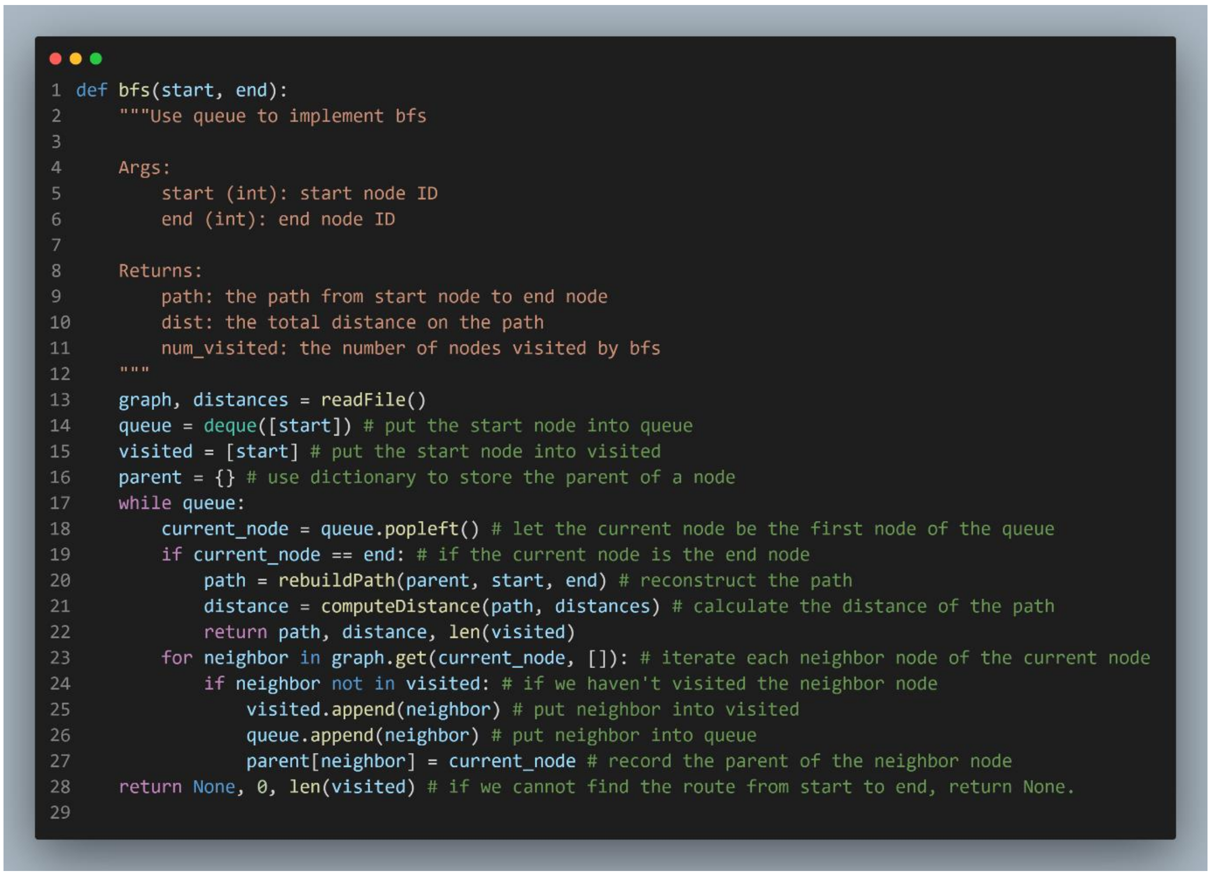
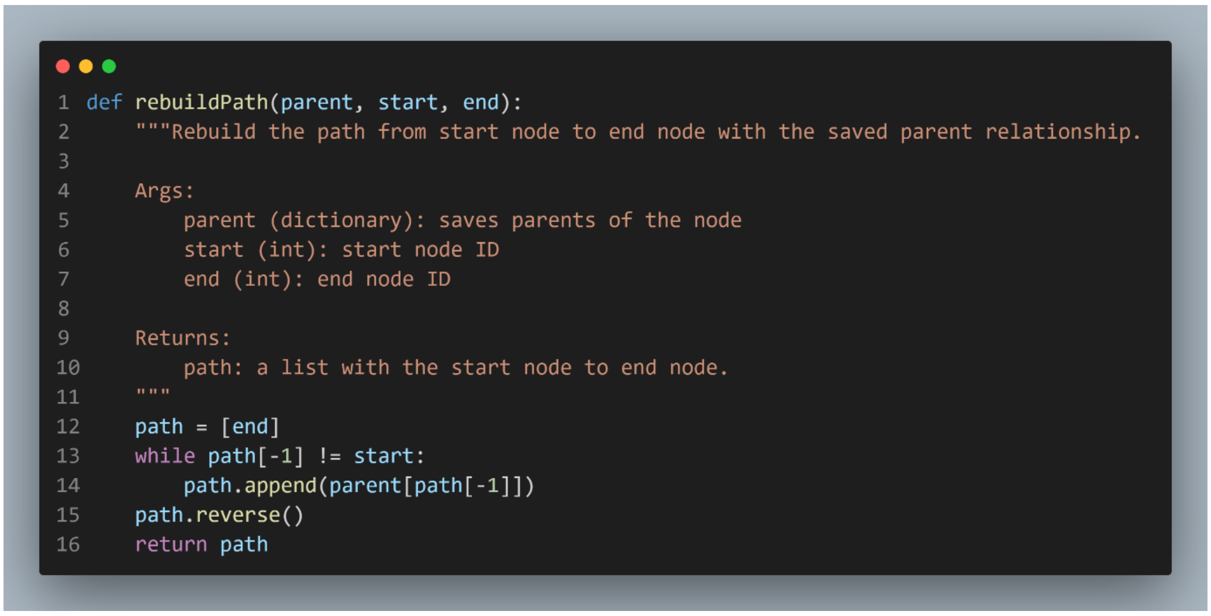
Homework 2: Route Finding

Part I. Implementation (6%):

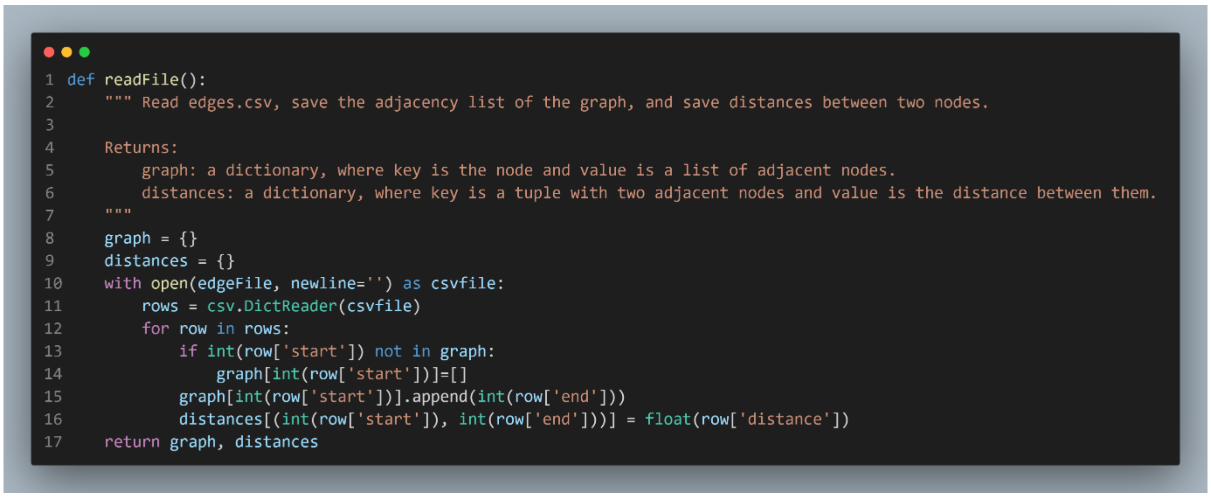
- **Part 1:**



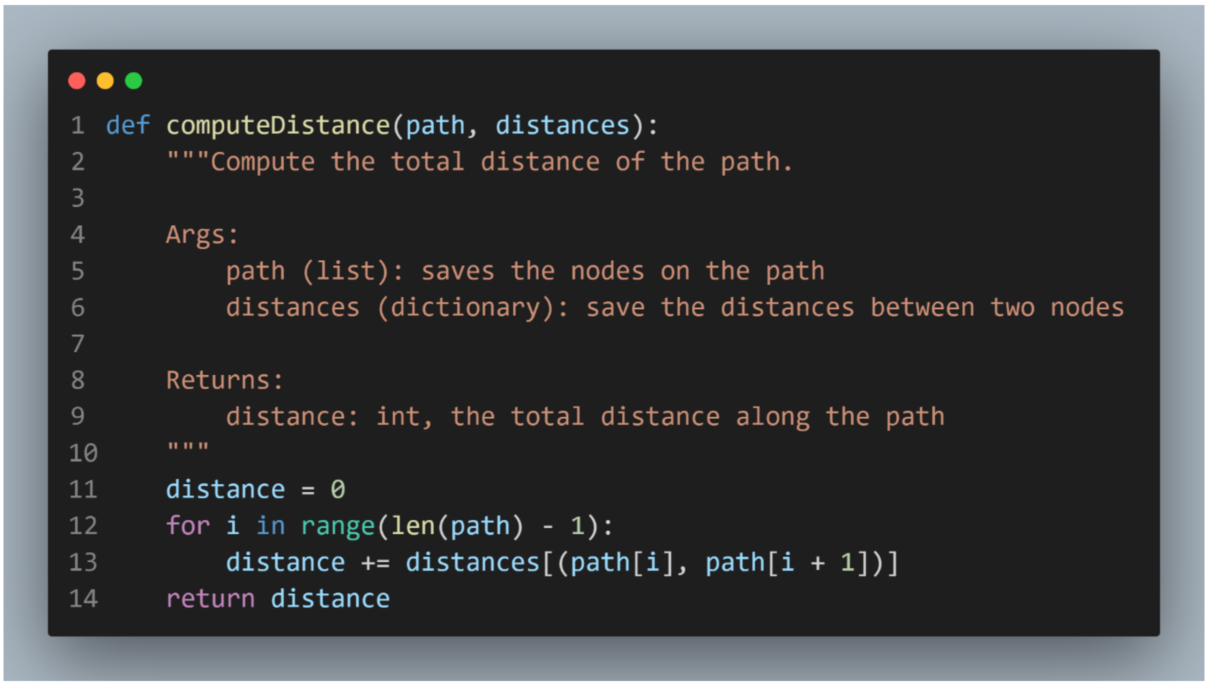
```python
def bfs(start, end):
    """Use queue to implement bfs

    Args:
        start (int): start node ID
        end (int): end node ID

    Returns:
        path: the path from start node to end node
        dist: the total distance on the path
        num_visited: the number of nodes visited by bfs
    """
    graph, distances = readFile()
    queue = deque([start]) # put the start node into queue
    visited = [start] # put the start node into visited
    parent = {} # use dictionary to store the parent of a node
    while queue:
        current_node = queue.popleft() # let the current node be the first node of the queue
        if current_node == end: # if the current node is the end node
            path = rebuildPath(parent, start, end) # reconstruct the path
            distance = computeDistance(path, distances) # calculate the distance of the path
            return path, distance, len(visited)
        for neighbor in graph.get(current_node, []): # iterate each neighbor node of the current node
            if neighbor not in visited: # if we haven't visited the neighbor node
                visited.append(neighbor) # put neighbor into visited
                queue.append(neighbor) # put neighbor into queue
                parent[neighbor] = current_node # record the parent of the neighbor node
    return None, 0, len(visited) # if we cannot find the route from start to end, return None.

