Homework 2: Route Finding

Part I. Implementation

part 1 (Breadth-first Search)

```
def bfs(start, end):
    # Begin your code (Part 1)
    '''
    First of all, I utilize a custom function, read_graph, to parse the edges.csv file and organize the data into a
    dictionary format. Subsequently, I use the deque function to create a queue so as to store the node I traverse.
    Additionally, I establish a set named visited to track whether a node has been traversed.
    Next, I implement breadth-first traversal using a while loop. During each iteration, I pop the front of queue to
    get the current node, and add it to the visited set. If the node I currently traverse is end node, then I return
    the path, distance and number of visited nodes. Otherwise, put the next unvisited nodes, updated path and updated
    distance into the queue. The traversal continues until the queue becomes empty. If the end node is not reached,
    the function returns an empty list, -1, and the count of visited nodes.
    '''
    graph = read_graph(edgeFile)

    queue = deque([(start, [start], 0)])  # Queue stores tuples of (node, path, distance)
    visited = set()
    while queue:
        node, path, dist = queue.popleft()
        visited.add(node)
        if(node == end):
            num_visited = len(visited)-1
            return path, dist, num_visited
        for next, weight, _  in graph.get(node, []):
            #print(f"not visited: {node}")
            #print(next)
            if next not in visited:
                queue.append((next, path + [next], dist+weight))

    # If we do not reach the end node
    num_visited = len(visited)-1
    return [], -1,  num_visited
    # End your code (Part 1)
```

part 2 (Depth-first Search using stack)

```
def dfs(start, end):
    # Begin your code (Part 2)
    '''
    First of all, I utilize a custom function, read_graph, to parse the edges.csv file and organize the data into a
    dictionary format. Subsequently, I use the list to create a stack so as to store the node I traverse.
    Additionally, I establish a set named visited to track whether a node has been traversed.
    Next, I implement depth-first traversal using a while loop. During each iteration, I pop the top of stack to get
    the current node, and add it to the visited set. If the node I currently traverse is end node, then I return the
    path, distance and number of visited nodes. Otherwise, put the next unvisited nodes, updated path and updated
    distance into the stack. The traversal continues until the stack becomes empty. If the end node is not reached,
    the function returns an empty list, -1, and the count of visited nodes.
    '''
    graph = read_graph(edgeFile)

    stack = [(start, [start], 0)] #stores tuples of (node, path, distance)
    visited = set()
    while stack:
        node, path, dist = stack.pop()
        visited.add(node)
        if node==end:
            num_visited = len(visited)-1
            return path, dist, num_visited
        for next, weight, _  in graph.get(node, []):
            if next not in visited:
                stack.append((next, path + [next], dist+weight))

    # If we do not reach the end node
    num_visited = len(visited)-1
    return [], -1,  num_visited
    # End your code (Part 2)
```

part 3 (Uniform Cost Search)

```
def ucs(start, end):
    # Begin your code (Part 3)
    '''
    First of all, I utilize a custom function, read_graph, to parse the edges.csv file and organize the data into a
    dictionary format. Subsequently, I use the list to create a priority queue so as to store the node I traverse.
    Additionally, I establish a set named visited to track whether a node has been traversed.
    Next, I implement uniform cost search using a while loop. During each iteration, I use heappop to extract the
    node with the highest priority from the priority queue, which maintains the priority queue's property.Then, add
    the node to the visited set. If the node I currently traverse is end node, then I return the path, distance and
    number of visited nodes. Otherwise, I use heappush to enqueue the next unvisited nodes, along with updated path
    and distance information, into the priority queue. The traversal continues until the priority queue becomes empty.
    If the end node is not reached, the function returns an empty list, -1, and the count of visited nodes.
    '''
    graph = read_graph(edgeFile)

    priority_queue = [(0, start, [start])]  # Queue stores tuples of (distance, node, path)
    visited = set()

    while priority_queue:
        dist, node, path = heapq.heappop(priority_queue)
        visited.add(node)
        if(node == end):
            num_visited = len(visited)-1
            return path, dist, num_visited

        for next, weight, _  in graph.get(node, []):
            if next not in visited:
                heapq.heappush(priority_queue, (dist + weight, next, path + [next]))

    # If we do not reach the end node
    num_visited = len(visited)-1
    return [], -1,  num_visited
    # End your code (Part 3)
```

