Homework 2: Route Finding

Name : 林伯偉 Student ID : 109612019

Part I. Implementation (6%):

- **Part 1**

```
# Begin your code (Part 1)
"""
    1.The code begins by initializing dictionaries to store edge distances (edges_distance),
      speed limits (speed_limit), and the maximum speed limit (maximum_speed_limit).
    2.It then reads data from the CSV files 'edges.csv' and 'heuristic.csv',
      populating the dictionaries with edge distances and heuristic distances respectively.
    3.Next, the code initializes variables for the path, distance, number of visited nodes,
      and a dictionary to store previous nodes in the BFS traversal.
    4.The breadth-first search algorithm is performed to find the shortest path from the start node to the end node.
    5.Finally, the code reconstructs the shortest path and calculates the total distance along that path,
      returning the path, distance, and number of visited nodes as a tuple.
"""
edges_distance = {}
speed_limit = {}
maximum_speed_limit = float('-inf')

# Read data from the CSV files 'edges.csv' and 'heuristic.csv'
with open('edges.csv', newline='') as csv_file:
    reader = csv.reader(csv_file, delimiter=',')
    next(reader)  # Skip header
    for row in reader:
        source, destination, distance, speed = map(float, row)  # Change int to float
        edges_distance.setdefault(int(source), {})[int(destination)] = distance
        speed_limit.setdefault(int(source), {})[int(destination)] = speed / 3.6
        maximum_speed_limit = max(maximum_speed_limit, speed / 3.6)

heuristic_distance = {}
with open('heuristic.csv', newline='') as csv_file:
    reader = csv.reader(csv_file, delimiter=',')
    next(reader)  # Skip header
    for row in reader:
        node, *heuristics = map(float, row)
        heuristic_distance[int(node)] = heuristics


# Initialize variables for path, distance, number of visited nodes, and previous nodes
path = []
dist = 0.0
num_visited = 0
previous_node = {}
bfs_queue = [start]

while bfs_queue:
    num_visited += 1
    node = bfs_queue.pop(0)
    if node == end:
        break
    if node not in edges_distance:
        continue
    for destination in edges_distance[node]:
        if destination == start or destination in previous_node:
            continue
        bfs_queue.append(destination)
        previous_node[destination] = node

path.append(end)
while path[0] in previous_node:
    dist += edges_distance[previous_node[path[0]]][path[0]]
    path.insert(0, previous_node[path[0]])

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

- **Part 2**

```python
# Begin your code (Part 2)
"""
    1.The code initializes dictionaries to store edge distances (edges_distance),
      speed limits (speed_limit), and maximum speed (max_speed).
    2.It reads data from the CSV files 'edges.csv' and 'heuristic.csv', populating the dictionaries.
    3.Variables for path, distance, number of visited nodes,
      and a dictionary to store previous nodes in the DFS traversal are initialized.
    4.Depth-first search algorithm is performed to find a path from the start node to the end node.
    5.The path is reconstructed, and the total distance calculated.
    6.The function returns the path, distance, and number of visited nodes.
"""
# Initialize dictionaries to store edge distances, speed limits, and maximum speed limit
edges_distance = {}
speed_limit = {}
maximum_speed_limit = float('-inf')

# Read data from the CSV files 'edges.csv' and 'heuristic.csv'
with open('edges.csv', newline='') as csv_file:
    reader = csv.reader(csv_file, delimiter=',')
    next(reader)  # Skip header
    for row in reader:
        source, destination, distance, speed = map(float, row)  # Change int to float
        edges_distance.setdefault(int(source), {})[int(destination)] = distance
        speed_limit.setdefault(int(source), {})[int(destination)] = speed / 3.6
        maximum_speed_limit = max(maximum_speed_limit, speed / 3.6)

heuristic_distance = {}
with open('heuristic.csv', newline='') as csv_file:
    reader = csv.reader(csv_file, delimiter=',')
    next(reader)  # Skip header
    for row in reader:
        node, *heuristics = map(float, row)
        heuristic_distance[int(node)] = heuristics

# Initialize variables for path, distance, number of visited nodes, and previous nodes
path = []
dist = 0.0
num_visited = 0
previous_node = {}
dfs_stack = []
dfs_stack.append(start)