```



```python
def rebuildPath(parent, start, end):
    """Rebuild the path from start node to end node with the saved parent relationship.

    Args:
        parent (dictionary): saves parents of the node
        start (int): start node ID
        end (int): end node ID

    Returns:
        path: a list with the start node to end node.
    """
    path = [end]
    while path[-1] != start:
        path.append(parent[path[-1]])
    path.reverse()
    return path
```

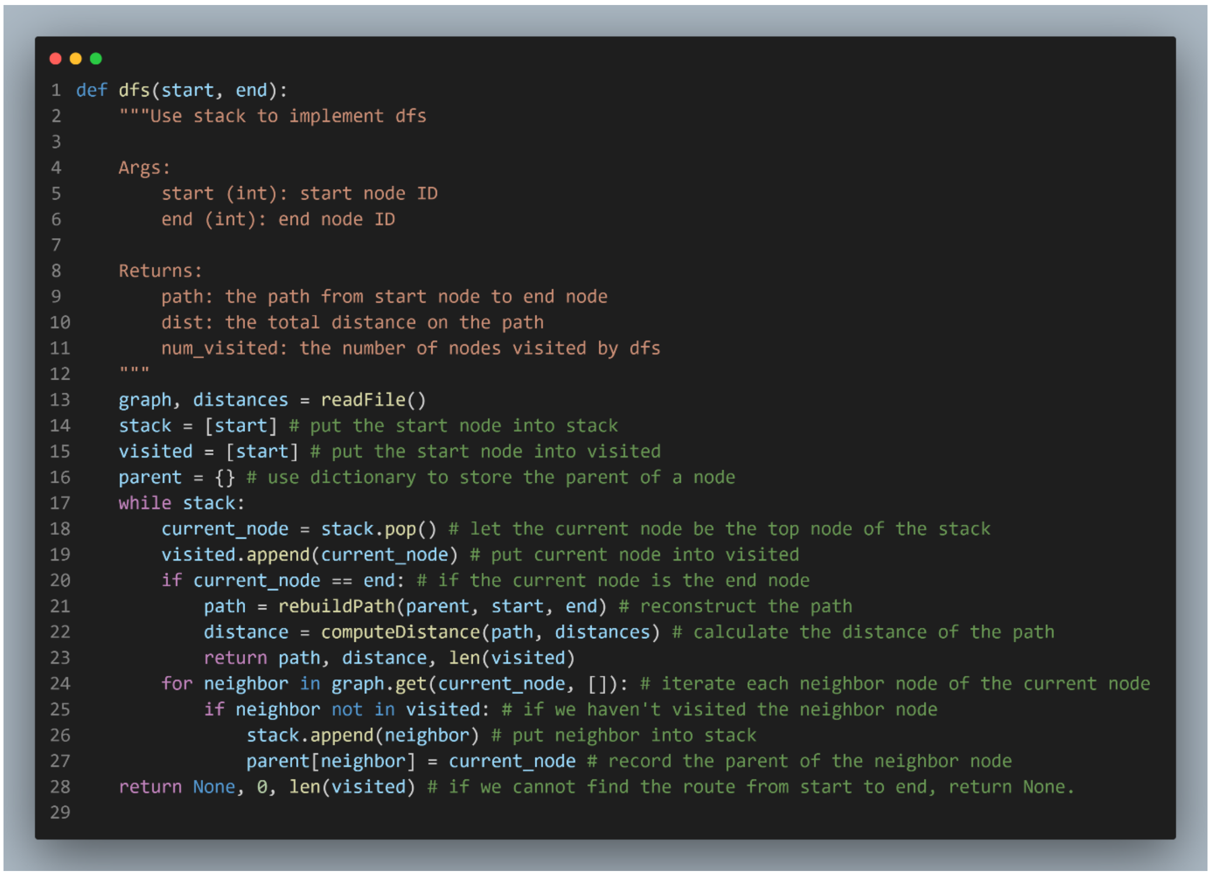


```python
def readFile():
    """ Read edges.csv, save the adjacency list of the graph, and save distances between two nodes.

    Returns:
        graph: a dictionary, where key is the node and value is a list of adjacent nodes.
        distances: a dictionary, where key is a tuple with two adjacent nodes and value is the distance between them.
    """
    graph = {}
    distances = {}
    with open(edgeFile, newline='') as csvfile:
        rows = csv.DictReader(csvfile)
        for row in rows:
            if int(row['start']) not in graph:
                graph[int(row['start'])]=[]
            graph[int(row['start'])].append(int(row['end']))
            distances[(int(row['start']), int(row['end']))] = float(row['distance'])
    return graph, distances
```



```python
def computeDistance(path, distances):
    """Compute the total distance of the path.

    Args:
        path (list): saves the nodes on the path
        distances (dictionary): save the distances between two nodes

    Returns:
        distance: int, the total distance along the path
    """
    distance = 0
    for i in range(len(path) - 1):
        distance += distances[(path[i], path[i + 1])]
    return distance
```

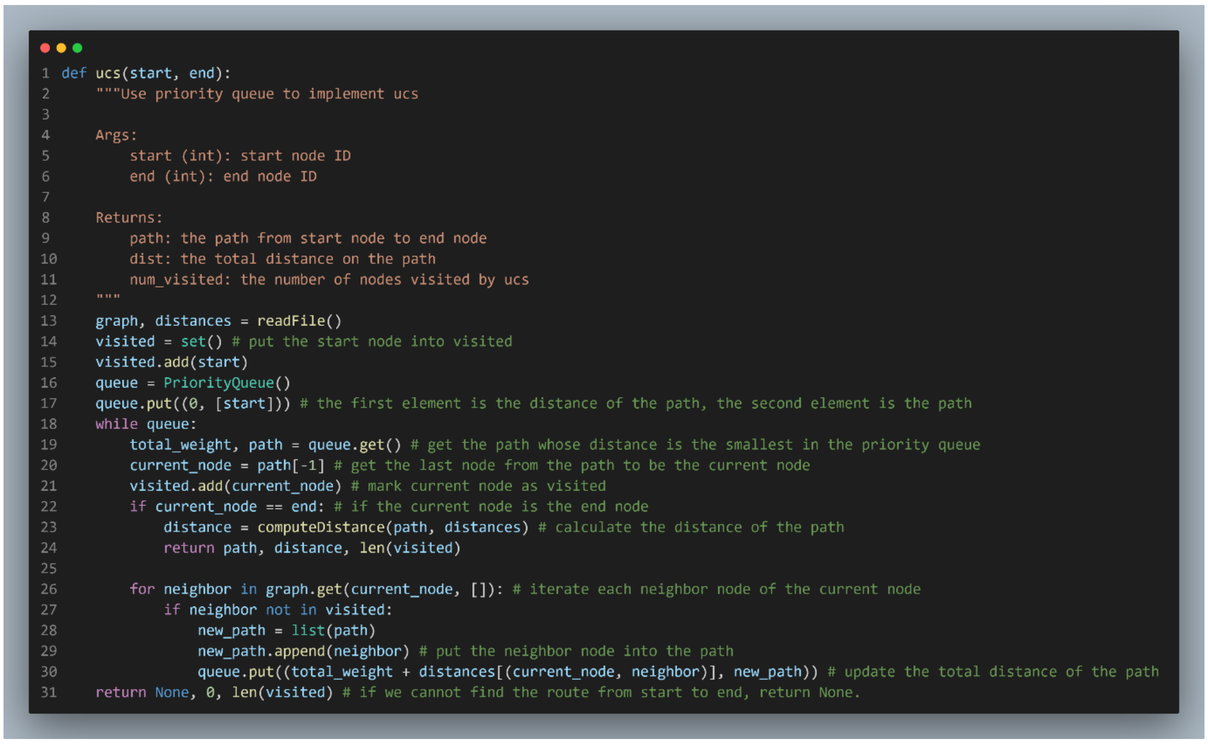
- **Part 2**



```python
def dfs(start, end):
    """Use stack to implement dfs

    Args:
        start (int): start node ID
        end (int): end node ID

    Returns:
        path: the path from start node to end node
        dist: the total distance on the path
        num_visited: the number of nodes visited by dfs
    """
    graph, distances = readFile()
    stack = [start] # put the start node into stack
    visited = [start] # put the start node into visited
    parent = {} # use dictionary to store the parent of a node
    while stack:
        current_node = stack.pop() # let the current node be the top node of the stack
        visited.append(current_node) # put current node into visited
        if current_node == end: # if the current node is the end node
            path = rebuildPath(parent, start, end) # reconstruct the path
            distance = computeDistance(path, distances) # calculate the distance of the path
            return path, distance, len(visited)
        for neighbor in graph.get(current_node, []): # iterate each neighbor node of the current node
            if neighbor not in visited: # if we haven't visited the neighbor node
                stack.append(neighbor) # put neighbor into stack
                parent[neighbor] = current_node # record the parent of the neighbor node
    return None, 0, len(visited) # if we cannot find the route from start to end, return None.

```

The remaining function is the same as the bfs.py, so I wouldn't paste it again.