part4 (A* Search)

```python
def astar(start, end):
    # Begin your code (Part 4)
    '''
    First of all, I utilize two custom functions. One of the functions is read_graph, to parse the edges.csv file
    and organize the data into a dictionary format (each dictionary includes one list). The other is read_heuristics,
    to parse the heuristicFile.csv and organize the data into a dictionary format (each dictionary includes another
    dictionary). Subsequently, I use the list to create a priority queue so as to store the node I traverse.
    Additionally, I establish a set named visited to track whether a node has been traversed.
    Next, I implement A* search using a while loop. During each iteration, I use heappop to extract the node with the
    highest priority from the priority queue, which maintains the priority queue's property, and add it to the visited
    set. If the node I currently traverse is end node, then I return the path, distance and number of visited nodes.
    Otherwise, put the evaluated value (the sum of the movement cost from starting node to current node and the estimated
    movement cost from current node to end node), updated distance, next unvisited nodes, and updated path into the
    priority queue. The traversal continues until the priority queue becomes empty. If the end node is not reached, the
    function returns an empty list, -1, and the count of visited nodes.
    '''
    graph = read_graph(edgeFile)
    heuristic = read_heuristics(heuristicFile)

    priority_queue = [(heuristic[start][str(end)], 0, start, [start])]  # (sum of the actual cost (dist) and the heuristic estimate
                                                                        #    , total distance, current node, path)
    visited = set()

    while priority_queue:
        _, dist, node, path = heapq.heappop(priority_queue)
        visited.add(node)
        if node==end:
            num_visited = len(visited)-1
            return path, dist, num_visited

        for next, weight, _ in graph.get(node, []):
            if next not in visited:
                updated_dist = dist + weight
                heapq.heappush(priority_queue, (updated_dist + int(heuristic[next][str(end)]), updated_dist, next, path+[next]))

    # If we do not reach the end node
    num_visited = len(visited)-1
    return [], -1,  num_visited
    # End your code (Part 4)
```

part6 (A*_time Search)

```python
def astar_time(start, end):
    # Begin your code (Part 6)
    '''
    First of all, I utilize two custom functions. One of the functions is read_graph, to parse the edges.csv file
    and organize the data into a dictionary format (each dictionary includes one list). The other is read_heuristics,
    to parse the heuristicFile.csv and organize the data into a dictionary format (each dictionary includes another
    dictionary). Subsequently, I use the list to create a priority queue so as to store the node I traverse.
    Additionally, I establish a set named visited to track whether a node has been traversed. Note that we should perform
    unit conversion before using the limit of the speed.
    Next, I implement A*_time search using a while loop. During each iteration, I use heappop to extract the node with the
    highest priority from the priority queue, which maintains the priority queue's property, and add it to the visited
    set. If the node I currently traverse is end node, then I return the path, time and number of visited nodes.
    Otherwise, put the evaluated value (the sum of the movement time from starting node to current node and the estimated
    movement cost divides the maximum speed limit from current node to end node), updated time, next unvisited nodes, and
    updated path into the priority queue. The traversal continues until the priority queue becomes empty. If the end node
    is not reached, the function returns an empty list, -1, and the count of visited nodes.
    '''
    graph, max_speed_limit = read_graph(edgeFile)
    heuristic = read_heuristics(heuristicFile)
    max_speed_limit = max_speed_limit*1000/3600

    # (sum of the actual time and the heuristic estimate (time)
    #   , total time, current node, path)
    priority_queue = [(heuristic[start][str(end)]/max_speed_limit, 0, start, [start])]
    visited = set()

    while priority_queue:
        _, time, node, path = heapq.heappop(priority_queue)
        if node in visited:
            continue

        visited.add(node)
        if node==end:
            num_visited = len(visited)-1
            return path, time, num_visited

        for next, weight, speed_limit in graph.get(node, []):
            if next not in visited:
                speed_limit = speed_limit * 1000 / 3600
                updated_time = time + weight / speed_limit
                heapq.heappush(priority_queue, (updated_time + float(heuristic[next][str(end)]) / max_speed_limit, updated_time, next, path+[next]))

    # If we do not reach the end node
    num_visited = len(visited)-1
    return [], -1,  num_visited
    # End your code (Part 6)
```

