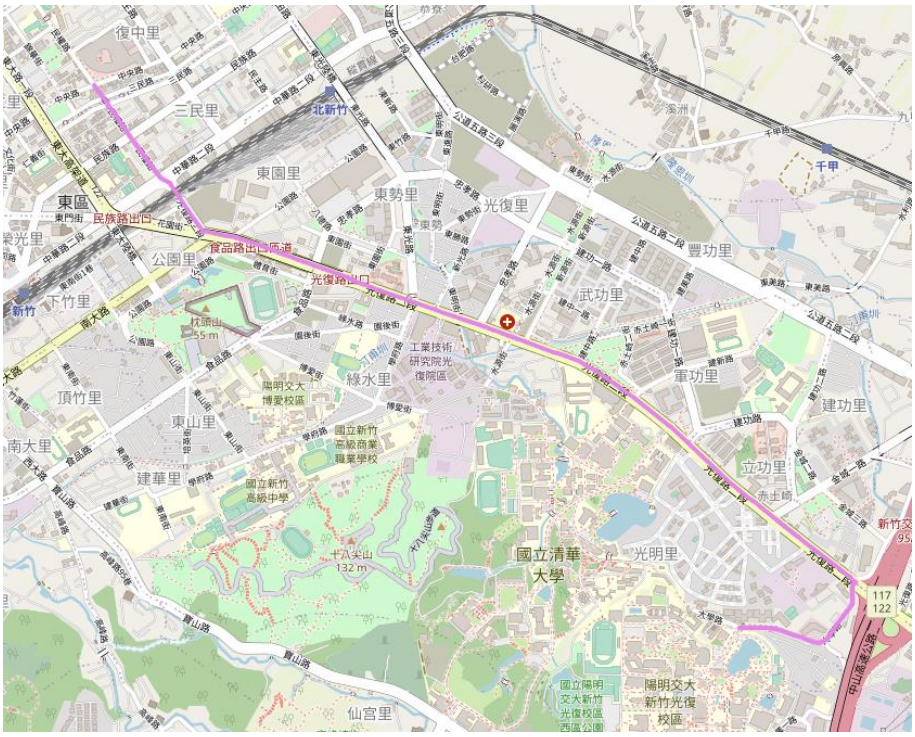
- **DFS(stack)**

```
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
```