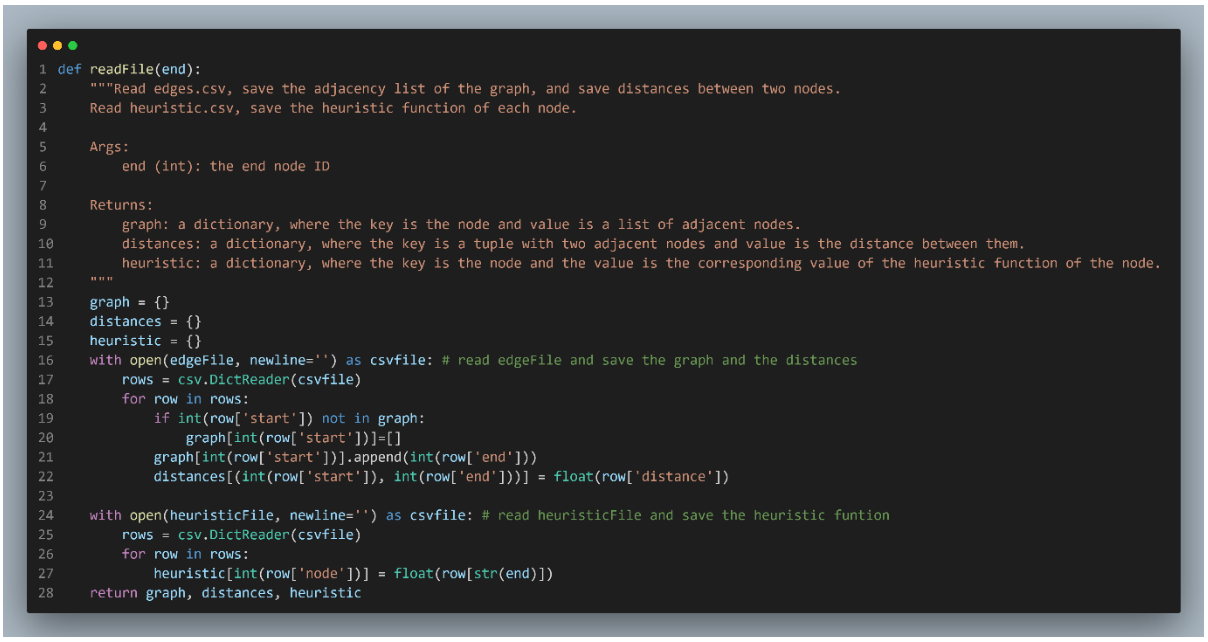
- ## Part 3



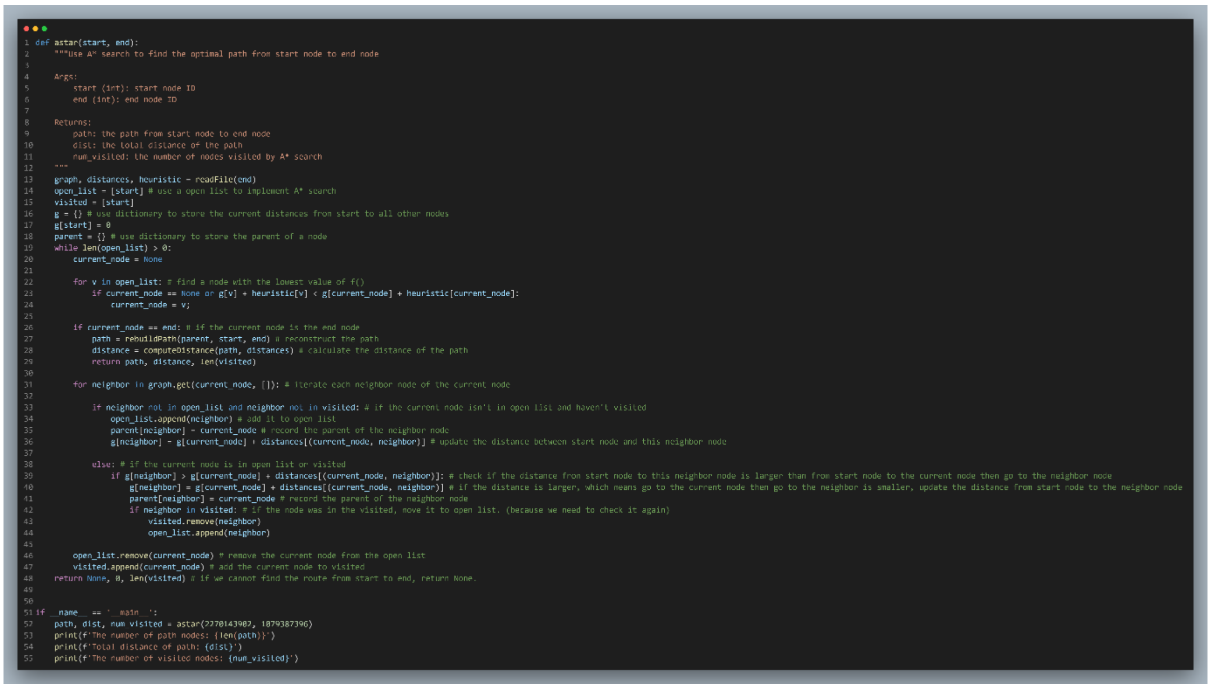
```python
def ucs(start, end):
    """Use priority queue to implement ucs

    Args:
        start (int): start node ID
        end (int): end node ID

    Returns:
        path: the path from start node to end node
        dist: the total distance on the path
        num_visited: the number of nodes visited by ucs
    """
    graph, distances = readFile()
    visited = set() # put the start node into visited
    visited.add(start)
    queue = PriorityQueue()
    queue.put((0, [start])) # the first element is the distance of the path, the second element is the path
    while queue:
        total_weight, path = queue.get() # get the path whose distance is the smallest in the priority queue
        current_node = path[-1] # get the last node from the path to be the current node
        visited.add(current_node) # mark current node as visited
        if current_node == end: # if the current node is the end node
            distance = computeDistance(path, distances) # calculate the distance of the path
            return path, distance, len(visited)

        for neighbor in graph.get(current_node, []): # iterate each neighbor node of the current node
            if neighbor not in visited:
                new_path = list(path)
                new_path.append(neighbor) # put the neighbor node into the path
                queue.put((total_weight + distances[(current_node, neighbor)], new_path)) # update the total distance of the path
    return None, 0, len(visited) # if we cannot find the route from start to end, return None.
```

The readFile function and computeDistance function are the same as the previous one, so I wouldn’ t paste it again.

- **Part 4**



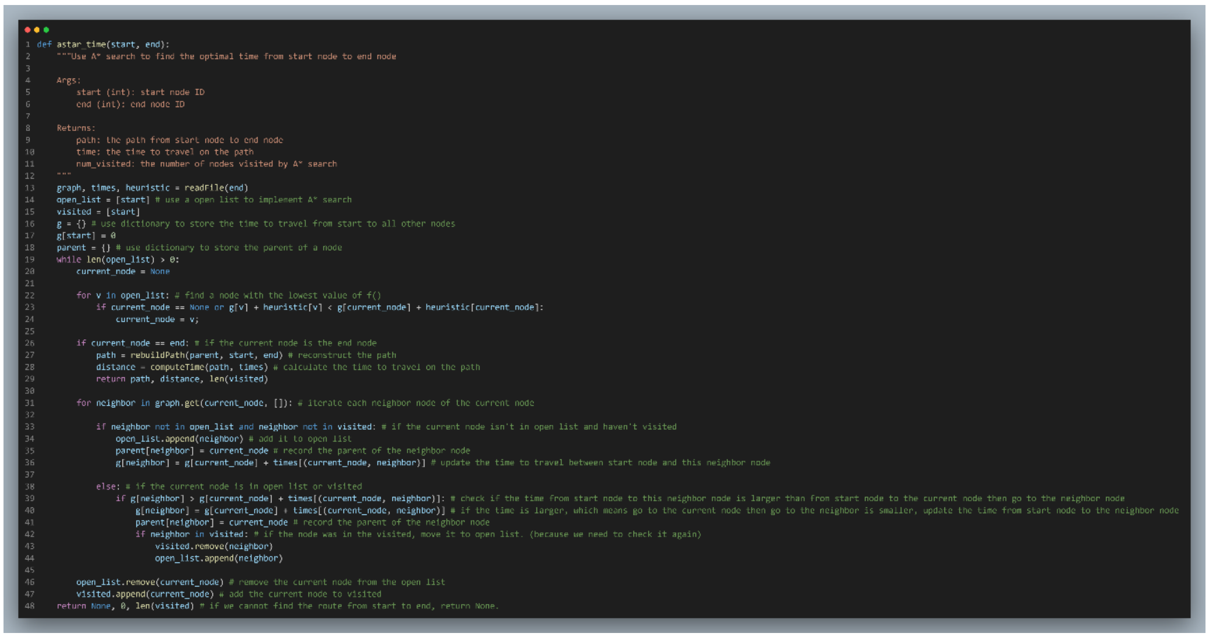
```python
def readFile(end):
    """Read edges.csv, save the adjacency list of the graph, and save distances between two nodes.
    Read heuristic.csv, save the heuristic function of each node.

    Args:
        end (int): the end node ID

    Returns:
        graph: a dictionary, where the key is the node and value is a list of adjacent nodes.
        distances: a dictionary, where the key is a tuple with two adjacent nodes and value is the distance between them.
        heuristic: a dictionary, where the key is the node and the value is the corresponding value of the heuristic function of the node.
    """
    graph = {}
    distances = {}
    heuristic = {}
    with open(edgeFile, newline='') as csvfile: # read edgeFile and save the graph and the distances
        rows = csv.DictReader(csvfile)
        for row in rows:
            if int(row['start']) not in graph:
                graph[int(row['start'])]=[]
            graph[int(row['start'])].append(int(row['end']))
            distances[(int(row['start']), int(row['end']))] = float(row['distance'])

    with open(heuristicFile, newline='') as csvfile: # read heuristicFile and save the heuristic funtion
        rows = csv.DictReader(csvfile)
        for row in rows:
            heuristic[int(row['node'])] = float(row[str(end)])
    return graph, distances, heuristic
```



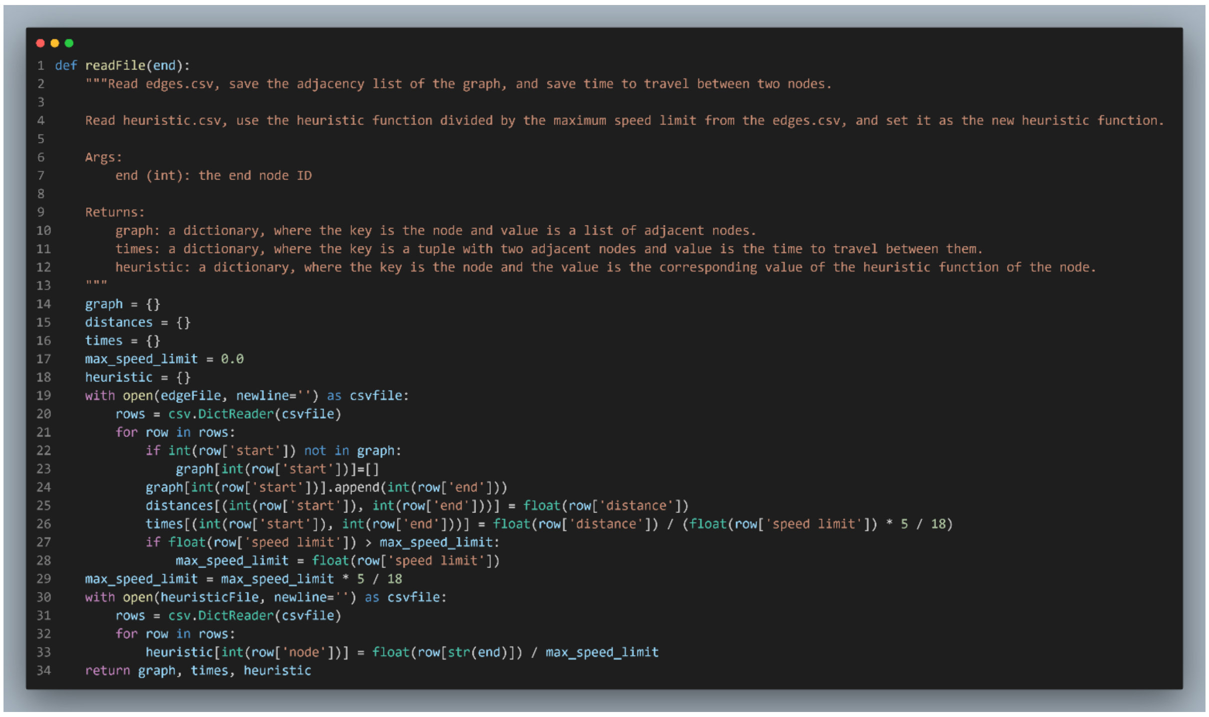
```python
def astar(start, end):
    """Use A* search to find the optimal path from start node to end node

    Args:
        start (int): start node ID
        end (int): end node ID

    Returns:
        path: the path from start node to end node
        dist: the total distance of the path
        num_visited: the number of nodes visited by A* search
    """
    graph, distances, heuristic = readFile(end)
    open_list = [start] # use a open list to implement A* search
    visited = [start]
    g = {} # use dictionary to store the current distances from start to all other nodes
    g[start] = 0
    parent = {} # use dictionary to store the parent of a node
    while len(open_list) > 0:
        current_node = None

        for v in open_list: # find a node with the lowest value of f()
            if current_node == None or g[v] + heuristic[v] < g[current_node] + heuristic[current_node]:
                current_node = v;

        if current_node == end: # if the current node is the end node
            path = rebuildPath(parent, start, end) # reconstruct the path
            distance = computeDistance(path, distances) # calculate the distance of the path
            return path, distance, len(visited)

        for neighbor in graph.get(current_node, []): # iterate each neighbor node of the current node

            if neighbor not in open_list and neighbor not in visited: # if the current node isn't in open list and haven't visited
                open_list.append(neighbor) # add it to open list
                parent[neighbor] = current_node # record the parent of the neighbor node
                g[neighbor] = g[current_node] + distances[(current_node, neighbor)] # update the distance between start node and this neighbor node

            else: # if the current node is in open list or visited
                if g[neighbor] > g[current_node] + distances[(current_node, neighbor)]: # check if the distance from start node to this neighbor node is larger than from start node to the current node then go to the neighbor node
                    g[neighbor] = g[current_node] + distances[(current_node, neighbor)] # if the distance is larger, which means go to the current node then go to the neighbor is smaller, update the distance from start node to the neighbor node
                    parent[neighbor] = current_node # record the parent of the neighbor node
                    if neighbor in visited: # if the node was in the visited, move it to open list. (because we need to check it again)
                        visited.remove(neighbor)
                        open_list.append(neighbor)

        open_list.remove(current_node) # remove the current node from the open list
        visited.append(current_node) # add the current node to visited
    return None, 0, len(visited) # if we cannot find the route from start to end, return None.


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}')
```

The rebuildPath and the computeDistance function are the same as the previous one, so I wouldn’ t paste it again.