Part II. Results & Analysis

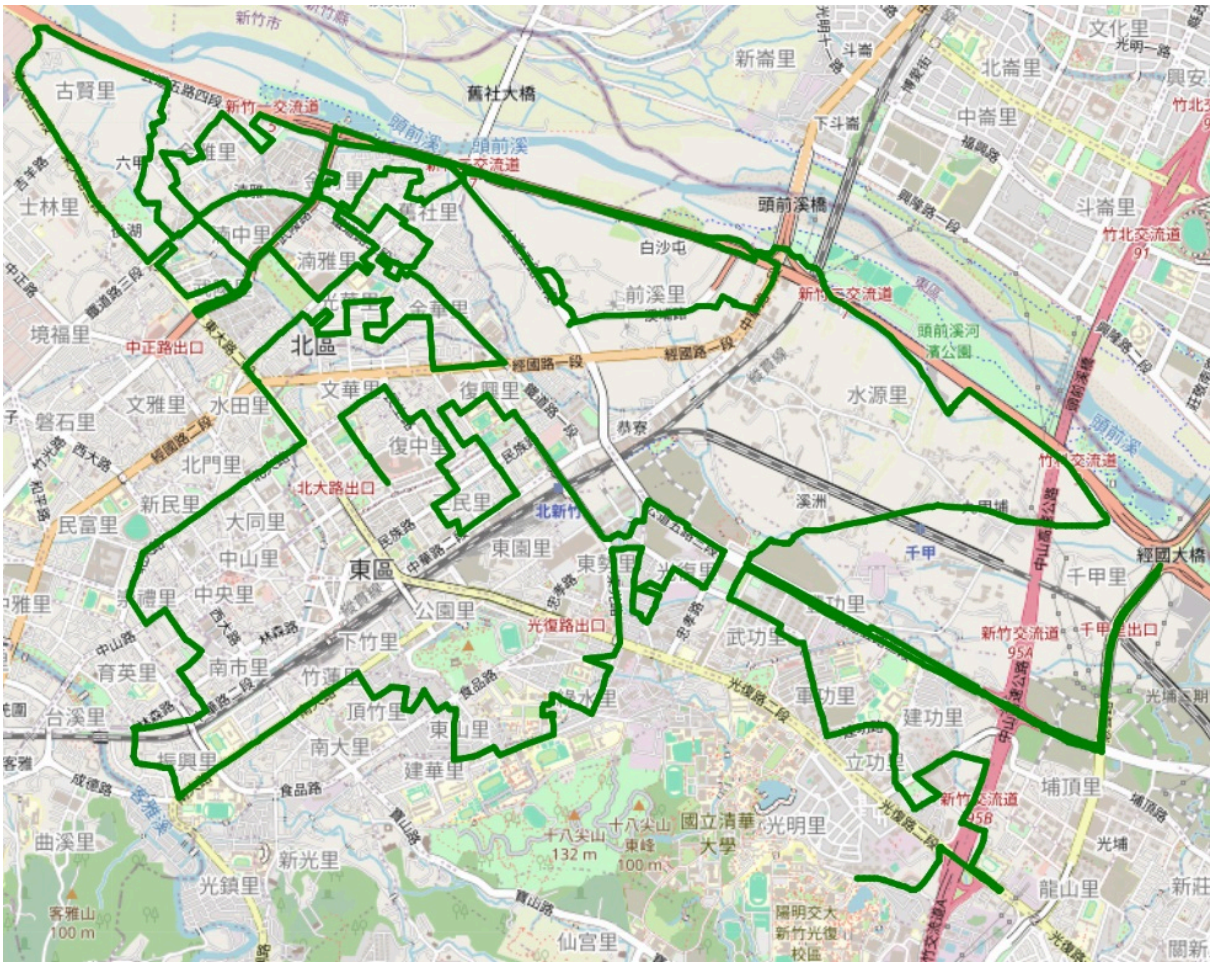
- Test1: from National Yang Ming Chiao Tung University (ID: 2270143902) to Big City Shopping Mall (ID: 1079387396)