# Perform depth-first search to find a path from start to end
while dfs_stack:
    num_visited += 1
    node = dfs_stack.pop()
    # If the end node is reached, exit the loop
    if node == end: break
    # Check if the current node has outgoing edges
    for dest in edges_distance.get(node, {}):
        if dest == start or dest in previous_node: continue
        dfs_stack.append(dest)
        previous_node[dest] = node

# Reconstruct the path and calculate the total distance
path.append(end)
while path[0] in previous_node:
    dist += edges_distance[previous_node[path[0]]][path[0]]
    path.insert(0, previous_node[path[0]])

# Return the path, distance, and number of visited nodes
return path, dist, num_visited
# End your code (Part 2)
```

- **Part 3**

```
# Begin your code (Part 3)
"""
    1.The code initializes dictionaries for edges (edges) and heuristics (heuristic), and a variable to store the maximum speed (max_speed).
    2.It reads data from the CSV files 'edges.csv' and 'heuristic.csv', populating the dictionaries accordingly.
    3.Variables for path, distance, and visited nodes are initialized.
    4.UCS algorithm is performed using heapq for priority queue implementation to find the shortest path.
    5.The shortest path is reconstructed, and the total distance calculated.
    6.The function returns the shortest path, total distance, and number of visited nodes.
"""
# Initialize dictionaries to store edge distances, speed limits, and maximum speed limit
edges_distance = {}
speed_limit = {}
maximum_speed_limit = float('-inf')

# Read data from the CSV files 'edges.csv' and 'heuristic.csv'
with open('edges.csv', newline='') as csv_file:
    reader = csv.reader(csv_file, delimiter=',')
    next(reader)  # Skip header
    for row in reader:
        source, destination, distance, speed = map(float, row)  # Change int to float
        edges_distance.setdefault(int(source), {})[int(destination)] = distance
        speed_limit.setdefault(int(source), {})[int(destination)] = speed / 3.6
        maximum_speed_limit = max(maximum_speed_limit, speed / 3.6)

heuristic_distance = {}
with open('heuristic.csv', newline='') as csv_file:
    reader = csv.reader(csv_file, delimiter=',')
    next(reader)  # Skip header
    for row in reader:
        node, *heuristics = map(float, row)
        heuristic_distance[int(node)] = heuristics



# Initialize variables for path, distance, number of visited nodes, and previous nodes
path = []
dist = float(0)
num_visited = int(0)
previous_node = {}
heap = []
heapq.heappush(heap, (float(0), start))

while heap:
    num_visited += 1
    distance, node = heapq.heappop(heap)

    if (node == end):
        break
    if node not in edges_distance:
        continue
    for destination in edges_distance[node]:
        if destination == start or destination in previous_node and previous_node[destination][1] < distance + edges_distance[node][destination]:
            continue
        heapq.heappush(heap, (distance + edges_distance[node][destination], destination))
        previous_node[destination] = (node, distance + edges_distance[node][destination])

path.append(end)
while path[0] in previous_node:
    dist += edges_distance[previous_node[path[0]][0]][path[0]]
    path.insert(0, previous_node[path[0]][0])