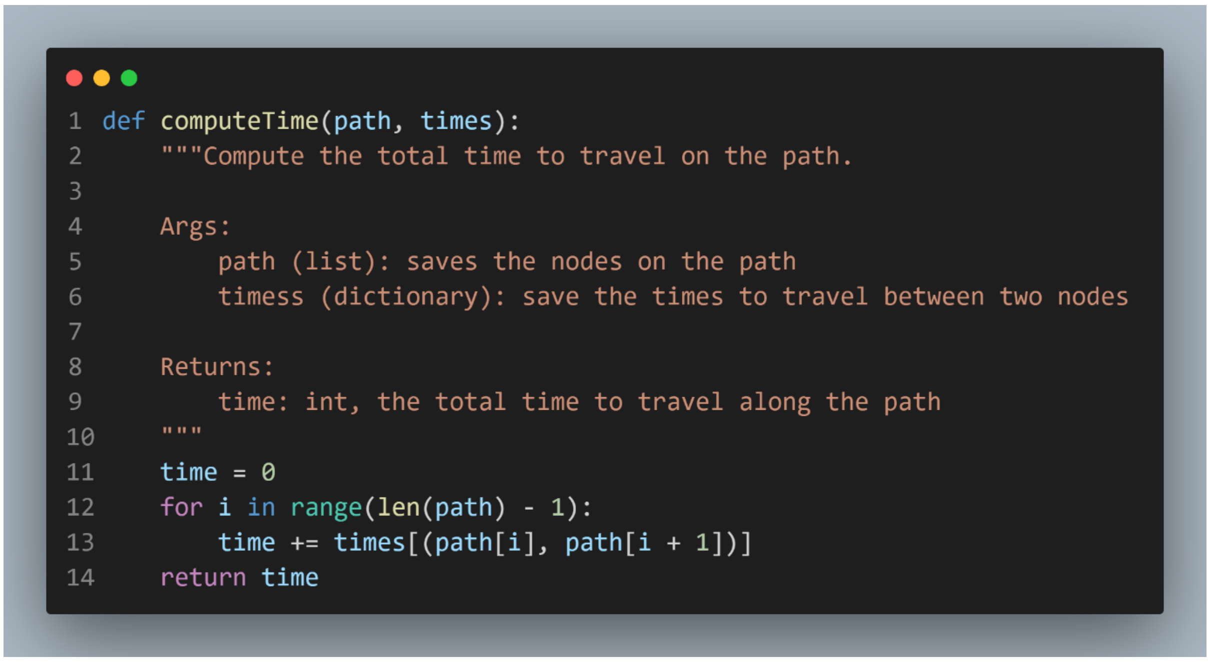
- **Part 6**



```python
def astar_time(start, end):
    """Use A* search to find the optimal time from start node to end node

    Args:
        start (int): start node ID
        end (int): end node ID

    Returns:
        path: the path from start node to end node
        time: the time to travel on the path
        num_visited: the number of nodes visited by A* search
    """
    graph, times, heuristic = readFile(end)
    open_list = [start] # use a open list to implement A* search
    visited = [start]
    g = {} # use dictionary to store the time to travel from start to all other nodes
    g[start] = 0
    parent = {} # use dictionary to store the parent of a node
    while len(open_list) > 0:
        current_node = None

        for v in open_list: # find a node with the lowest value of f()
            if current_node == None or g[v] + heuristic[v] < g[current_node] + heuristic[current_node]:
                current_node = v;

        if current_node == end: # if the current node is the end node
            path = rebuildPath(parent, start, end) # reconstruct the path
            distance = computeTime(path, times) # calculate the time to travel on the path
            return path, distance, len(visited)

        for neighbor in graph.get(current_node, []): # iterate each neighbor node of the current node

            if neighbor not in open_list and neighbor not in visited: # if the current node isn't in open list and haven't visited
                open_list.append(neighbor) # add it to open list
                parent[neighbor] = current_node # record the parent of the neighbor node
                g[neighbor] = g[current_node] + times[(current_node, neighbor)] # update the time to travel between start node and this neighbor node

            else: # if the current node is in open list or visited
                if g[neighbor] > g[current_node] + times[(current_node, neighbor)]: # check if the time from start node to this neighbor node is larger than from start node to the current node then go to the neighbor node
                    g[neighbor] = g[current_node] + times[(current_node, neighbor)] # if the time is larger, which means go to the current node then go to the neighbor is smaller, update the time from start node to the neighbor node
                    parent[neighbor] = current_node # record the parent of the neighbor node
                    if neighbor in visited: # if the node was in the visited, move it to open list. (because we need to check it again)
                        visited.remove(neighbor)
                        open_list.append(neighbor)

        open_list.remove(current_node) # remove the current node from the open list
        visited.append(current_node) # add the current node to visited
    return None, 0, len(visited) # if we cannot find the route from start to end, return None.
```



```python
def readFile(end):
    """Read edges.csv, save the adjacency list of the graph, and save time to travel between two nodes.

    Read heuristic.csv, use the heuristic function divided by the maximum speed limit from the edges.csv, and set it as the new heuristic function.

    Args:
        end (int): the end node ID

    Returns:
        graph: a dictionary, where the key is the node and value is a list of adjacent nodes.
        times: a dictionary, where the key is a tuple with two adjacent nodes and value is the time to travel between them.
        heuristic: a dictionary, where the key is the node and the value is the corresponding value of the heuristic function of the node.
    """
    graph = {}
    distances = {}
    times = {}
    max_speed_limit = 0.0
    heuristic = {}
    with open(edgeFile, newline='') as csvfile:
        rows = csv.DictReader(csvfile)
        for row in rows:
            if int(row['start']) not in graph:
                graph[int(row['start'])]=[]
            graph[int(row['start'])].append(int(row['end']))
            distances[(int(row['start']), int(row['end']))] = float(row['distance'])
            times[(int(row['start']), int(row['end']))] = float(row['distance']) / (float(row['speed limit']) * 5 / 18)
            if float(row['speed limit']) > max_speed_limit:
                max_speed_limit = float(row['speed limit'])
    max_speed_limit = max_speed_limit * 5 / 18
    with open(heuristicFile, newline='') as csvfile:
        rows = csv.DictReader(csvfile)
        for row in rows:
            heuristic[int(row['node'])] = float(row[str(end)]) / max_speed_limit
    return graph, times, heuristic
```



```python
def computeTime(path, times):
    """Compute the total time to travel on the path.

    Args:
        path (list): saves the nodes on the path
        timess (dictionary): save the times to travel between two nodes

    Returns:
        time: int, the total time to travel along the path
    """
    time = 0
    for i in range(len(path) - 1):
        time += times[(path[i], path[i + 1])]
    return time
```

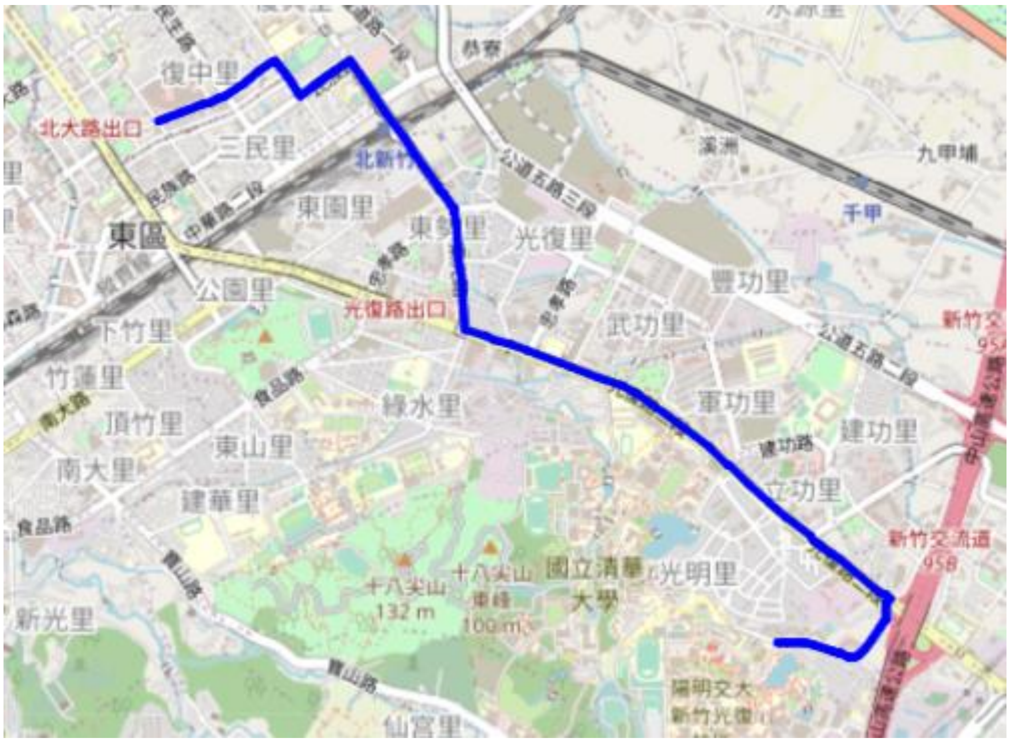
The rebuildPath function is the same as the previous one, so I wouldn’ t paste it again.

Part II. Results & Analysis (12%):

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

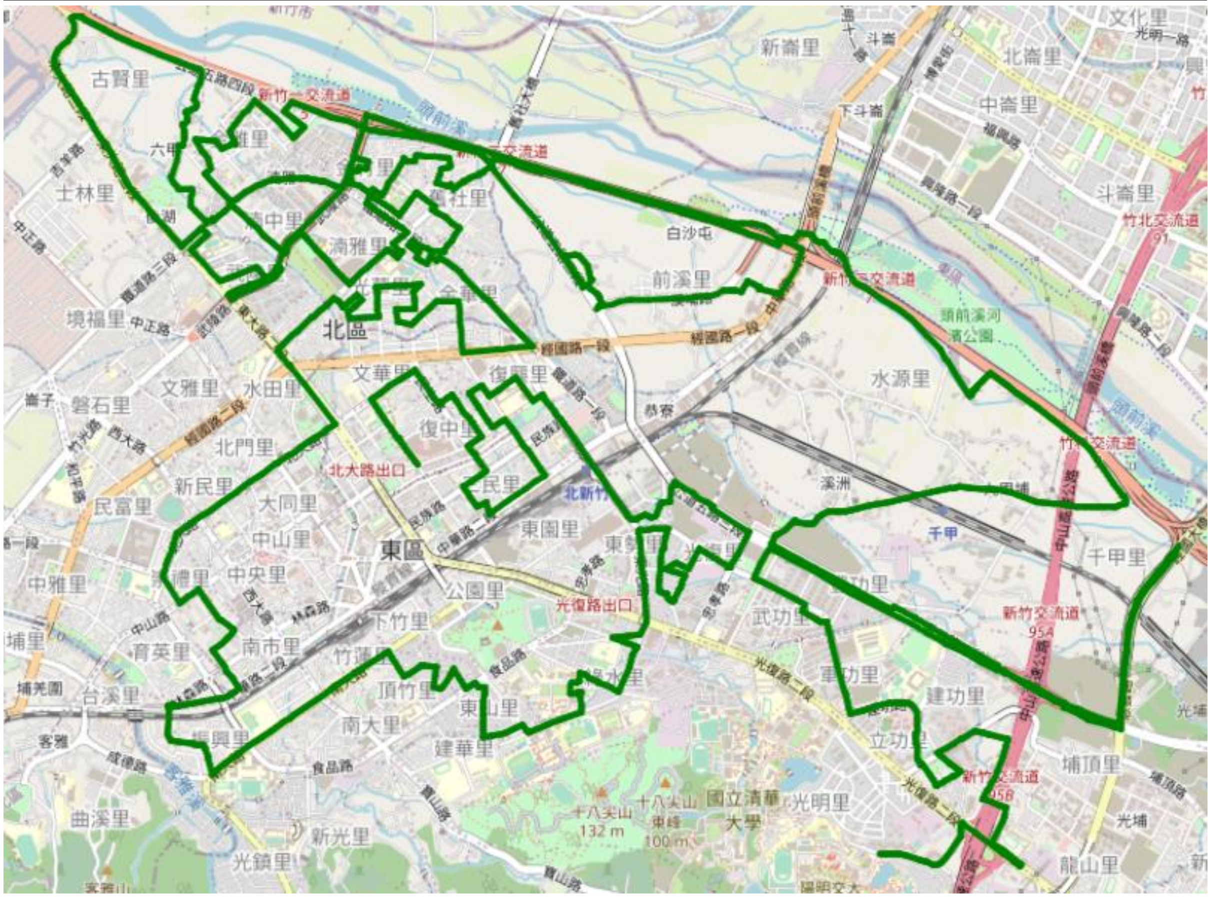
- BFS

```
The number of nodes in the path found by BFS: 88
Total distance of path found by BFS: 4978.8820000000005 m
The number of visited nodes in BFS: 4403
```



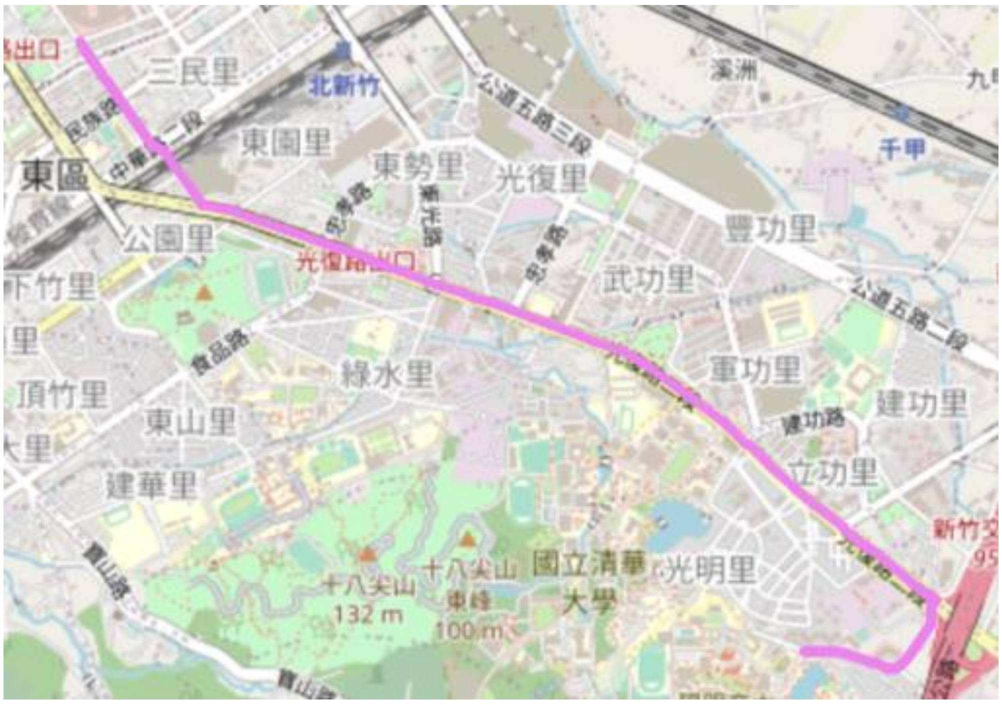
■ DFS (stack)

```
The number of nodes in the path found by DFS: 1232
Total distance of path found by DFS: 57208.987000000045 m
The number of visited nodes in DFS: 4382
```