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: 4273
```

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: 4210
```



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: 5085
```

A*:

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

comparison:

	BFS	DFS	UCS	A*
number of nodes in the path	88	1232	89	89
total distance of the path	4978.88	57208.99	4367.88	4367.88
number of visited nodes	4273	4210	5085	261

➞ We can observe that the path DFS finds is the worst path, since the number of nodes and the total distance is the highest. Although the numbers of the nodes in path BFS finds are the fewest, the distance of the path is not optimal. UCS and A* algorithm find the same best path. However, we can find that the numbers of nodes A* algorithm needs to search are much fewer than the ones of UCS.

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: 1933
```

➞ The path I find in A* algorithm (based on time) is the same path as I find in A* algorithm (based on distance). However, the numbers of visited nodes in the A* algorithm (based on time) are much more than ones in A* algorithm (based on distance)

- Test2: from HsinchuZoo (ID:426882161) to COSTCO HsinchuStore (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: 4606
```

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: 8030
```

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: 7212
```

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: 1172
```

comparison:

	BFS	DFS	UCS	A*
number of nodes in the path	60	998	63	63
total distance of the path	4215.52	41094.66	4101.84	4101.84
number of visited nodes	4606	8030	7212	1172

➞ We can observe that the path DFS finds is the worst path, since the number of nodes and the total distance is the highest. Although the numbers of the nodes in path BFS finds are the fewest, the distance of the path is not optimal. UCS and

A* algorithm find the same best path. However, we can find that the numbers of nodes A* algorithm needs to search are much fewer than the ones of UCS.

