Homework 2: Route Finding

Part I. Implementation (6%):

BFS

```python
def bfs(start, end):
    # Begin your code (Part 1)
    '''
    First, use csv to read egeFile. I used a list of dictionaries to save the
    information of every edges In the next for loop, I used a dictionary of
    list to create a adjacency list for every vertice. The key of the
    dictionary is the starting vertice, and the value is a list that stores its
    neighboring vertices.
     '''
    f = open(edgeFile)
    reader = csv.DictReader(f)
    edges = [] #list of dicts [{}, {}...]
    for r in reader:
        edges.append(r)

    adj = {} #dict of lists{[], []...}
    for i in range(len(edges)):
        cur = edges[i]['start']
        cur = int(cur)
        dest = edges[i]['end']
        dest = int(dest)
        if cur not in adj:
            adj[cur] = []
            adj[cur].append(dest)
        else:
            adj[cur].append(dest)
```

```python
    Second, implement BFS algorithm to find the route, which uses a queue to
    determine which vertice is going to process. In every loop, we need to
    process all of the neighboring vertices. Also, I use a list called
    'visited' to see if a vertice is visited or not. If it hasn't been visited,
    append the current vertice into to queue and mark it as visited. On the
    other hand, a dictionary called 'parents' is also very important, because
    it tells us which vertice is the current vertice's predecessor. Finally,
    when the algorithm reach the ending point, I clear up the queue and finish
    the BFS searching.
    '''
    queue = []
    queue.append(start)
    visited = []
    visited.append(start)
    parents = {}
    # parents[start] = -1
    while (len(queue)):
        s = queue.pop(0)
        for i in range(len(adj[s])):
            cur = adj[s][i]
            if cur not in adj:
                visited.append(cur)
                continue
            elif cur not in visited:
                queue.append(cur)
                visited.append(cur)
                parents[cur] = s
                if cur == end:
                    queue.clear()
                    parents[end] = s
                    break
```

```
    '''
    Last, we need to calculate path, distance and the number of visited
    vertices. Since I used the length of the list 'visited' to calculate the
    visited vertices, I need to minus its length by one since the list contains
    the starting point, which should not be counted in the answer. Next, I used
    the dictionary 'parents' to find the path by going to the vertice's
    predecessor one by one, and append them into the list called 'path'.
    At the same time, I sum up their distance to get the total distance.
    After traversing from the ending point to the starting point, I reverse the
    list 'path' and return the assigned data: path, dist, num_visited.
    '''
    path = [end]
    dist = 0
    num_visited = len(visited) - 1 # start doesn't count
    parent = end
    while(parent != start):
        cur = parent
        parent = parents[cur]
        path.append(parent)
        for e in edges:
            s = str(parent)
            dest = str(cur)
            if e['start'] == s and e['end'] == dest:
                dist += float(e['distance'])
    path.reverse()

    return(path, dist, num_visited)
    # raise NotImplementedError("To be implemented")
    # End your code (Part 1)
```

DFS

```
def dfs(start, end):
    # Begin your code (Part 2)
    '''
    same concept as explained in BFS
    '''
    f = open(edgeFile)
    reader = csv.DictReader(f)
    edges = [] #list of dicts [{}, {}...]
    for r in reader:
        edges.append(r)

    adj = {} #dict of list{[], []...}
    for i in range(len(edges)):
        cur = edges[i]['start']
        cur = int(cur)
        dest = edges[i]['end']
        dest = int(dest)
        if cur not in adj:
            adj[cur] = []
            adj[cur].append(dest)
        else:
            adj[cur].append(dest)
```

```python
    '''
    Well, DFS and BFS are only slightly different. The main difference is that
    BFS use queue and DFS use stack to determine which vertice is going to
    process. So, silimarly, I used a list called 'visited', a dictionary called
    'parents' to help me save the information I need. However, as I have
    mentioned above, I used stack to implement DFS instead of queue, which is
    used in BFS. During the implementation, if we confronted a vertice which
    hasn't been visited, I append it onto the stack. And the important thing
    is, this time, I append the list 'visited' out of the for loop instead of
    marking them in the for loop, which means that I would mark vertices as
    visited only after all of their reachable vertices are prcessed. On the
    contrary, BFS algorithm marked vertices visited 'once' they are processed,
    regardless of its neighoring vertices. Now back to DFS, after finding the
    ending point, clear up the stack and finish the DFS searching.
    '''
    stack = []
    stack.append(start)
    visited = []
    visited.append(start)
    parents = {}
    # parents[start] = -1
    while (len(stack)):
        s = stack[-1]
        stack.pop()
        visited.append(s)
        if s not in visited:
            visited.append(s)
        if s not in adj:
            continue
```

```python
        for node in range(len(adj[s])):
            cur = adj[s][node]
            if cur not in visited:
                stack.append(cur)
                parents[cur] = s
                if cur == end:
                    stack.clear()
                    parents[end] = s
                    break

    '''
    same concept as explained in BFS
    '''
    path = [end]
    dist = 0
    num_visited = len(visited) - 1 # start doesn't count
    parent = end
    while(parent != start):
        cur = parent
        parent = parents[cur]
        path.append(parent)
        for e in edges:
            s = str(parent)
            dest = str(cur)
            if e['start'] == s and e['end'] == dest:
                dist += float(e['distance'])
    path.reverse()

    return(path, dist, num_visited)
    # raise NotImplementedError("To be implemented")
    # End your code (Part 2)
```