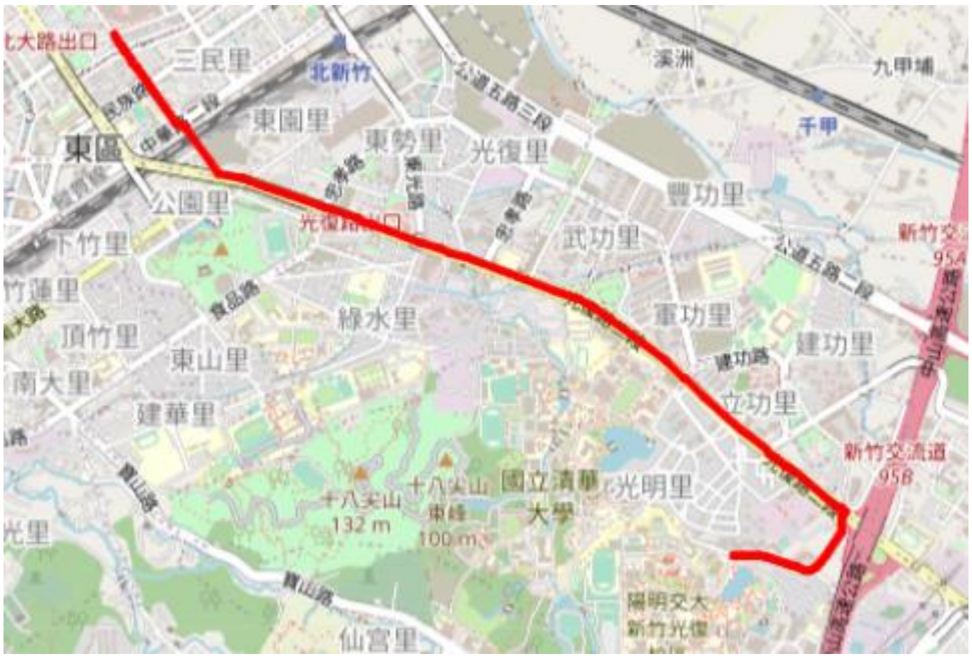
- UCS

```
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: 5086
```

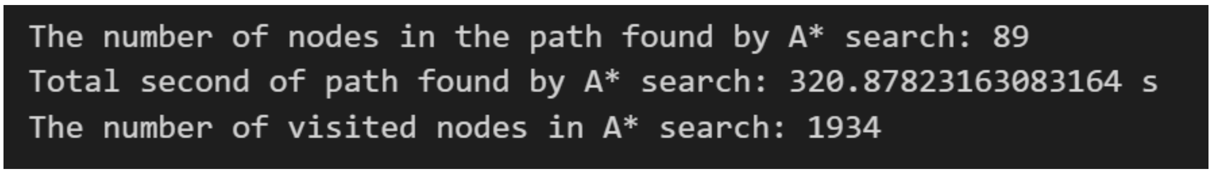


- Astar

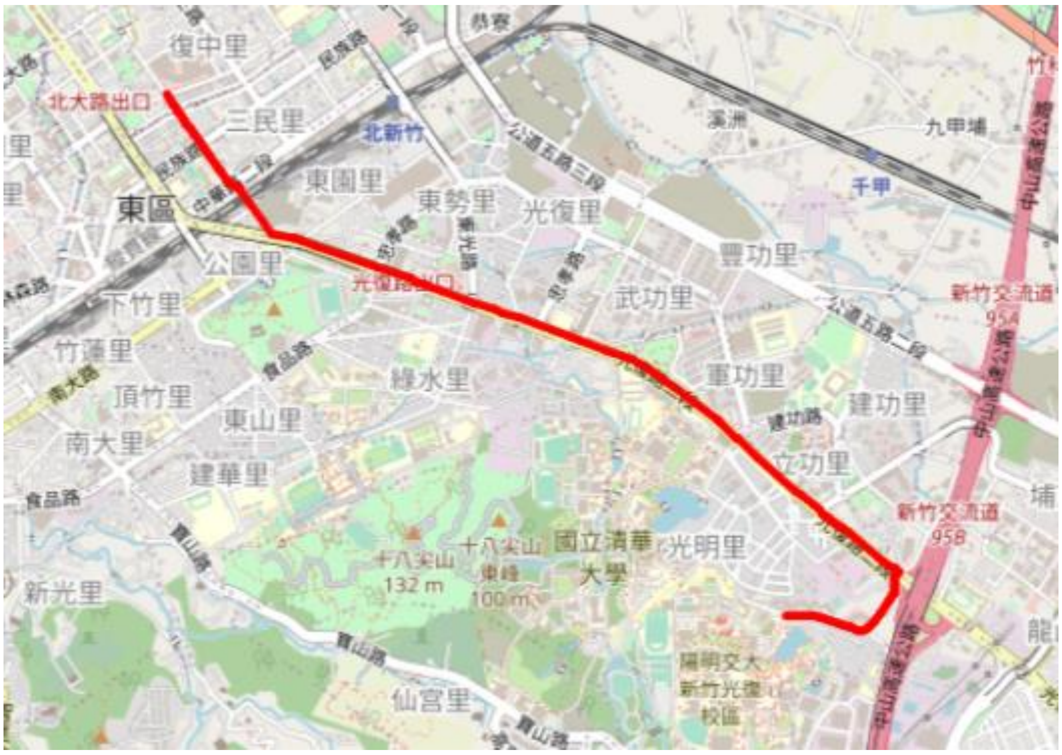
```
The number of nodes in the path found by A* search: 89
Total distance of path found by A* search: 4367.881 m
The number of visited nodes in A* search: 261
```



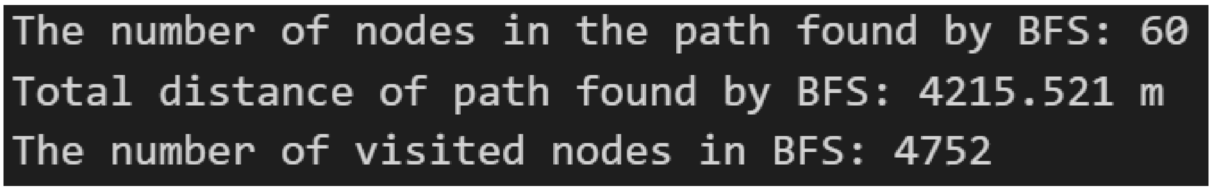
- Astar_time



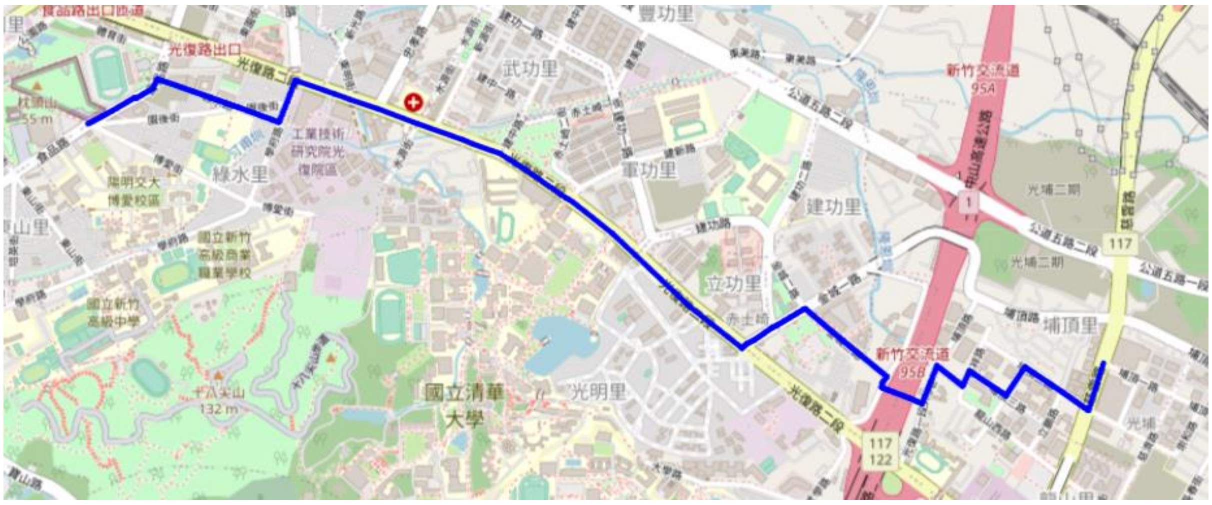
```
The number of nodes in the path found by A* search: 89
Total second of path found by A* search: 320.87823163083164 s
The number of visited nodes in A* search: 1934
```



- **Test 2：from Hsinchu Zoo (ID: 426882161) to COSTCO Hsinchu Store (ID: 1737223506)**
  - BFS

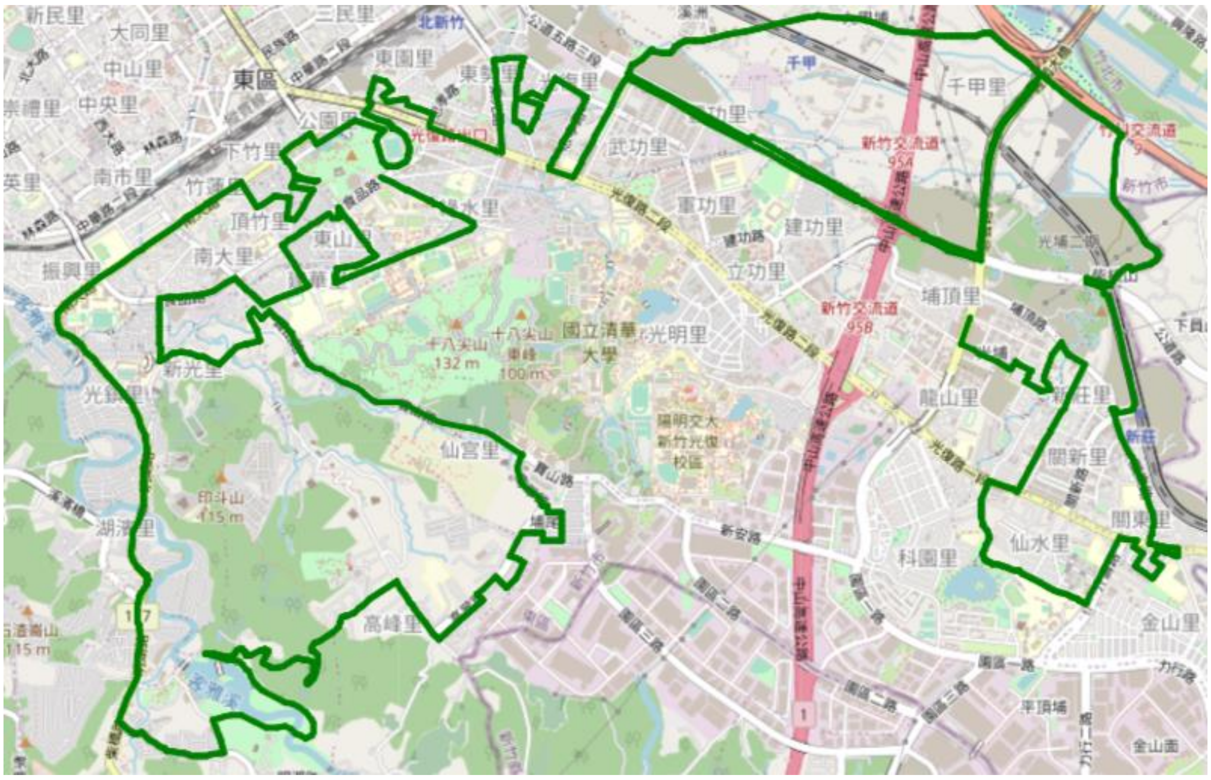


```
The number of nodes in the path found by BFS: 60
Total distance of path found by BFS: 4215.521 m
The number of visited nodes in BFS: 4752
```



