Homework 2: Route Finding

Report Template

Part I. Implementation (6%):

openFile function

```
4   def openFile():
5       graph = {} # start : [end1, end2, ...]
6       distance = {} # (start, end) : dist
7
8       with open(edgeFile, newline='') as csvfile:
9           rows = csv.DictReader(csvfile)
10          for row in rows:
11              s = int(row['start'])
12              e = int(row['end'])
13              d = float(row['distance'])
14              if s not in graph: # create a list for s if it has not been seen yet
15                  graph[s]=[]
16              graph[s].append(e) # add the edge
17              distance[s, e] = d # add the distance of the edge
18      return graph, distance
```

findPath function

```
20  def findPath(end, parent):
21      path = []
22      current = end
23      while current != -1: # parent of start = -1
24          path.append(current) # append the current node to path
25          current = parent[current] # and move onto its parent
26      path.reverse() # reverse the order so it's the correct way
27      return path
```

findDistance funciton

```
29  def findDistance(path, distance):
30      dist = 0
31      for i in range(len(path)-1):
32          node = path[i]
33          next_node = path[i+1]
34          if(distance.get(node, next_node)): # if there exists a path
35              dist += distance[node, next_node] # then add on to the distance
36      return dist
```

Part.1 BFS

```python
def bfs(start, end):
    # Begin your code (Part 1)
    graph, distance = openFile()
    visited = set()
    visited.add(start)
    queue = [start]
    parent = {start: -1}

    while queue:
        node = queue.pop(0) # pop first element (first in first out)
        if node == end: # if reached the destination
            path = findPath(end, parent) # find the corresponding path using parent-child relationship
            dist = findDistance(path, distance) # calculate the distance between each node
            return path, dist, len(visited)

        for adj in graph.get(node, []): # if not reached the destination
            if adj not in visited: # make sure the adjacent node has not been visited
                queue.append(adj) # put it at the end of queue
                visited.add(adj) # mark it as visited
                parent[adj] = node # remember its parent-child relationship

    return [], 0, len(visited) # return these if no route is found
    # End your code (Part 1)
```

Part.2 DFS

```python
def dfs(start, end):
    # Begin your code (Part 2)
    graph, distance = openFile()
    visited = set()
    visited.add(start)
    stack = [start]
    parent = {start: -1}

    while stack:
        node = stack.pop() # pop last element (first in last out)
        if node == end: # if reached the destination
            path = findPath(end, parent) # find the corresponding path using parent-child relationship
            dist = findDistance(path, distance) # calculate the distance between each node
            return path, dist, len(visited)

        for adj in graph.get(node, []): # if not reached the destination
            if adj not in visited: # make sure the adjacent node has not been visited
                stack.append(adj) # put it at the top of stack
                visited.add(adj) # mark the node as visited
                parent[adj] = node # remember its parent-child relationship

    return [], 0, len(visited) # return these if no route is found
    # End your code (Part 2)
```

Part.3 UCS

```python
def ucs(start, end):
    # Begin your code (Part 3)
    graph, distance = openFile()
    visited = set()
    visited.add(start)
    p_queue = [(0, [start])] # priority queue (cumulated distance, path)

    while p_queue:
        d, path = p_queue.pop(0) # pop the first item of priority queue (shortest cumulated distance)
        node = path[-1] # mark current node as the last node of this path
        visited.add(node) # mark this node as visited

        if node == end: # if reached the destination
            dist = findDistance(path, distance) # calculate the distance between each node
            return path, dist, len(visited)

        for adj in graph.get(node, []): # if not reached the destination
            if adj not in visited: # make sure the adjacent node has not been visited
                new_path = list(path) # create new path
                new_path.append(adj) # add the adjacent node to its end
                p_queue.append((d + distance[node, adj], new_path)) # put it on pq with updated distance
                p_queue.sort() # sort it by distance so it is prioritized

    return [], 0, len(visited) # return these if no route is found
    # End your code (Part 3)
```