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: 2869
```

➞ The path I find in A* algorithm (based on time) is the same path as I find in A* algorithm (based on distance). However, the numbers of visited nodes in the A* algorithm (based on time) are much more than ones in A* algorithm (based on distance)

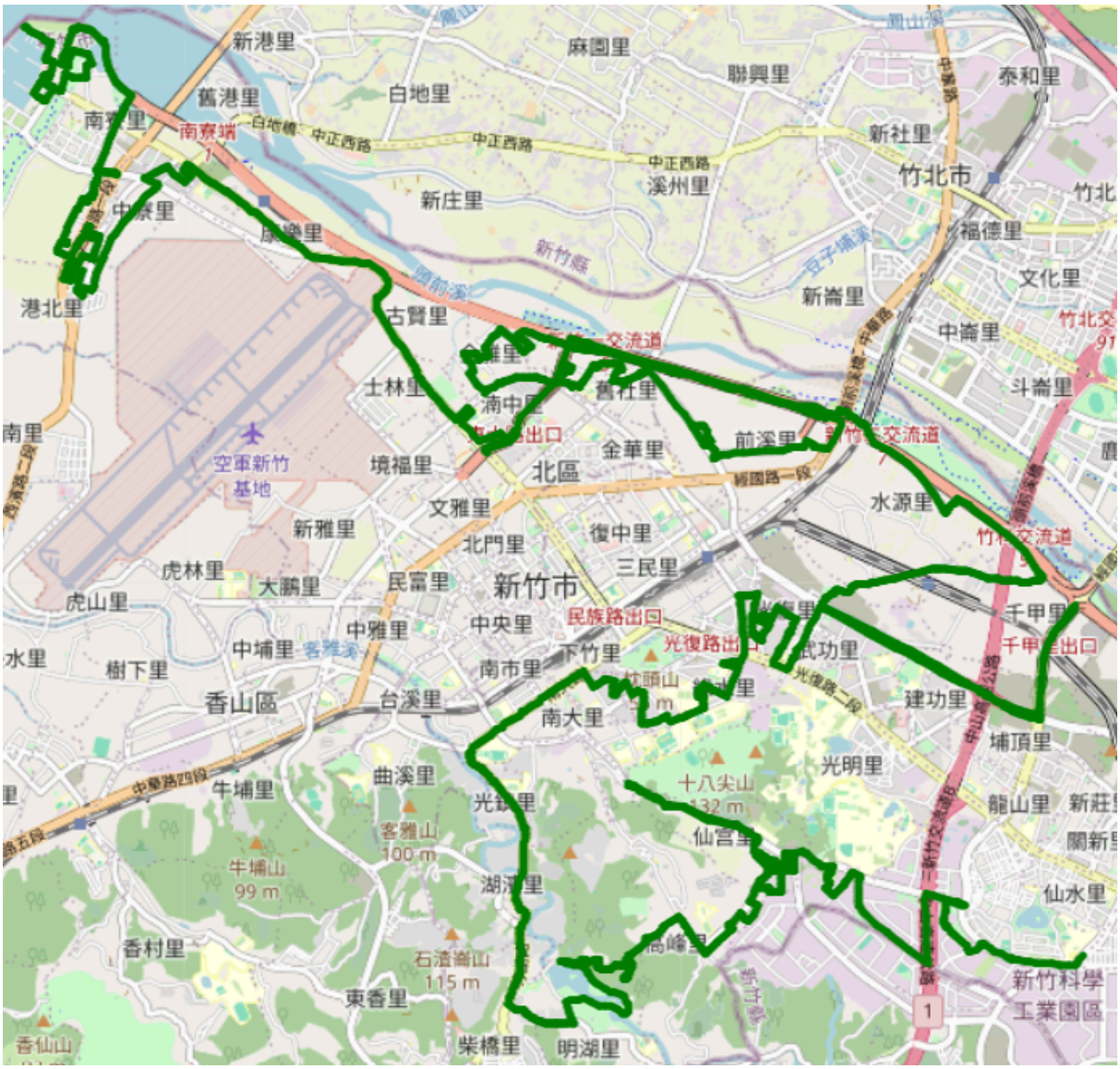
- Test 3：from National Experimental High School At Hsinchu Science Park (ID: 1718165260)to Nanliao Fighing Port (ID: 8513026827)

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: 11241
```

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: 3291
```



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: 11925
```

A*:

```
The number of nodes in the path found by A* search: 288
Total distance of path found by A* search: 14212.412999999997 m
The number of visited nodes in A* search: 7074
```

comparison:

	BFS	DFS	UCS	A*
number of nodes in the path	183	1521	288	288
total distance of the path	15442.39	64821.6	14212.41	14212.41
number of visited nodes	11241	3291	11925	7074

➞ We can observe that the path DFS finds is the worst path, since the number of nodes and the total distance is the highest. Although the numbers of the nodes in path BFS finds are the fewest, the distance of the path is not optimal. UCS and A* algorithm find the same best path. However, we can find that the numbers of nodes A* algorithm needs to search are much fewer than the ones of UCS.