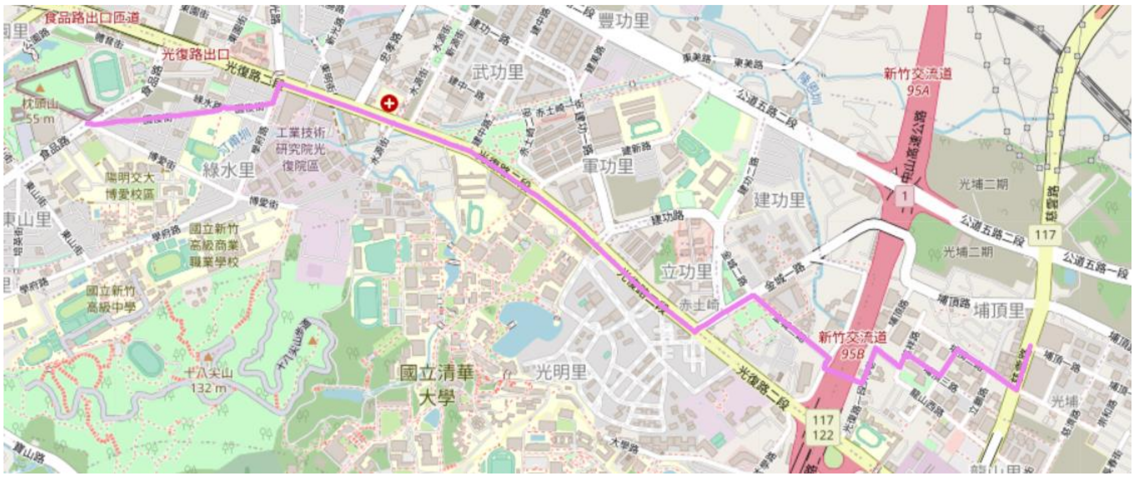
- DFS (stack)

```
The number of nodes in the path found by DFS: 998
Total distance of path found by DFS: 41094.65799999992 m
The number of visited nodes in DFS: 8629
```

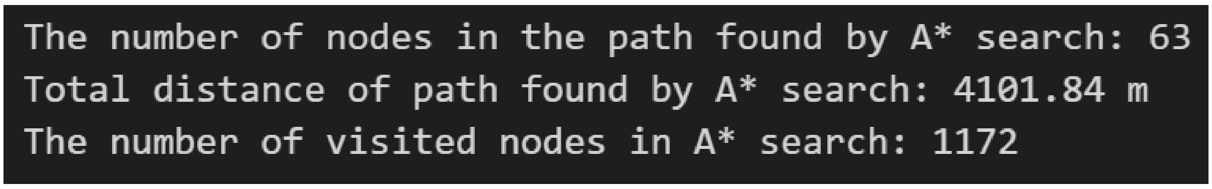


- UCS

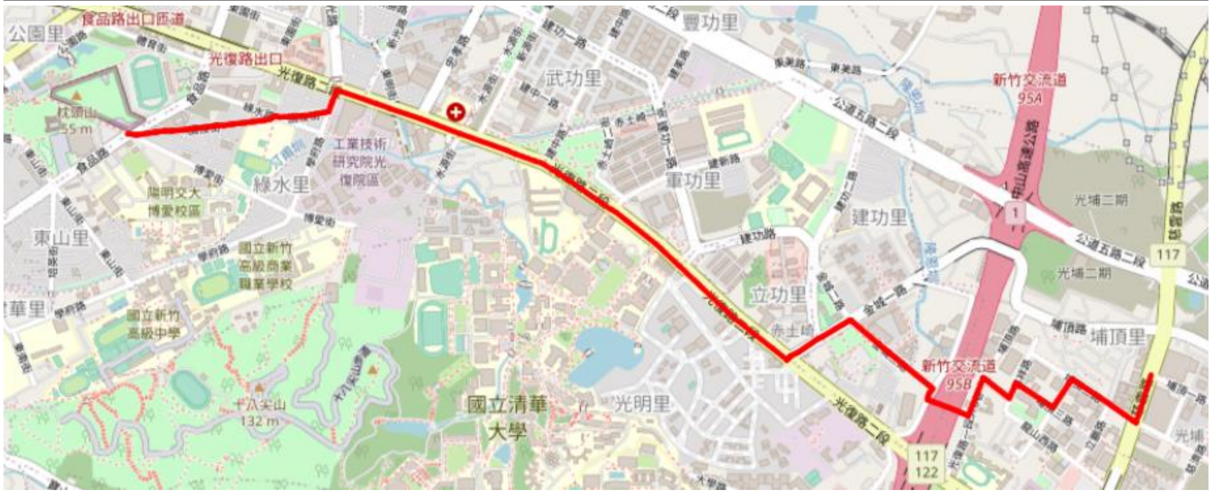
```
The number of nodes in the path found by UCS: 63
Total distance of path found by UCS: 4101.84 m
The number of visited nodes in UCS: 7213
```



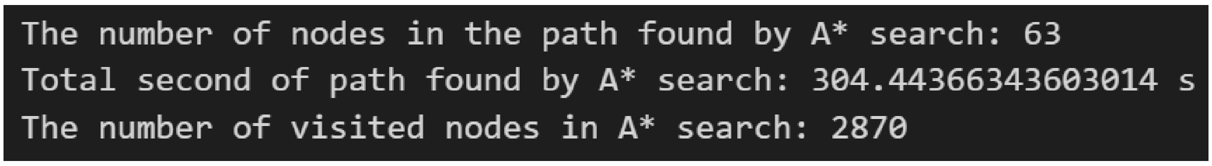
- Astar



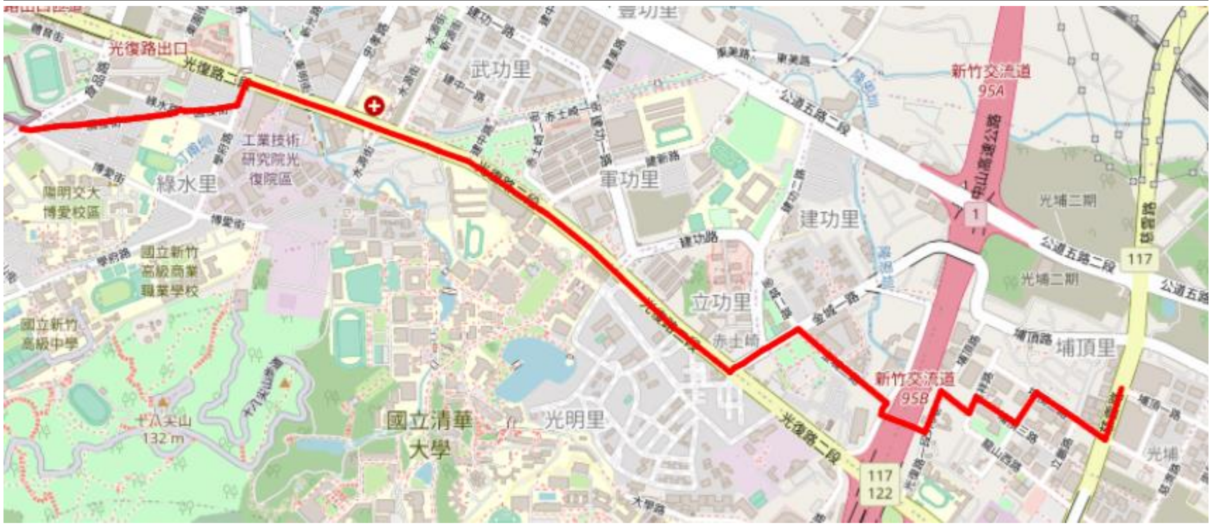
```
The number of nodes in the path found by A* search: 63
Total distance of path found by A* search: 4101.84 m
The number of visited nodes in A* search: 1172
```



- Astar_time

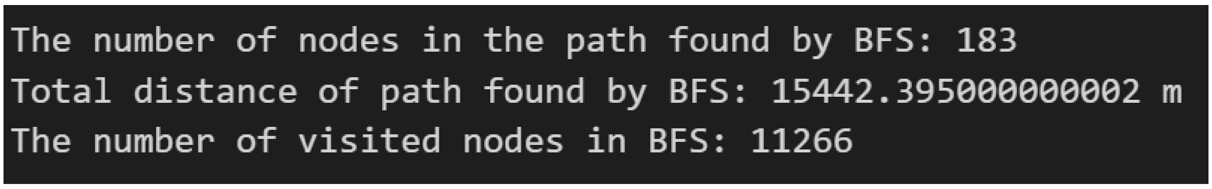


```
The number of nodes in the path found by A* search: 63
Total second of path found by A* search: 304.44366343603014 s
The number of visited nodes in A* search: 2870
```

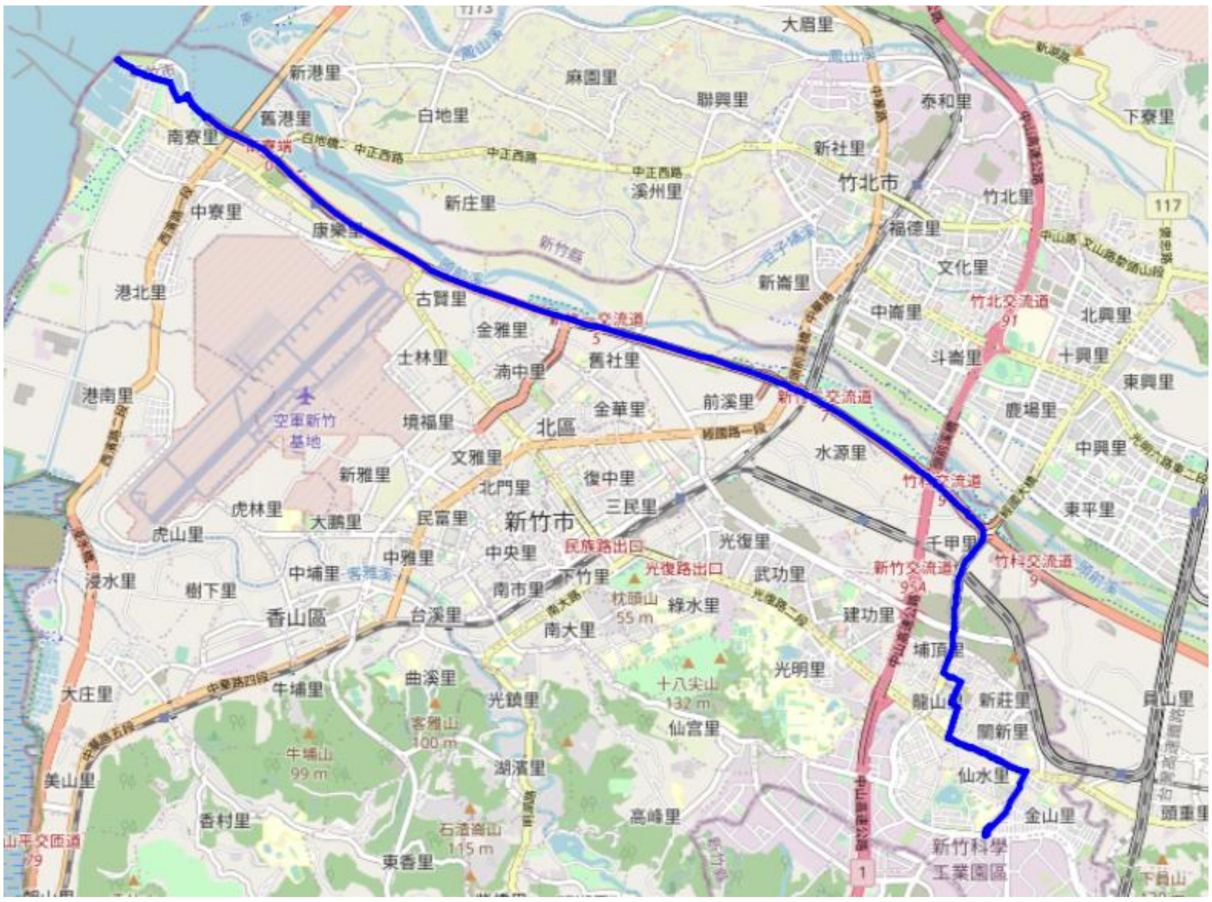


- **Test 3：from National Experimental High School At Hsinchu Science Park (ID: 1718165260)to Nanliao Fighing Port (ID: 8513026827)**