Part.4 A*

openFile function of A* (includes heuristic file)

```python
def openFile(end):
    graph = {} # start : [end1, end2, ...]
    distance = {} # (start, end) : dist
    h1 = {}
    h2 = {}
    h3 = {}
    with open(edgeFile, newline='') as csvfile:
        rows = csv.DictReader(csvfile)
        for row in rows:
            s = int(row['start'])
            e = int(row['end'])
            d = float(row['distance'])
            if s not in graph: # create a list for s if it has not been seen yet
                graph[s]=[]
            graph[s].append(e) # add the edge
            distance[s, e] = d # add the distance of the edge
    with open(heuristicFile, newline='') as csvfile: # heuristic numbers
        rows = csv.DictReader(csvfile)
        for row in rows:
            n = int(row['node'])
            d1 = float(row['1079387396'])
            d2 = float(row['1737223506'])
            d3 = float(row['8513026827'])
            h1[n] = d1
            h2[n] = d2
            h3[n] = d3
    if end == 1079387396: h = h1
    elif end == 1737223506: h = h2
    elif end == 8513026827: h = h3
    return graph, distance, h
```

Support function for sorting queue.

```
5   def myFunc(e):
6       return e['g'] + e['h']
```

Astar function

```
57  def astar(start, end):
58      # Begin your code (Part 4)
59      graph, distance, h = openFile(end)
60      p_queue = [{'node': start, 'g': 0, 'h': 0}] # priority queue (node, g, h)
61      closed_list = []
62      parent = {start: -1} # parent of start is -1
63      num_visited = 0 # remember the number of visited nodes
64
65      while p_queue:
66          l = p_queue.pop(0) # pop off first item of priority queue
67          node = l['node']
68          g_node = l['g']
69
70          if node == end:
71              if node == end: # if reached the destination
72                  path = findPath(end, parent) # find the corresponding path using parent-child relationship
73                  dist = findDistance(path, distance) # calculate the distance between each node
74                  return path, dist, num_visited
75
76          for adj in graph.get(node, []): # if not reached the destination
77              num_visited += 1 # update visited number
78              g_adj = g_node + distance[node, adj] # update distance of adjacent node
79              h_adj = h[adj] # find the heuristic number of adj
80              found = False
81              el = 0
```

```
82              for element in p_queue:
83                  if element['node'] == adj:
84                      el = element
85              if el: # if found element in pq
86                  if(g_adj + h_adj > el['g'] + el['h']):
87                      # if the f of the found element is less than the one found here
88                      # then don't update the node
89                      found = True
90              for element in closed_list:
91                  if element['node'] == adj:
92                      el = element
93              if el: # if found element in the closed list
94                  if(g_adj + h_adj > el['g'] + el['h']):
95                      # if the f of the found element is less than the one found here
96                      # then don't update the node
97                      found = True
98              if not found: # if no node is found in the lists or they're greater in distance
99                  p_queue.append({'node': adj, 'g': g_adj, 'h': h_adj}) # add new node to pq
100                 p_queue.sort(key=myFunc) # sort pq by f
101                 parent[adj] = node # remember its parent-child relationship
102
103         closed_list.append(l) # append node to closed list
104
105     return [], 0, num_visited # return these if no route is found
106     # End your code (Part 4)
```

Part.6 Bonus A* time

openFile function of A* time (includes time dictionary and updated heuristic num)

```
def openFile(end):
    graph = {} # start : [end1, end2, ...]
    distance = {} # (start, end) : dist
    time = {}
    h1 = {}
    h2 = {}
    h3 = {}
    sl = 80 # estimated speed limit
    with open(edgeFile, newline='') as csvfile:
        rows = csv.DictReader(csvfile)
        for row in rows:
            s = int(row['start'])
            e = int(row['end'])
            d = float(row['distance'])
            sp = float(row['speed limit'])
            if s not in graph:
                graph[s]=[] # create a list for s if it has not been seen yet
            graph[s].append(e) # add the edge
            distance[s, e] = d # add the distance of the edge
            time[s, e] = (d/sp)*3.6 # add the time of travel of the edge
    with open(heuristicFile, newline='') as csvfile: # heuristic numbers
        rows = csv.DictReader(csvfile)
        for row in rows:
            n = int(row['node'])
            d1 = float(row['1079387396'])
            d2 = float(row['1737223506'])
            d3 = float(row['8513026827'])
            h1[n] = d1/(sl/3.6)
            h2[n] = d2/(sl/3.6)
            h3[n] = d3/(sl/3.6)
    if end == 1079387396: h = h1
    elif end == 1737223506: h = h2
    elif end == 8513026827: h = h3
    return graph, distance, h, time
```