UCS

```python
def ucs(start, end):
    # Begin your code (Part 3)
    file = open(edgeFile)
    reader = csv.DictReader(file)
    edges = []
    for r in reader:
        edges.append(r)
    # start end distance speedLimit
    '''
    This time, append another element which represents the distance of the
    adjacency vertices. Other part just stay the same.
    '''
    adj = {}
    for i in range(len(edges)):
        cur = edges[i]['start']
        cur = int(cur)
        dest = edges[i]['end']
        dest = int(dest)
        dist = edges[i]['distance']
        dist = float(dist)
        if cur not in adj:
            adj[cur] = []
            adj[cur].append([dest, dist])
        else:
            adj[cur].append([dest, dist])
```

```python
    '''
    In UCS, I used the concept of priority queue to implement the alogprithm.
    In every while loop, I choose the vertice with the smallest distance to
    process, and find its neighboring vertices. If that vertice hasn't been
    visited before, sum up their distance and put the [vertice, distance] into
    pq. Else, I compared new calculated total distance with existing total
    distance, if the current one is smaller than the existing one, update its
    value. This step is also called relaxation. Again, once we reach to the
    ending point, stop the algorithm and do the next part.
    '''
    pq = [] #priority queue
    pq.append([start, 0]) # [vertice, dist]
    visited = []
    visited.append(start)
    parents = {}
    while (len(pq)):
        pq = sorted(pq, key = lambda pq : pq[1]) # sort by dist
        top = pq.pop(0)
        s = top[0]
        dist = top[1]
        for i in range(len(adj[s])):
            cur = adj[s][i][0]
            curDist = adj[s][i][1]
            if cur not in adj:
                continue
            elif cur not in visited:
                pq.append([cur, dist + curDist])
                visited.append(cur)
                parents[cur] = s
```

```python
                if cur == end:
                    pq.clear()
                    parents[end] = s
                    break
            elif cur in visited:
                # relaxation
                for j in range(len(pq)):
                    if pq[j][0] == cur and pq[j][1] > dist + curDist:
                        pq[j][1] = dist + curDist
                        parents[cur] = s
    '''
    Well, stay the same again.
    '''
    path = [end]
    dist = 0
    num_visited = len(visited) - 1 # start doesn't count
    parent = end
    while(parent != start):
        cur = parent
        parent = parents[cur]
        path.append(parent)
        for e in edges:
            s = str(parent)
            dest = str(cur)
            if e['start'] == s and e['end'] == dest:
                dist += float(e['distance'])
    path.reverse()

    return(path, dist, num_visited)
    # raise NotImplementedError("To be implemented")
    # End your code (Part 3)
```