- BFS

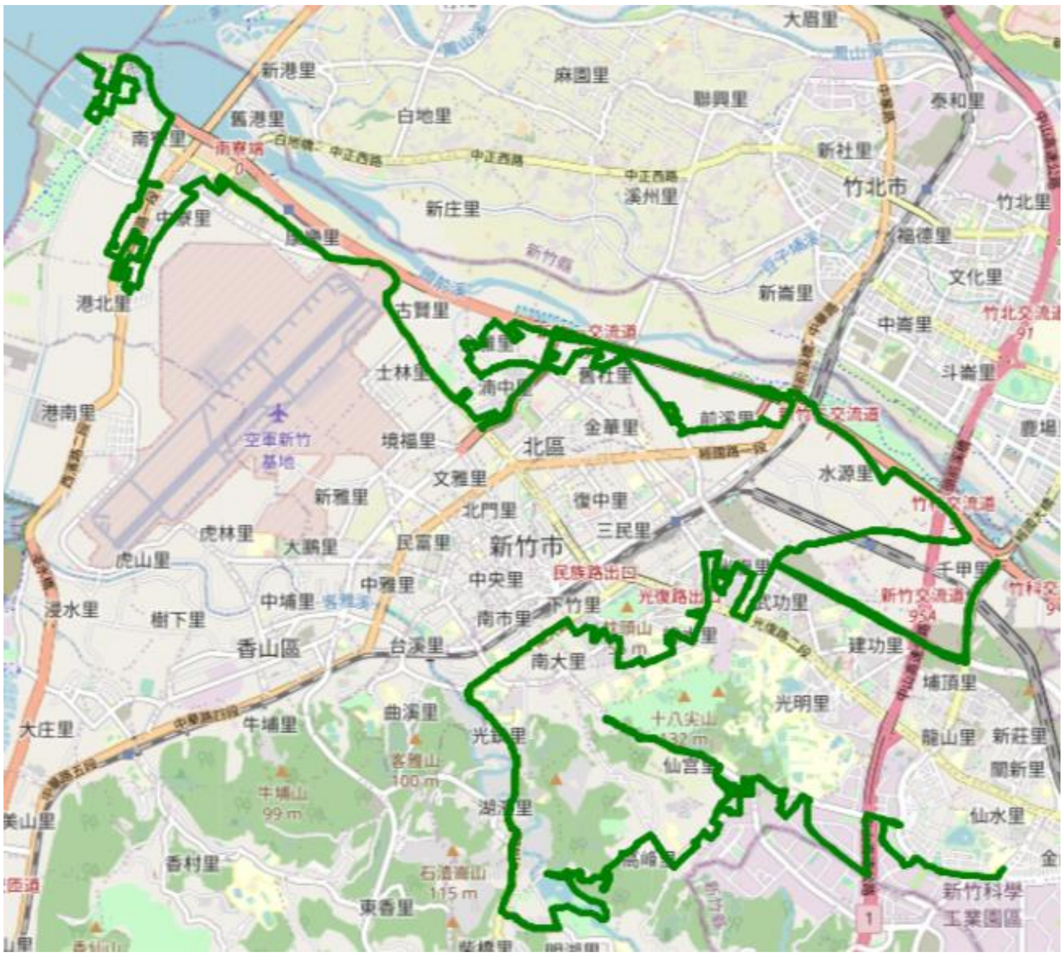


```
The number of nodes in the path found by BFS: 183
Total distance of path found by BFS: 15442.395000000002 m
The number of visited nodes in BFS: 11266
```

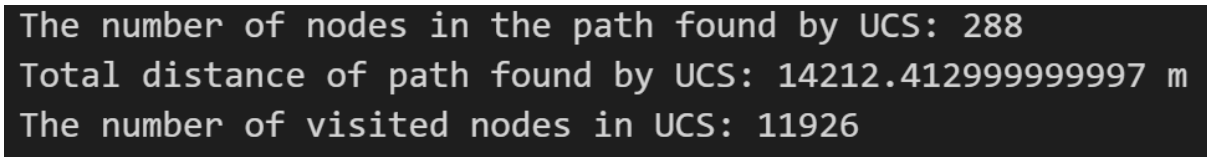


- DFS (stack)

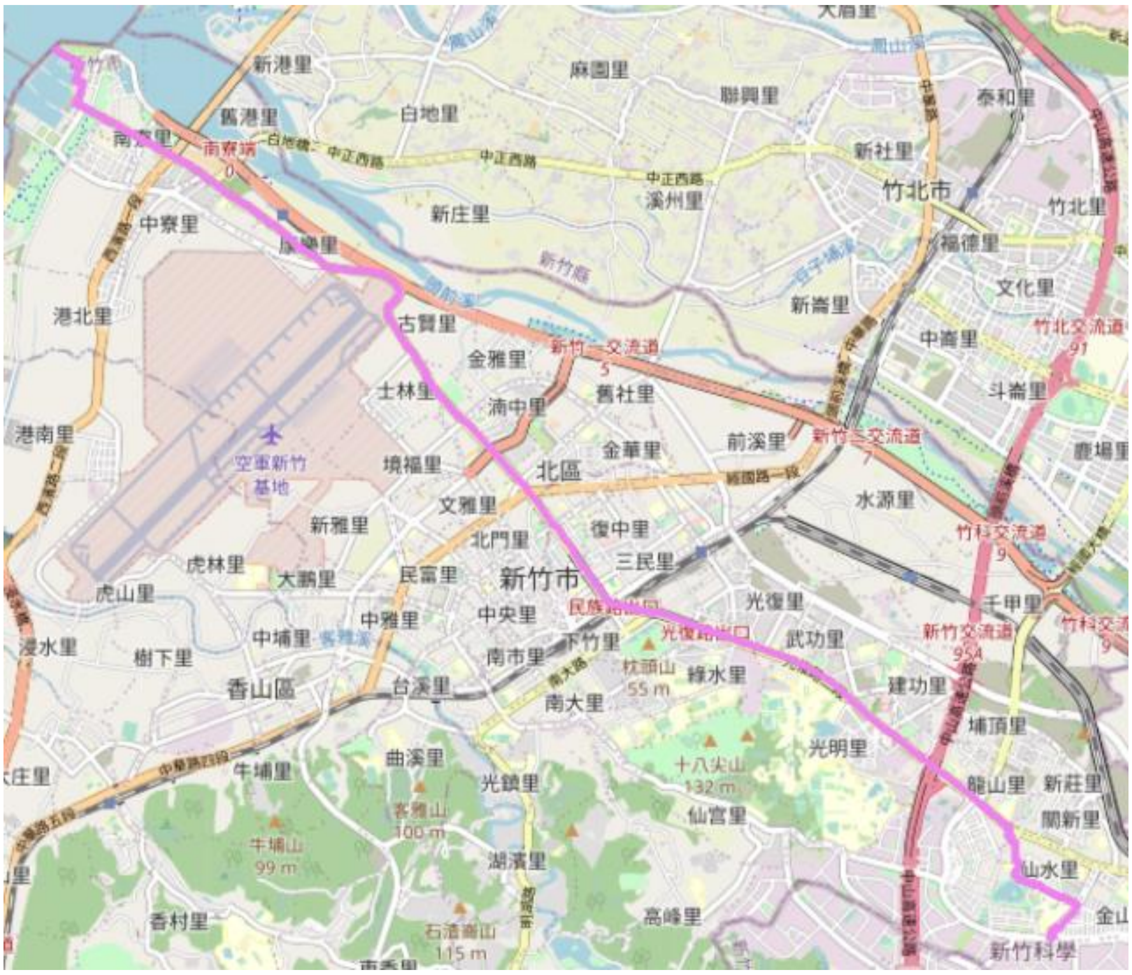
```
The number of nodes in the path found by DFS: 1521
Total distance of path found by DFS: 64821.60399999987 m
The number of visited nodes in DFS: 3372
```



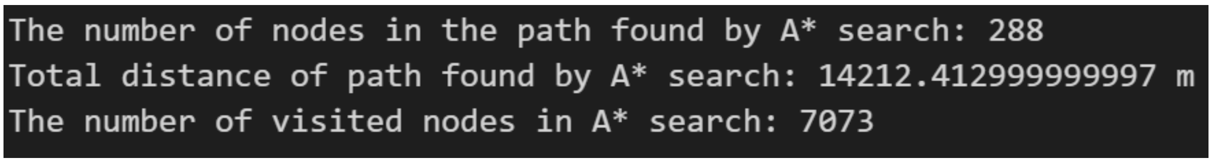
- UCS



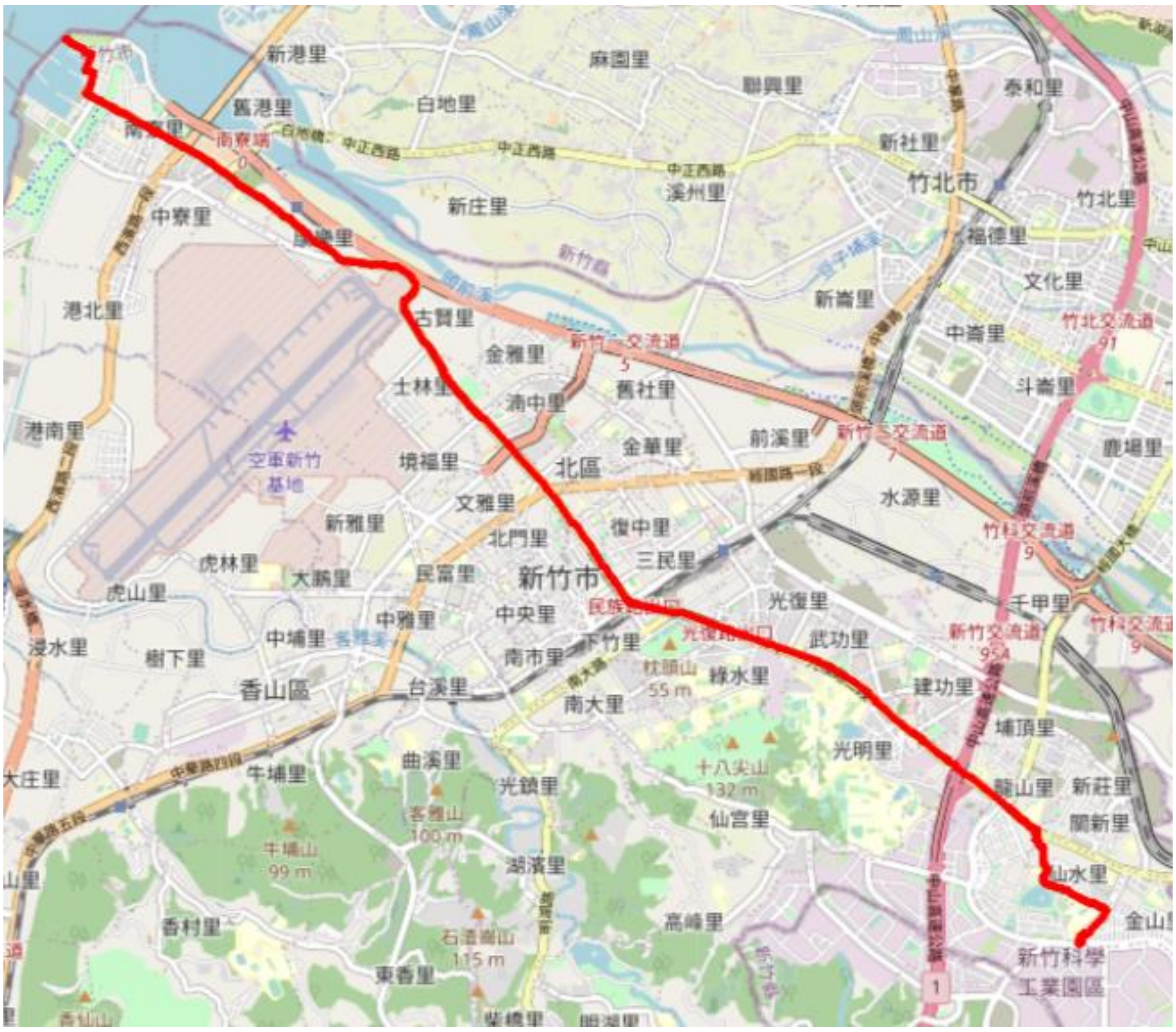
```
The number of nodes in the path found by UCS: 288
Total distance of path found by UCS: 14212.412999999997 m
The number of visited nodes in UCS: 11926
```