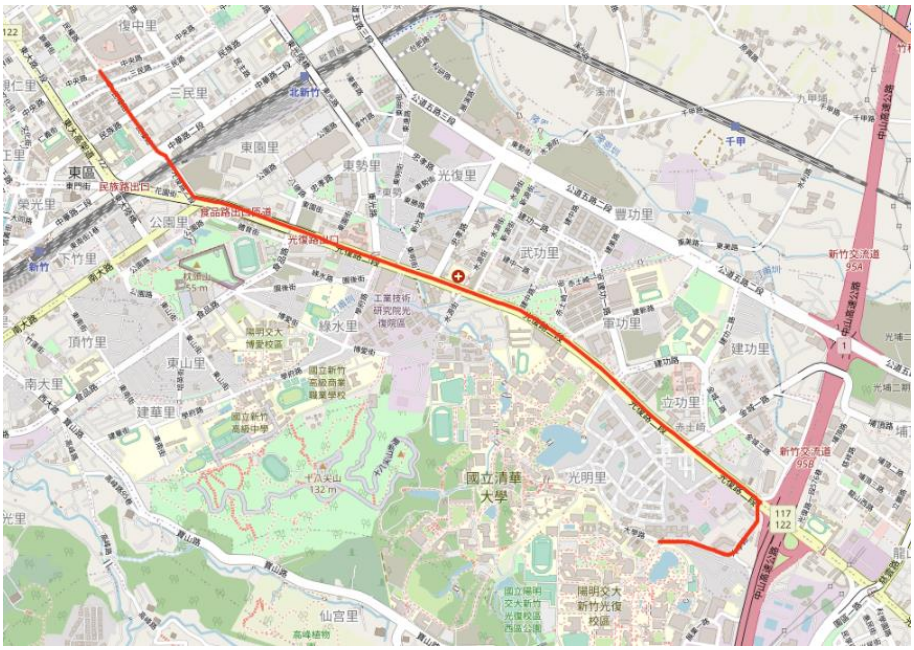
- **UCS**

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



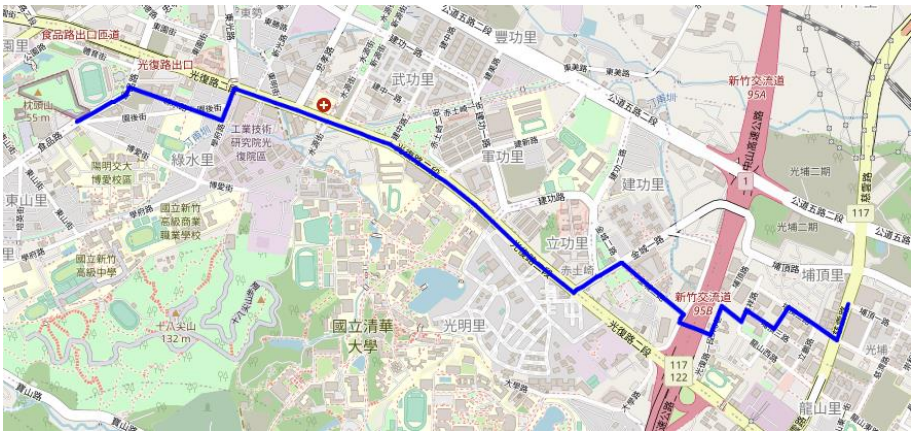
- **A* (time)**

```
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: 2016
```

Test 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: 4468
```

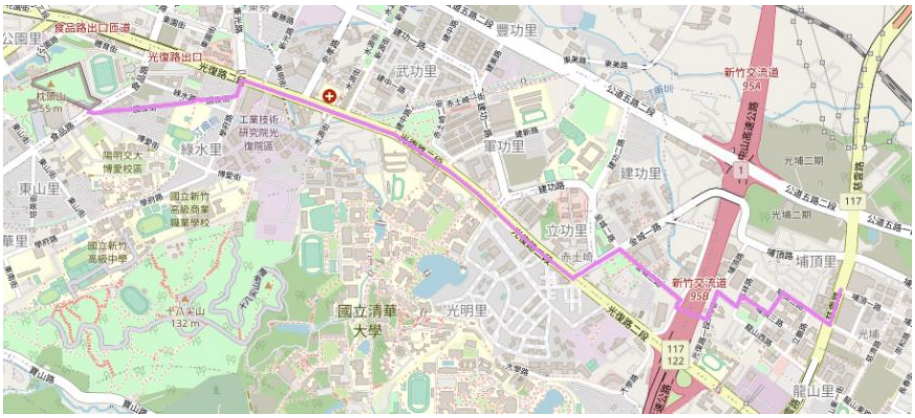


- DFS(stack)

```
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
```

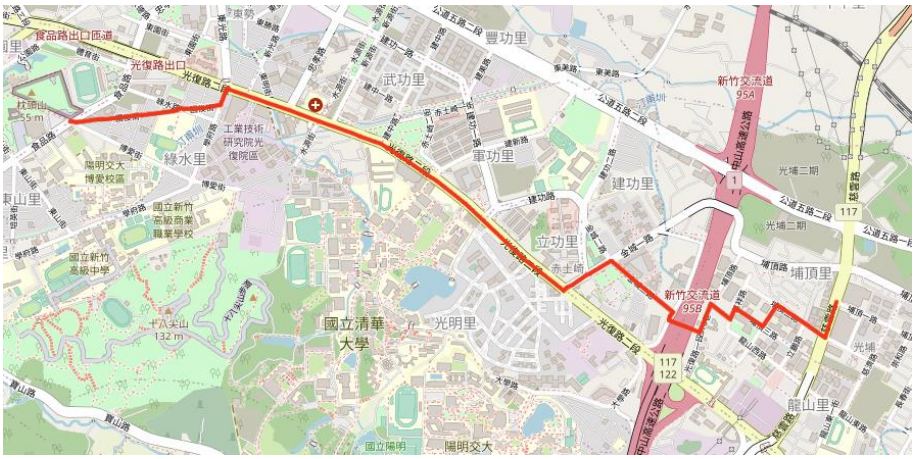
- **UCS**

```
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
```



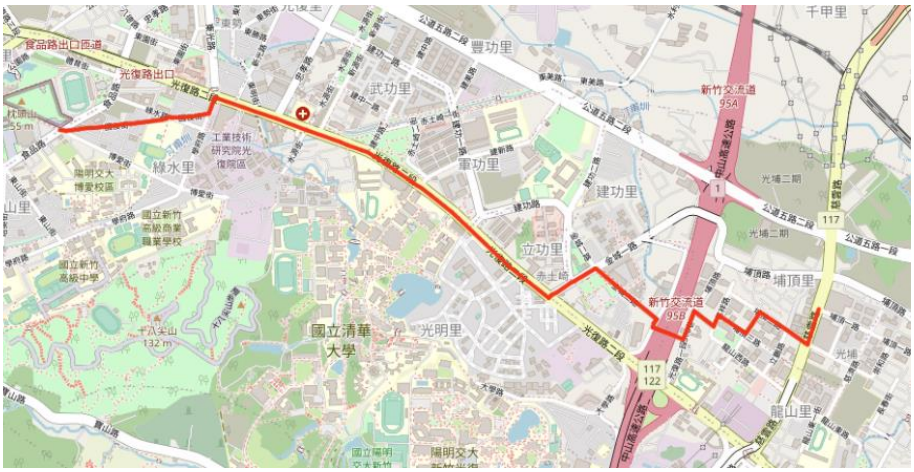
- **A***