A*

```python
def astar(start, end):
    # Begin your code (Part 4)
    '''
    In order to implement a* algorithm, we need to read the heuristicFile,
    which contains each vertice's heuristic distance.
    '''
    hfile = open(heuristicFile)
    hreader = csv.DictReader(hfile)
    h = []
    for r in hreader:
        h.append(r)
    # node 1079387396 1737223506 8513026827

    file = open(edgeFile)
    reader = csv.DictReader(file)
    edges = []
    for r in reader:
        edges.append(r)
    # start end distance speedLimit

    adj = {}
    for i in range(len(edges)):
        cur = edges[i]['start']
        cur = int(cur)
        dest = edges[i]['end']
        dest = int(dest)
        dist = edges[i]['distance']
        dist = float(dist)
        if cur not in adj:
            adj[cur] = []
            adj[cur].append([dest, dist])
        else:
            adj[cur].append([dest, dist])
```

```python
39      '''
40      This for loop is to find the heuristic distance of the starting point
41      '''
42      # find the heuristic distance
43      for i in range(len(h)):
44          if h[i]['node'] == str(start):
45              hDist = float(h[i][str(end)])
46      '''
47      In a* algorithm, we not only need to concern each edge's distance, but also
48      the heuristic distance of the coming node. So I put [vertice,
49      distance, heuristic distance] into the priority queue. In the for loop at
50      64-89, we need to access the current comparing vertice's heuristic
51      distance, and sum up the accumulated distance so far, the edge's distance
52      and the heuristic distance of current vertice, then put these information
53      back in the queue. On the other hand, if a vertice is already in the queue,
54      do the relaxation steps, which can help us find the shortest path to the
55      ending point. Keep doing these steps until we reach to the ending point.
56      '''
57      pq = []  # priority queue
58      pq.append([start, 0, hDist])  # [vertice, dist, total dist]
59      visited = []
60      visited.append(start)
61      compared = []
62      compared.append(start)
63      parents = {}
64      while (len(pq)):
65          pq = sorted(pq, key=lambda pq: pq[2])  # sort by total dist
66          top = pq.pop(0)
67          s = top[0]
68          dist = top[1]
69          visited.append(s)
70          for i in range(len(adj[s])):
71              cur = adj[s][i][0]
72              curDist = adj[s][i][1]
73              for j in range(len(h)):
74                  if h[j]['node'] == str(cur):
75                      hDist = float(h[j][str(end)])
76              if cur not in adj:
77                  continue
78              elif cur not in compared:
79                  totDist = dist + curDist + hDist
80                  pq.append([cur, dist + curDist, totDist])
81                  compared.append(cur)
82                  parents[cur] = s
83                  if cur == end:
84                      visited.append(end)
85                      pq.clear()
86                      parents[end] = s
87                      break
88              elif cur in compared:
89                  # relaxation
90                  for k in range(len(pq)):
91                      if pq[k][0] == cur and pq[k][1] > dist + curDist:
92                          pq[k][1] = dist + curDist
93                          pq[k][2] = pq[k][1] + hDist
94                          parents[cur] = s
```

```python
    '''
    As usual, the same
    '''
    path = [end]
    dist = 0
    num_visited = len(visited) - 1 # start doesn't count
    parent = end
    while(parent != start):
        cur = parent
        parent = parents[cur]
        path.append(parent)
        for e in edges:
            s = str(parent)
            dest = str(cur)
            if e['start'] == s and e['end'] == dest:
                dist += float(e['distance'])
    path.reverse()

    return(path, dist, num_visited)
    # raise NotImplementedError("To be implemented")
    # End your code (Part 4)
```