Support function for sorting queue.

```
def myFunc(e):
    return e['g'] + e['h']
```

findTime function (same as findDistance but with time)

```
def findTime(path, time):
    t = 0
    for i in range(len(path)-1):
        node = path[i]
        next_node = path[i+1]
        if(time.get(node, next_node)): # if there exists a path
            t += time[node, next_node] # then add on to the distance
    return t
```

A* time function (mostly the same as A* function but with findTime)

```
61  def astar_time(start, end):
62      # Begin your code (Part 6)
63      graph, distance, h, time = openFile(end)
64      p_queue = [{'node': start, 'g': 0, 'h': 0}]
65      closed_list = []
66      parent = {}
67      parent[start] = -1
68      num_visited = 0
69
70      while p_queue:
71          l = p_queue.pop(0)
72          node = l['node']
73          g_node = l['g']
74
75          if node == end: # if reached the destination
76              path = findPath(end, parent) # find the corresponding path using parent-child relationship
77              dist = findTime(path, time) # calculate the time between each node
78              return path, dist, num_visited
79
80          for adj in graph.get(node, []): # if not reached the destination
81              num_visited += 1 # update visted number
82              g_adj = g_node + time[(node, adj)] # update distance of adjacent node
83              h_adj = h[adj] # find the heuristic number of adj
84              el = 0
85              found = False
86              for element in p_queue:
87                  if element['node'] == adj:
88                      el = element
89              if el: # if found element in pq
90                  if(g_adj + h_adj > el['g'] + el['h']):
91                      # if the f of the found element is less than the one found here
92                      # then don't update the node
93                      found = True
94              for element in closed_list:
95                  if element['node'] == adj:
96                      el = element
97              if el: # if found element in closed list
98                  if(g_adj + h_adj > el['g'] + el['h']):
99                      # if the f of the found element is less than the one found here
100                     # then don't update the node
101                     found = True
102             if not found: # if no node is found in the lists or they're greater in distance
103                 p_queue.append({'node': adj, 'g': g_adj, 'h': h_adj}) # add new node to pq
104                 p_queue.sort(key=myFunc) # sort pq by f
105                 parent[adj] = node # remember its parent-child relationship
106
107         closed_list.append(l) # append node to closed list
108
109     return [], 0, num_visited # return these if no route is found
110     # 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:

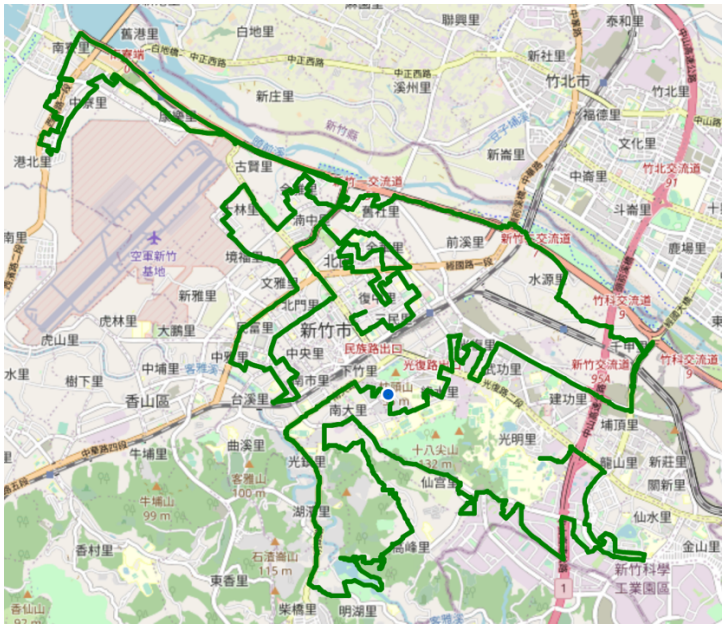
BFS is supposed to give out the route with the least nodes, however, there is actually routes with less nodes than 88, so it took me a few tries to get the right algorithm to have the same result as in the 測試結果.pdf.