```
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*(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: 2955
```



Test 3：from National Experimental High School At Hsinchu Science Park (ID: 1718165260) to Nanliao Fighing Port (ID: 8513026827)

- BFS

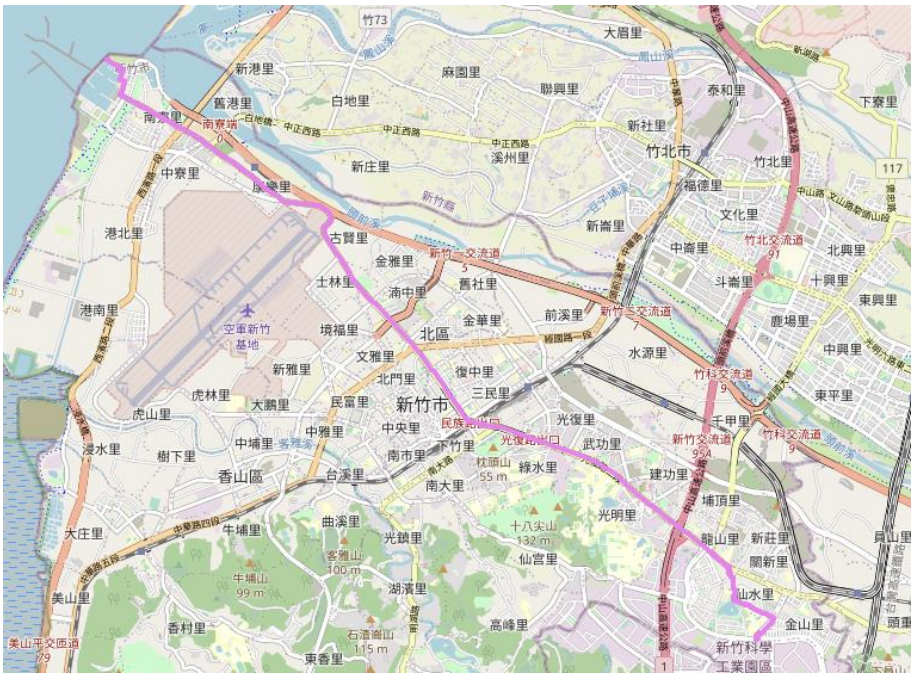
```
The number of nodes in the path found by BFS: 183
Total distance of path found by BFS: 15442.394999999995 m
The number of visited nodes in BFS: 11216
```

- **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: 14212.413 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: 14212.413 m
The number of visited nodes in A* search: 7571
```

- A*(time)

```
The number of nodes in the path found by A* search: 209
Total second of path found by A* search: 779.527922836848 s
The number of visited nodes in A* search: 8727
```

- Conclusion:

From the results, we found that UCS and A* are both two good ways in calculating the shortest path. If the target is far away from the start point, A* will be more effective than UCS, since UCS expands the search frontier uniformly in all directions, without any knowledge of the goal location, while A* using a heuristic function to estimate the distance between a node and the goal node.

For BFS, its performance is acceptable in comparison of DFS, though it is not an effect way, it is able to find the shortest way at least. For DFS, this algorithm does not take the distance between nodes into consideration, and it may miss the best solution while searching.

Part III. Question Answering (12%):

1. Please describe a problem you encountered and how you solved it.

   A:

   While loading the data from .csv file and storing it into a nest map, I made a mistake that use both the start node and end node as key to est ablish the dictionary.

```
graph[data[0]][data[1]] = (float(data[2]), float(data[3]))
graph[data[1]][data[0]] = (float(data[2]), float(data[3]))
```

   This would lead to the wrong result because it assumes that the sh ortest path between any two nodes is the same regardless of the directio n of the path. However, in some cases, the shortest path between two no des can be different depending on the direction of the path. I took few da ys to realize and fix it.

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 explai n the rationale.

   A:

   The traffic condition like the amount of car will also affect the time we take to arrive the destination. During the rush hour, we might stock in the traffic jam, and to choose the longer path other than the shortest pat h might be able to arrive the destination earlier. Therefore, the road cong estion is also essential for route finding.

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

   A:

   For localization, GPS is a widely used technology It uses a network of satellites to determine the location of a device on the Earth. It can be u sed for outdoor navigation but may not be accurate enough for indoor na vigation.

   For mapping, we can take the photos by street view car or satellite s, then convert the image data to vector data, with which we can use in ro ute finding.

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 upd ate ETA based on their mechanism. Please define a dynamic heuristic equ ation for ETA and explain the rationale of your design. Hint: You can consi der meal prep time, delivery priority, multiple orders, etc.

   A:

   Time_1 = meal prep time
   Time_2 = for the delivery partner to reach the restaurant and picks up the meal
   Time_3 = for the delivery partner to reach the delivery location
   Time_4 = buffer for unexpected delays or traffic

   ETA = Time_1 + Time_2 + Time_3 + Time_4