A*(time)

```python
def astar_time(start, end):
    # Begin your code (Part 6)
    hfile = open(heuristicFile)
    hreader = csv.DictReader(hfile)
    h = []
    for r in hreader:
        h.append(r)
    # node 1079387396 1737223506 8513026827

    file = open(edgeFile)
    reader = csv.DictReader(file)
    edges = []
    for r in reader:
        edges.append(r)
    # start end distance speedLimit
    '''
    Above is all the same as astar.
    However, in the folowing part, since we need take 'time' as the comparing
    factor instead of the distance, this time I save the speed limit of each
    edge into adjacency list.
    '''
    adj = {}
    for i in range(len(edges)):
        cur = edges[i]['start']
        cur = int(cur)
        dest = edges[i]['end']
        dest = int(dest)
        dist = edges[i]['distance']
        dist = float(dist)
        speed = edges[i]['speed limit']
        speed = float(speed)
        time = dist / speed
        if cur not in adj:
            adj[cur] = []
            adj[cur].append([dest, time, speed])
        else:
            adj[cur].append([dest, time, speed])

    # find the heuristic distance
    for i in range(len(h)):
        if h[i]['node'] == str(start):
            hDist = float(h[i][str(end)])
```

```python
    '''
    The main idea is still the same as astar while the factor we concerned
    change from distance to time. In this algorithm, every time we access a
    vertice, we need to also access its speed limit to calculate the time if we
    take on this edge would cost.

    '''
    pq = []  # priority queue
    hTime = hDist / (60 * 1000 / 3600) # set a speed
    pq.append([start, 0, hTime])  # [vertice, time, total time]
    visited = []
    visited.append(start)
    compared = []
    compared.append(start)
    parents = {}
    while (len(pq)):
        pq = sorted(pq, key=lambda pq: pq[2])  # sort by total time
        top = pq.pop(0)
        s = top[0]
        time = top[1]
        visited.append(s)
        for i in range(len(adj[s])):
            cur = adj[s][i][0]
            curTime = adj[s][i][1]
            curSpeed = adj[s][i][2]
            for j in range(len(h)):
                if h[j]['node'] == str(cur):
                    hDist = float(h[j][str(end)])
            hTime = hDist / curSpeed
            if cur not in adj:
                continue
            elif cur not in compared:
                totTime = time + curTime + hTime
                pq.append([cur, time + curTime, totTime])
                compared.append(cur)
                parents[cur] = s
                if cur == end:
                    visited.append(end)
                    pq.clear()
                    parents[end] = s
                    break
            elif cur in compared:
                # relaxation
                for k in range(len(pq)):
                    if pq[k][0] == cur and pq[k][1] > time + curTime:
                        pq[k][1] = time + curTime
                        pq[k][2] = pq[k][1] + hTime
                        parents[cur] = s
```

```python
    '''
    Only change the total distance into total time. Remember to do the unit
    conversion.
    '''
    path = [end]
    time = 0
    num_visited = len(visited) - 1 # start doesn't count
    parent = end
    while(parent != start):
        cur = parent
        parent = parents[cur]
        path.append(parent)
        for e in edges:
            s = str(parent)
            dest = str(cur)
            if e['start'] == s and e['end'] == dest:
                speed = float(e['speed limit']) * 1000 / 3600
                time += float(e['distance']) / speed
    path.reverse()

    return(path, time, num_visited)
    # raise NotImplementedError("To be implemented")
    # End your code (Part 6)
```

Part II. Results & Analysis (12%):

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: 4978.881999999998 m

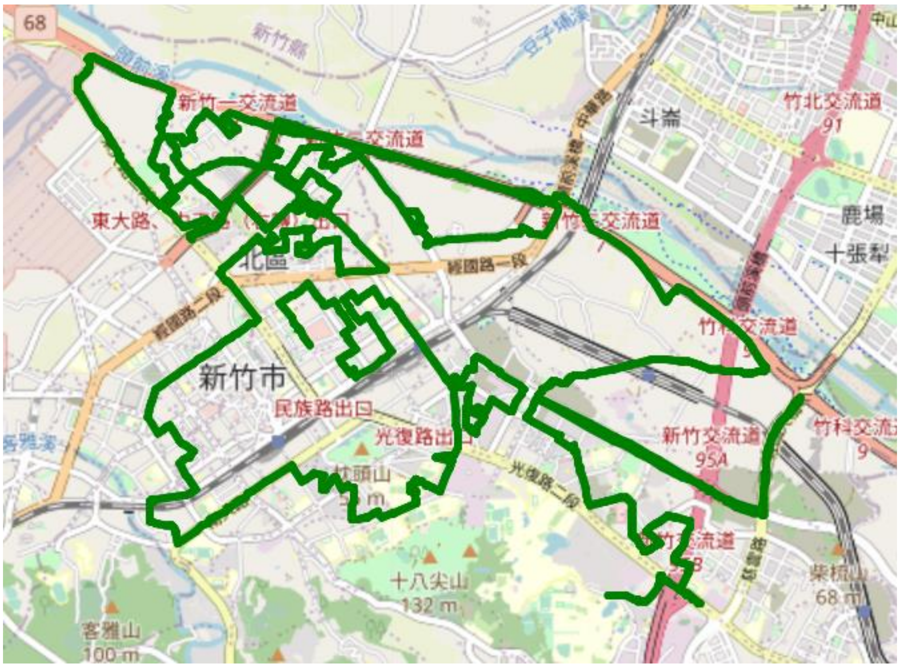
The number of visited nodes in BFS: 4273