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

- **Part 4**

```
# Begin your code (Part 4)
"""
    1.This code implements the A* algorithm to find the shortest path between two nodes in a graph.
    2.It initializes dictionaries to store edge distances and heuristic estimates.
    3.Performs the A* algorithm by iteratively exploring the nodes with the lowest combined cost of distance
      and heuristic estimate until the destination node is reached.
    4.The algorithm reconstructs the shortest path and calculates the total distance traveled.
    5.Finally, it returns the shortest path, total distance, and number of visited nodes.
"""
# Initialize dictionaries to store edge distances, speed limits, and maximum speed limit
edges_distance = {}
speed_limit = {}
maximum_speed_limit = float('-inf')

# Read data from the CSV files 'edges.csv' and 'heuristic.csv'
with open('edges.csv', newline='') as csv_file:
    reader = csv.reader(csv_file, delimiter=',')
    next(reader)  # Skip header
    for row in reader:
        source, destination, distance, speed = map(float, row)  # Change int to float
        edges_distance.setdefault(int(source), {})[int(destination)] = distance
        speed_limit.setdefault(int(source), {})[int(destination)] = speed / 3.6
        maximum_speed_limit = max(maximum_speed_limit, speed / 3.6)

heuristic_distance = {}
with open('heuristic.csv', newline='') as csv_file:
    reader = csv.reader(csv_file, delimiter=',')
    next(reader)  # Skip header
    for row in reader:
        node, *heuristics = map(float, row)
        heuristic_distance[int(node)] = {
            1079387396: heuristics[0],
            1737223506: heuristics[1],
            8513026827: heuristics[2]
        }


# Initialize variables for path, distance, number of visited nodes, and previous nodes
path = []
dist = 0
num_visited = 0
previous_node = {}
# Initialize heap with start node and its heuristic distance to the end node
heap = [(heuristic_distance[start][end], start)]

while heap:
    num_visited += 1
    # Pop the node with the smallest distance from the heap
    distance, node = heapq.heappop(heap)
    # Subtract the heuristic distance of the current node to the end node from the total distance
    distance -= heuristic_distance[node][end]

    # If the current node is the end node, break the loop
    if node == end:
        break
    # If the current node has no outgoing edges, skip it
    if node not in edges_distance:
        continue

    # For each destination node that the current node has an edge to
    for destination, edge_dist in edges_distance[node].items():
        # Calculate the new distance
        new_dist = distance + edge_dist + heuristic_distance[destination][end]
        # If the destination node is the start node or it has been visited before with a smaller distance, skip it
        if destination == start or (destination in previous_node and previous_node[destination][1] < new_dist):
            continue
        # Push the destination node and its new distance into the heap
        heapq.heappush(heap, (new_dist, destination))
        # Record the current node as the previous node of the destination node
        previous_node[destination] = (node, new_dist)

# Start the path with the end node
path = [end]
# Build the path from the end node to the start node
while path[0] in previous_node:
    # Add the distance from the previous node to the current node to the total distance
    dist += edges_distance[previous_node[path[0]][0]][path[0]]
    # Insert the previous node at the beginning of the path
    path.insert(0, previous_node[path[0]][0])

# Return the path, the total distance, and the number of visited nodes
return path, dist, num_visited
# End your code (Part 4)
```

- **Part 6 (bonus)**

```
# Begin your code (Part 6)
"""
    1.This code implements the A* algorithm to find the shortest path considering time constraints between two nodes in a graph.
    2.It initializes dictionaries to store edge distances, speed limits, and the maximum speed limit.
    3.Then, it reads edge and heuristic data from CSV files and populates the dictionaries.
    4.The A* algorithm is performed using a priority queue to explore neighboring nodes efficiently.
    5.It reconstructs the shortest path considering time and calculates the total time.
    6.Finally, it returns the shortest path, total time, and the number of visited nodes.
"""
# Initialize dictionaries to store edge distances, speed limits, and maximum speed limit
edges_distance = {}
speed_limit = {}
maximum_speed_limit = float('-inf')

# Read data from the CSV files 'edges.csv' and 'heuristic.csv'
with open('edges.csv', newline='') as csv_file:
    reader = csv.reader(csv_file, delimiter=',')
    next(reader)  # Skip header
    for row in reader:
        source, destination, distance, speed = map(float, row)  # Change int to float
        edges_distance.setdefault(int(source), {})[int(destination)] = distance
        speed_limit.setdefault(int(source), {})[int(destination)] = speed / 3.6
        maximum_speed_limit = max(maximum_speed_limit, speed / 3.6)

heuristic_distance = {}
with open('heuristic.csv', newline='') as csv_file:
    reader = csv.reader(csv_file, delimiter=',')
    next(reader)  # Skip header
    for row in reader:
        node, *heuristics = map(float, row)
        heuristic_distance[int(node)] = {
            1079387396: heuristics[0],
            1737223506: heuristics[1],
            8513026827: heuristics[2]
        }
```

```
# Initialize variables for path, time, number of visited nodes, and previous nodes
path = []
time = 0
num_visited = 0
previous_node = {}
# Initialize heap with start node and its heuristic time to the end node
heap = [(heuristic_distance[start][end] / maximum_speed_limit, start)]

while heap:
    num_visited += 1
    # Pop the node with the smallest time from the heap
    t, node = heapq.heappop(heap)
    # Subtract the heuristic time of the current node to the end node from the total time
    t -= heuristic_distance[node][end] / maximum_speed_limit

    # If the current node is the end node, break the loop
    if node == end:
        break
    # If the current node has no outgoing edges, skip it
    if node not in edges_distance:
        continue

    # For each destination node that the current node has an edge to
    for destination, edge_dist in edges_distance[node].items():
        # Calculate the new time
        new_time = t + edge_dist / speed_limit[node][destination] + heuristic_distance[destination][end] / maximum_speed_limit
        # If the destination node is the start node or it has been visited before with a smaller time, skip it
        if destination == start or (destination in previous_node and previous_node[destination][1] < new_time):
            continue
        # Push the destination node and its new time into the heap
        heapq.heappush(heap, (new_time, destination))
        # Record the current node as the previous node of the destination node
        previous_node[destination] = (node, new_time)

# Start the path with the end node
path = [end]
# Build the path from the end node to the start node
while path[0] in previous_node:
    # Add the time from the previous node to the current node to the total time
    time += edges_distance[previous_node[path[0]][0]][path[0]] / speed_limit[previous_node[path[0]][0]][path[0]]
    # Insert the previous node at the beginning of the path
    path.insert(0, previous_node[path[0]][0])

# Return the path, the total time, and the number of visited nodes
return path, time, num_visited
# End your code (Part 6)
```