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: 8457
```

➞ The path I find in A* algorithm (based on time) is not the same path as I find in A* algorithm (based on distance). The reason why this path takes less time is that most parts of the path are on the Hsinchu Interchange, where the limit of the speed is higher. Moreover, we can observe the numbers of visited nodes in the A* algorithm (based on time) are much more than ones in A* algorithm (based on distance)

Part III. Question Answering

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

In this homework, I encountered two problems.
First of all, when I was a freshman, I used the C++ to learn the data structure. It contributes to that I know what data structure I need to use in different algorithm, but I do not know how to call it in python. Therefore, I have conducted some online

research. The following is the result of my research:
(1)
When I need to use a map to store the imformation, I will use " map<string, vector<string>>graph " in C++. In python, I found that I can achieve the same functionality by using the "defaultdict" function to initialize the dictionary.
(2)
If I want to perform BFS, it is neccessary to create a queue. In C++, I utilize "queue" to create one and perform some relative operation. In python, there is a little bit difference. I found that I can utilize "deque" function to achieve the same functionality.
(3)
If I want to perform DFS in stack method,it is neccessary to create a stack. In C++, I utilize "stack" to create one and perform some relative operation. In python, there is a little bit difference. I found that I can just utilize "list" to achieve the same functionality.
(4)
In Uniform Cost Search and A star algorithm, I need to use the priority queue. In C++, I use "priority_queue" to create one and perform some relative operation. In Python, I just use "list" to store the data, but use operations in "heapq" library, which can maintain the property of priority queue.

Secondly, I encountered a problem that the path of BFS is not the same as I expected. After the long time of debugging, I discovered that the problem is from this two line:

```
path.append(next)
queue.append((next, path, dist+weight))
```

If I modify the data in path list, the other path lists in queue will be modified, too. Therefore, if I do not want to change the other path lists but still want to add a new element in new path. I need to modify my code:

```
queue.append((next, path + [next], dist+weight))
```

In this way, the algorithm can be performed correctly.
refrence: https://medium.com/starbugs/python-一次搞懂-pass-by-value-pass-by-reference-與-pass-by-sharing-1873a2c6ac46 (https://medium.com/starbugs/python-%E4%B8%80%E6%AC%A1%E6%90%9E%E6%87%82-pass-by-value-pass-by-reference-%E8%88%87-pass-by-sharing-1873a2c6ac46)

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.

In my opinion, "traffic flow" can be another essential attribute for route finding in the real world. For example, heavy traffic can lead to congestion during certain times, such as rush hours around the companies, as seen on Guangfu Road near NYCU during peak hours. In this condition, even though we have routes calculated based on

speed limits and distances, significant delays may still occur due to high traffic volume. Therefore, it is essential for us to take traffic flow into account to predict the most suitable route.

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:
(1) The easiest one is that we can just utilize the pre-existing maps, such as Google maps.
(2) Utilize crowdsourced mapping. Allowing users to contribute to map updates and corrections through feedback mechanisms.
(3) Utilize SLAM (Simultaneous Localization and Mapping). We can use some sensors, such as LiDAR, cameras, or radar, to create maps in real-time. (This method can also determine the position of the sensor within the map.)

Localization:
(1) We can utilize GPS(Global Positioning System), which can determine the device's position on the Earth's surface.
(2) We can use computer vision techniques to analyze real-time camera images and try to match them with pre-existing landmarks in the map.
(3) We can utilize beacons to localize the position. This approach involves placing beacons (e.g., RFID tags, Bluetooth beacons, or visual markers) at known locations in the environment and using them as reference points for localization.

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

From my perspective, I design a simplified heuristic equation along with the rationale behind each component:
ETA = Meal Prep Time × Orders Factor + Delivery Time × Delivery Factor
where,
(1) **Meal Prep Time**: The normal time required to prepare the food. It can be estimated based on historical data, menu complexity, and the restaurant's reputation for efficiency.
(2) **Orders Factor**: In some specific time, the number of customers in restaurants will be skyrocketing. Therefore, we should take it account to calculate the time. Moreover, the "Orders Factor" component also includes situational events, such as instances of being short-staffed.

(3) **Delivery Time**: The normal time required for delivers to transport the order from the restaurant to the customer's location. It can be estimated based on factors such as distance and speed limit of the road.
(4) **Delivery Factor**: Numerous factors can affect the delivery time, such as weather, traffic flow and delivery priority. With an eye to predicting accurate ETA, we should take it account to adjust the total delivery time it should take.