DFS:

The number of nodes in the path found by DFS: 1232

Total distance of path found by DFS: 57208.987 m

The number of visited nodes in DFS: 4380

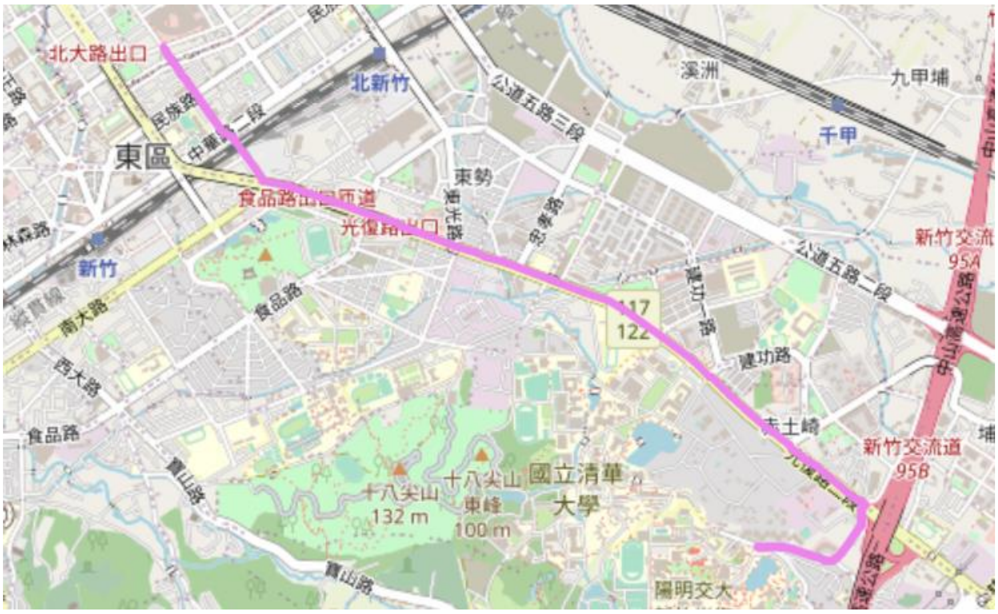


UCS:

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: 5067



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: 10834

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: 218

Test2: 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: 4606

DFS:

The number of nodes in the path found by DFS: 998

Total distance of path found by DFS: 41094.657999999916 m

The number of visited nodes in DFS: 8627

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: 6895

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: 1171

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: 1075

Test3: 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:

The number of nodes in the path found by DFS: 1521

Total distance of path found by DFS: 64821.60399999999 m

The number of visited nodes in DFS: 3370

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: 11913

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: 7067

A*(time):

The number of nodes in the path found by A* search: 234

Total second of path found by A* search: 914.3259868458375 s

The number of visited nodes in A* search: 5556

Part III. Question Answering (12%):

1. Please describe a problem you encountered and how you solved it.
   First, I have never heard of UCS and A* before, so it took me a lot of time to

understand them. Also, BFS and DFS in this homework is somehow different from ones I’ve known. And I want to say that whether the vertice is visited is very important. Sometimes I didn’t put the ‘visited’ list in the correct place, and it leads to compile error.

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.

   Traffic, traffic light and whether there is an accident or not. Because these factors affect the speed directly. Also, weather would sometimes affect the speed, such as rainy or a foggy weather, drivers tend to drive slower than they used to.

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 mapping, we can use the satellite to help us see the surrounding, and build the map. As for localization, we can use sensors and a feature-matching algorithm to find where we are.

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 based on other attributes. Please define a dynamic heuristic function for ETA. Please explain the rationale of your design.

   In my opinion, besides distance, I would put traffic(current road speed), traffic light into concern. The former is that we can avoid being stuck in the traffic. If a certain road, even a highway, has a traffic jam, why would we just change to other road even if its distance is a little further. As for the latter, if we can select a path with less traffic light, we can save the time of waiting for the red lights.