```
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:

DFS just finds a route and continues until it reaches the end. So the result isn’ t too surprising when the distance from school to Big City is 75 km.

```
The number of nodes in the path found by DFS: 1718
Total distance of path found by DFS: 75504.31499999983 m
The number of visited nodes in DFS: 5236
```



UCS:

UCS finds the shortest path with multiple trials and erros, so the number of visited n odes is quite a bit compared to that of A*.

```
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
```

A*:

A star finds the shortest path using a smart algorithm by predicting and guessing the best route at first. So the number of visited nodes is quite low compared to that of th e UCS algorithm.

```
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: 531
```

A* time:

The number of visited nodes become a lot bigger than that of A star distance even th ough this is the same route. And I never knew you could get to big City in just unde r 6 minutes if there is no traffic lights!

```
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: 3140
```

Test.2

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

BFS:

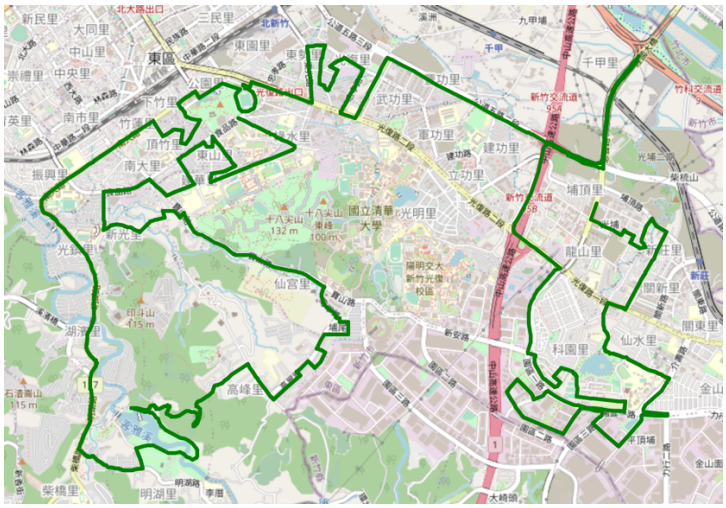
This is actually a pretty good match to the shortest route considering how easy this a lgorithm is to implement.

```
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:

DFS almost visited the entire Hsinchu city before reaching the final destination.

```
The number of nodes in the path found by DFS: 930
Total distance of path found by DFS: 38752.30799999996 m
The number of visited nodes in DFS: 9616
```



UCS:

At first, my UCS algorithm cannot choose the right path on the upper left coner, it w ill always choose the one the takes a longer path, so the result is off by just 60 meter s. It was really frustrating, but I solved this problem by saving the list of path each ti

me after inserting a new adjacent node instead of remebering its parent-child relation ship like I did in the other algorithms. I’ m not sure why it resolved the problem, b ut I’ m glad it did.

```
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
```

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: 2630
```

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: 4629
```

Test.3

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

BFS:

This is actually very similar to the route of A star time algorithm. It takes the highw ay instead of the downtown area. This route has the least number of nodes.

```
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:

DFS actually did not bad in this test, it followed a pretty accurate route.

```
The number of nodes in the path found by DFS: 900
Total distance of path found by DFS: 39219.99299999996 m
The number of visited nodes in DFS: 2494
```

UCS:

Surprisingly, UCS had less visited nodes than that of A star. Perhaps it’ s because t his route is quite long and there are too many choices for A star to choose from, hen ce the large number of visited nodes needed.

```
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
```

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: 15860
```

A* time:

At first I didn’ t adjust the heuristic function of A star time, I just uesd the given on e and it yielded the result of about 1100 seconds. Later, I changed the heuristic funct ion to all be 0, so the algorithm will visit almost all the nodes and pick the best one. This way isn’ t smart and it ignores the smart characteristic of the A star algorithm. In the end, I adjusted the heuristic function to be the original h(x)/(81.2/3.6) which i s about the average of the highest speed limit of each route in m/s unit. I tried the nu mber from 65~100 and 81.2 yielded the best result in the number of visited nodes.