Part II. Results & Analysis (12%):

- **Test 1:**

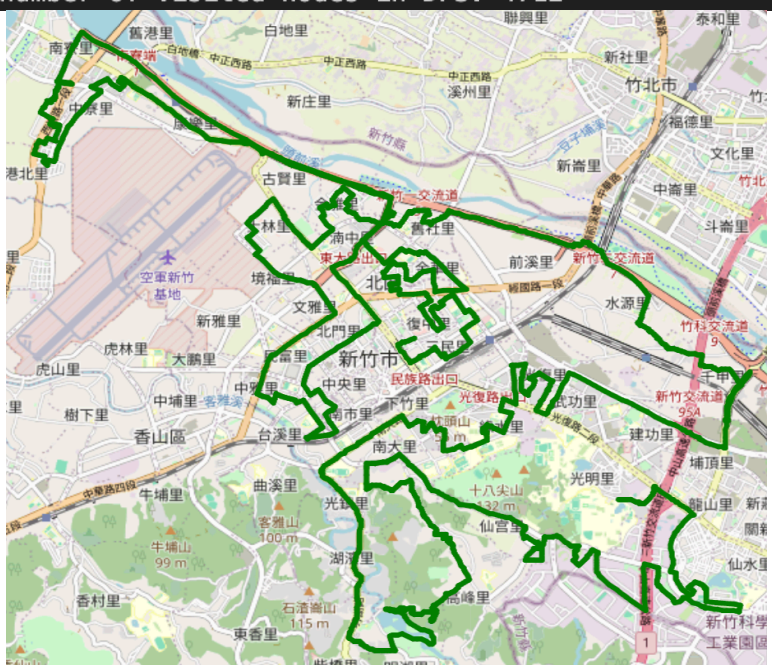
  from National Yang Ming Chiao Tung University (ID: 2270143902) to Big City Shopping Mall (ID: 1079387396)

  BFS:

```
The number of nodes in the path found by BFS: 88
Total distance of path found by BFS: 4978.881999999998 m
The number of visited nodes in BFS: 4274
```