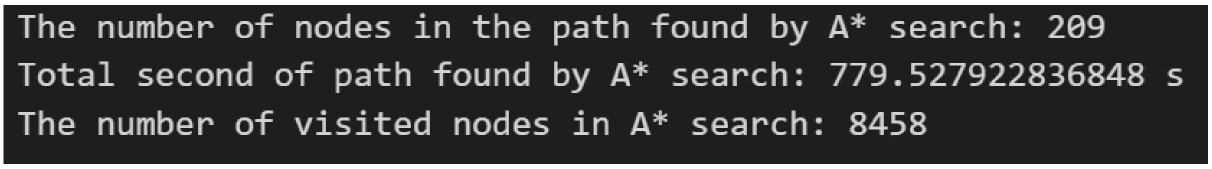
- Astar



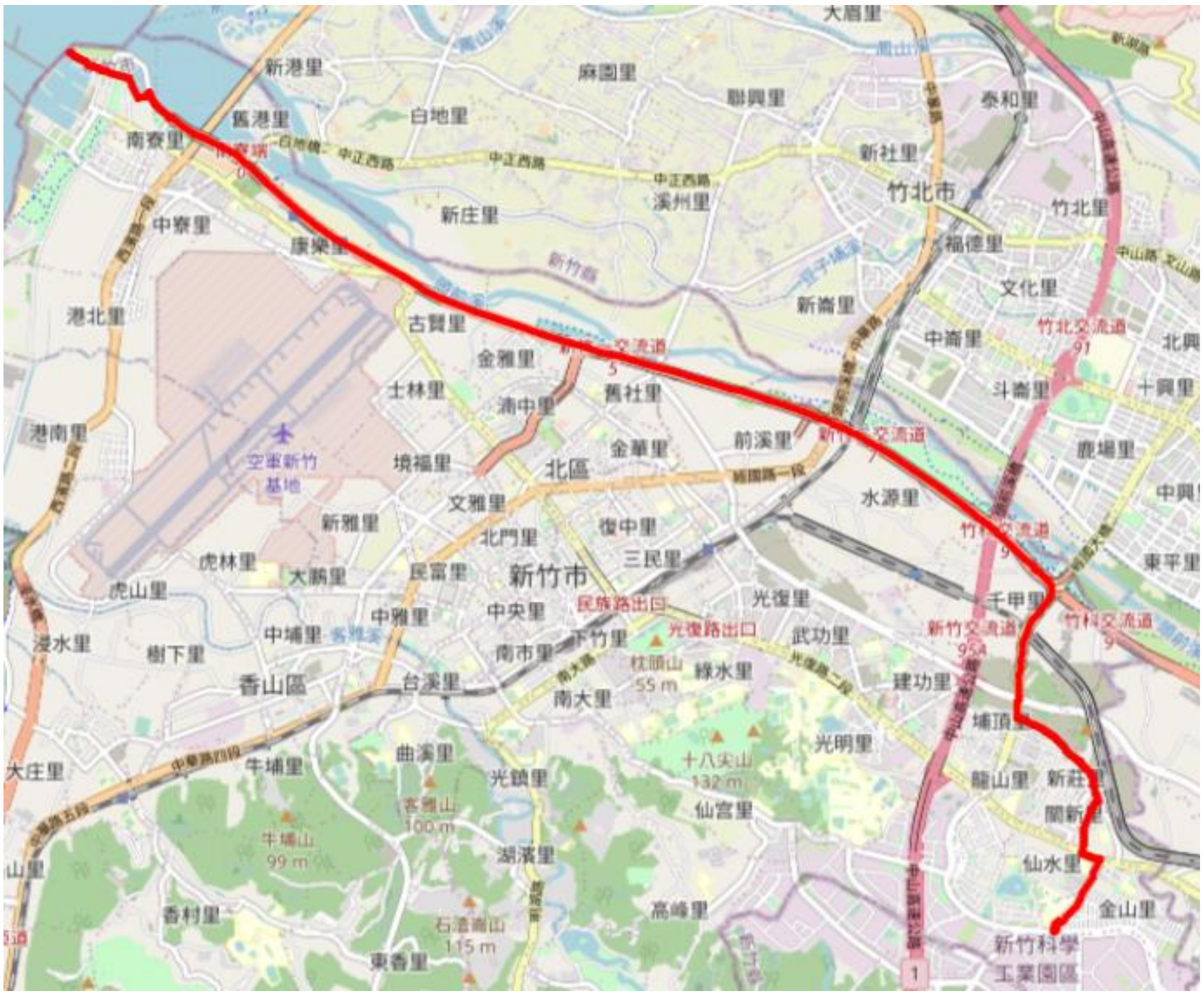
```
The number of nodes in the path found by A* search: 288
Total distance of path found by A* search: 14212.412999999997 m
The number of visited nodes in A* search: 7073
```



- Astar_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: 8458
```



- Overall Analysis:
  - The average performance of each path finding algorithm:

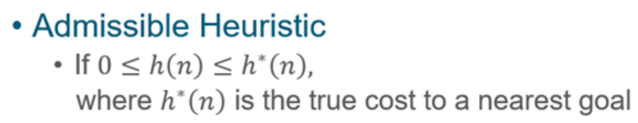
    Based on my testing of the path finding algorithms, the A* search and UCS algorithms have shown to have the best performance in terms of finding an optimal path. However, UCS requires iterating through more nodes than A* search.

    On the other hand, BFS also performs reasonably well and is able to find a decent result with fewer node iterations than UCS.

    In contrast, DFS typically produces longer paths than the other algorithms. However, the number of nodes iterated by DFS can vary depending on the way the adjacent list is stored. For instance, in test 1 and test 2, DFS iterated more nodes than BFS. However, in test 3, DFS only iterated 2494 nodes while BFS iterated 11266 nodes.

- The design of heuristic function in part 6:

  For the design of the heuristic function in part 6, I used the original heuristic function found in the heuristic.csv file and divided it by the maximum speed limit that can be found in the edge.csv file. This heuristic function is considered admissible because it follows the admissible heuristic principle that was discussed in class, as shown in the following image:

  - Admissible Heuristic
    - If $0 \leq h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to a nearest goal

  Since there are various speed limits on each edge, selecting the maximum speed limit implies that we can travel faster than the limit on some edges. As a result, the heuristic function I designed can ensure that it is always less than or equal to the true cost to the nearest goal, making it an admissible heuristic.

- The comparison of Part 4 and Part 6:

  In part 4, the number of visited nodes is typically lower than in part 6. Additionally, in test 1 and test 2, both algorithms discovered the same path. However, in test 3, the path found in part 4 differed from the path found in part 6. The reason for this difference is due to the varying speed limits on the edges.

  The pathfinding algorithm used in part 4 does not consider speed limits, whereas the algorithm used in part 6 does. As a result, the path discovered in part 4 may not be optimal or even valid if the edges' speed limits affect the path's feasibility.

  In contrast, the pathfinding algorithm used in part 6 accounts for speed limits when determining the optimal path, which results in a path that is both valid and optimal. As a consequence, the algorithm in part 6 may visit more nodes than the algorithm used in part 4 since it considers additional factors that impact the pathfinding process.

Part III. Question Answering (12%):

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

   While implementing the BFS algorithm, I encountered some difficulties with storing the edge information. In my initial attempt, I used a dictionary to store the edge information where the first node was the key, and the value was a tuple of the second node and the distance between them. However, accessing and implementing the BFS algorithm using this data structure was challenging.

   After a few days, I came up with an alternative solution. I decided to store the adjacent edges and their distances separately. To my surprise, I discovered that a tuple could be used as the key for a dictionary, which simplified the process of accessing the distances between the adjacent nodes significantly.

By storing the adjacent edges and their distances separately and using tuples as the keys for the dictionary, I was able to access and implement the BFS algorithm with greater ease and efficiency.

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.

   Another attribute that is crucial for route finding in the real world is the traffic flow. Knowing the traffic flow of each road segment can help in determining the optimal route to take. High traffic flow can lead to congested roads, which would result in longer travel times and potentially impact the overall travel schedule.

   By incorporating real-time traffic flow information into the route-finding algorithm, users can receive up-to-date information on the current traffic conditions, allowing them to adjust their route and avoid potential traffic delays. This can result in faster and more efficient travel times, as well as a more accurate representation of the expected travel time.

   To sum up, incorporating traffic flow data into the route-finding algorithm can provide a more accurate and reliable navigation experience for users and help them make better-informed decisions when planning their travel routes.

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

- For the mapping component, one possible solution is to use satellite imagery or aerial photography to create high-resolution maps. These maps can be generated using computer vision techniques, such as image segmentation and object detection, to identify and label different features, such as roads, buildings, and landmarks. The maps can also be updated in real-time using crowdsourcing or sensor data from vehicles or drones.

- For the localization component, one possible solution is to use GPS or other positioning technologies, such as Bluetooth beacons or Wi-Fi triangulation, to determine the location of the user or vehicle. However, GPS signals can be affected by interference or obstacles, such as tall buildings or tunnels, which can result in inaccurate or unreliable location estimates.

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 ETA based on their mechanism. Please define a **dynamic heuristic equation** for ETA and explain the rationale of your design. Hint: You can consider meal prep time, delivery priority, multiple orders, etc.

A dynamic heuristic equation for ETA in Uber Eats can be defined as follows:

1. If there is no order on the nearest deliver,

ETA = Meal preparation time + (Time to travel from the location of nearest deliver to restaurant) + (Time to travel from restaurant to customer)

2. If there is one or more order on the nearest deliver and the priority of this new order is higher than the previous one,

ETA = Meal preparation time + (Time to travel from the location of nearest deliver to restaurant) + (Time to travel from restaurant to customer who is the highest priority) + The average interacting time between delivers and customers + (Time to travel from customer who is the highest priority to the second priority) + The average interacting time between delivers and customers + … + (Time to travel from the previous customer to the customer who ordered this order)

Noted that we don't need to consider the previous meal with lower priority in this case, because VIP is always the best 😊.

- The logic flow I think a general deliver will do is:

  1. Turn on the App and check the nearest order and the destination

  2. Click accept the order and go to the restaurant

  3. While waiting the meal, check whether there is an order that will have a short distance between the deliver and the restaurant, and a short distance between two destinations. If so, click accept.

  4. Deliver the meal to the customer.

  I must admit that this is a relatively easy logic flow. In the real world, the nearest deliver may not to accept the meal due to the time and the money. Or the deliver may not accept another order while waiting the meal due to some reasons. This is just my logic flow, don't dig in too much.

- When the nearest deliver has no orders, we can easily define the ETA.

  ■ Meal preparation time: The amount of time it takes for the restaurant to prepare the order. This factor can vary depending on the type of cuisine, the complexity of the order, and the restaurant's current workload.

  ■ Time to travel from the location of nearest deliver to restaurant: If we need to implement this, there are too many attributes we need to take into consideration. For example, the traffic light waiting time and the current traffic flow on each road. I think when Uber needs to implement this, they can simply get a google map API to know the possible travel time of the deliver to the restaurant.

  - Time to travel from restaurant to customer: This is the same problem as “Time to travel from the location of nearest deliver to restaurant”.

- However, if there are one or more orders on the deliver, I think we need to take the priority of each customer. For example, Uber VIP may have the highest priority, the normal user have the second high priority, and the user who may ordered the meal and didn’t pay for it may have the lowest priority. I assume that the priority had been decided in the Uber’s App.

  - Meal preparation time + (Time to travel from the location of nearest deliver to restaurant): This is like the previous one.
  - Time to travel from restaurant to customer who is the highest priority: This is the same problem as the previous “Time to travel from restaurant to customer”.
  - The average interacting time between delivers and customers: Because there are multiple order between customers, I think Uber may have the average interacting time between delivers and customers in their database, so we need to take it into consideration.
  - (Time to travel from customer who is the highest priority to the second priority) + The average interacting time between delivers and customers + … + (Time to travel from the previous customer to the customer who ordered this order): This is like combining previous two problems. Noted that if the priority of two meals is the same, pick the closer one to the deliver.
- Noted that meal preparation time may overlap with the time to travel from the location of nearest deliver to restaurant. In my opinion, the value of admissible heuristic must be smaller than the true cost to the nearest goal, so it is tolerable.