```
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: 10254
```

Part III. Question Answering (12%):

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

   My Jupyter notebook cannot find the folium module whatsoever, no matter what command s I typed or where I typed my commands. I asked the TA for the solution, but it seems like I can only solve this problem by setting up a virtual environment or use Google Colab. At f irst, I tried to set up a virtual environment, but it was quite far away from my comfort zon e, so I couldn't make it work. Thus, I switched to Google Colab. Thank goodness it work ed there. However, I still didn't quite get how to let Google Colab access my local folder s, so I had to manually add cells of code of the searches to run them on a map. In the end, I had successfully finished my code in Google Colab and have the result show on a folium m ap!

2. Besides speed limit and distance, could you please come up with another attribute that is es sential for route finding in the real world? Please explain the rationale.

   Another essential attribute for route finding in the real world is Real-time updates. It refers to the ability to receive and incorporate live information about road conditions, traffic cong estion, and other disruptions into the navigation system. This allows the route-finding syste m to dynamically adjust the suggested route based on the current situation on the road. So me examples are traffic congestion, road closures, inclement weather, events or emergencie s, points of interest, etc. Google Map also uses real-time updates when suggesting the best route. It tells us the traffic level and whether there are constructions in action.

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: traditional maps, geospatial data APIs, sensor-based mapping (SLAM). Tradition al maps, whether in physical or digital form, can serve as a mapping component in a navig ation system. Geospatial data APIs provided by mapping service providers (Google Maps, OpenStreeMap, …) offer programmable interfaces to access map data. Sensor-based mapp ing, where data from various onboard sensors, such as GPS, accelerometers, gyroscopes, a nd cameras, are collected to create maps.

   Localization: GPS, Wi-Fi or Bluetooth beacons, visual localization, etc. GPS receivers can provide accurate positioning information using signals from GPS satellites, allowing the sy stem to determine the current location of the user. Wi-Fi or Bluetooth beacons placed in kn own locations can be used for localization. These beacons emit signals that can be detected by the user's device, allowing the system to determine the proximity to these known loca tions and estimate the current location based on the received signals. Visual localization in

volves using visual cues, such as landmarks, building facades, or road signs, for localizatio n. Images captured by the device’ s camera can be compared to a pre-built database of vis ual features or reference images to estimate the current location.

4. The estimated time of arrival (ETA) is one of the features of Uber Eats. To provide accurat e estimates for users, Uber Eats needs to dynamically update ETA based on their mechanis m. Please define a dynamic heuristic equation for ETA and explain the rationale of your de sign. Hint: You can consider meal prep time, delivery priority, multiple orders, etc.

   Taken the hint, I will design the heuristic equation for ETA as:
   ETA = (meal prep time + delivery priority) * multiple orders

   Meal prep time: This factor considers the time required for meal preparation at the restaura nt. It includes the time taken to cook, package, and prepare the meal for delivery. This can vary depending on the complexity of the meal, restaurant’ s kitchen capacity, and staff av ailability. This factor should be provided by the restaurant.

   Delivery priority: This factor considers the priority level of the order for delivery. For exa mple, if the order is marked as urgent or has a higher priority due to some special circumst ances, it may receive faster processing and delivery compared to regular orders. This facto r can be assigned based on the level of customer consumption (regular, bronze, silver, gol d, premium…). Perhaps we can mark regular as 35 min, bronze as 30 min, silver as 25 mi n, gold as 20 min, and premium as 15 min.

   Multiple orders: This factor accounts for the presence of multiple orders in the delivery qu eue. If a delivery driver has multiple orders to deliver in a single trip, the ETA can be adju sted accordingly. This can be done by optimizing the delivery route to minimize detours an d maximize efficiency in delivering multiple orders together, thus reducing the overall ET A. Perhaps we can mark a good route to be 0.8, a so-so route to be 1, and a bad route to be 1.2.

   The exact values and weights for these factors can be fine-tuned based on real-time data an d feedback from the delivery process to continuously improve the ETA estimation accurac y.