  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: 4712
```



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:**

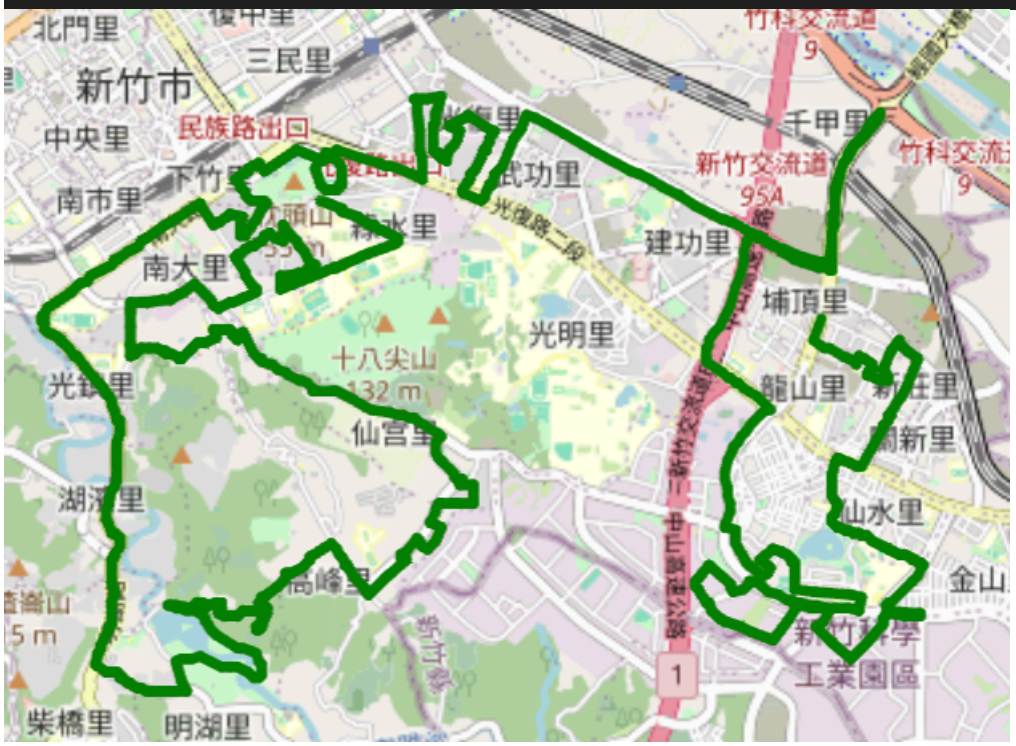
  from Hsinchu Zoo (ID: 426882161)
  to COSTCO Hsinchu Store (ID: 1737223506)

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: 4607
```

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: 9366
```



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: 304.4436634360302 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: 11242
```

  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: 2248
```

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
```

Part III. Answer the questions (12%):

1. Please describe a problem you encountered and how you solved it.
   - While implementing the A* search algorithm, I initially encountered difficulties in designing the heuristic function to ensure admissibility and efficiency.

   - After researching various approaches, I decided to use the straight-line distance between nodes as the heuristic function. This choice ensured admissibility since it never overestimated the actual cost to reach the goal node. Additionally, it improved efficiency by guiding the search towards the goal node more directly.

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.
   - **Road Conditions**: Besides speed limit and distance, considering road conditions such as traffic congestion, road closures, construction sites, and weather conditions is essential for route finding. These factors significantly impact travel time and route feasibility. Incorporating real-time traffic data and historical traffic patterns into the route-finding algorithm can help optimize routes and provide more accurate arrival time estimates.

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

   - **Mapping**: One solution for mapping is leveraging crowdsourced data from platforms like OpenStreetMap or Google Maps. These platforms allow users to contribute and update map data, ensuring that the map is continually updated with accurate information. Additionally, using satellite imagery and geographical data from government agencies can enhance mapping accuracy.

   - **Localization**: For localization, techniques such as GPS-based localization, Wi-Fi positioning, and visual odometry can be employed. GPS provides global positioning information but may suffer from inaccuracies in urban environments or indoor spaces. Wi-Fi positioning utilizes Wi-Fi signals from nearby access points to estimate the device's location. Visual odometry involves using visual cues from cameras to estimate position changes over time. Combining these techniques can improve localization accuracy in various environments.

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.

   - **Dynamic Heuristic Equation**:
     ETA = BaseTime + DeliveryPriorityFactor + MealPrepTimeFactor + MultipleOrdersFactor

     - BaseTime: Represents the base travel time calculated based on distance and speed limit.
     - DeliveryPriorityFactor: Incorporates delivery priority, giving higher priority orders shorter ETAs.
     - MealPrepTimeFactor: Accounts for the time required for meal preparation at the restaurant before delivery.
     - MultipleOrdersFactor: Adjusts ETA based on the number of concurrent orders assigned to the delivery driver. More orders may increase ETA due to additional stops.

   - **Rationale**: This dynamic heuristic equation considers various factors affecting delivery time beyond distance and speed limit. By incorporating delivery priority, meal prep time, and multiple orders, the ETA becomes more accurate and adaptable to changing conditions, enhancing the user experience and satisfaction.