Hw 2: Route Finding Report

111652040 吳泓諺

Part I. Implementation

utils

Note that: Functions in utils are used in part 1, 2, 3, 4, so I exhibit these functions here

```python
import csv
edgeFile = 'edges.csv'
hFile = "heuristic.csv"


def read_edges():
    """
    read edges from edges.csv
    """
    graph = {}
    with open(edgeFile, "r") as file:
        # Create a CSV reader object
        csv_reader = csv.DictReader(file)

        # Iterate over each row in the CSV file
        for row in csv_reader:
            u = int(row["start"])
            if u not in graph:
                graph[u] = []
            graph[u].append((int(row["end"]), float(row["distance"]), float(row["speed limit"])))

    return graph
```

```python
def get_heuristic(end):
    """
    read edges from heuristic.csv
    """
    h = {}
    with open(hFile, "r") as file:
        # Create a CSV reader object
        csv_reader = csv.DictReader(file)

        # Iterate over each row in the CSV file
        for row in csv_reader:
            id = int(row["node"])
            h[id] = float(row[f"{end}"])

    return h

def get_path(last, start, end):
    """
    get the path from start to end with dictionary last
    """
    reverse_path = []
    now = end
    while now != start:
        reverse_path.append(now)
        now = last[now]
    reverse_path.append(start)

    return reverse_path[::-1]


def Nearest(open):
    """
    get the node with smallest f
    """
    nearest = 0
    f_min, g_min = float('Inf'), float('Inf')
    for key, value in open.items():
        if value['f'] < f_min:
            f_min = value['f']
            g_min = value['g']
            nearest = key
        elif value['f'] == f_min and value['g'] < g_min:
            g_min = value['g']
            nearest = key

    return nearest
```

part 1. BFS

```python
from utils import read_edges, get_path
import queue


def bfs(start, end):
    # Begin your code (Part 1)
    """
    bfs is done with queue.
    In each time, we update queue if nodes of path to v < open[v]["nodes"]
    , putting neighbor v into queue
    Then we take out the nearest node from queue
    until we find destination

    """
    graph = read_edges()
    que = queue.Queue()
    open, last = {}, {}

    now, bfs_visited = start, 0
    open[now], open[start]["nodes"], open[start]["dist"] = {}, 0, 0

    while now != end:
        bfs_visited += 1
        update(now, graph, que, open, last)
        now = que.get()

    return bfs_path, bfs_dist, bfs_visited
    # End your code (Part 1)


def update(now, graph, que, open, last):
    if now not in graph:
        return

    nodes = open[now]["nodes"]
    dist_now = open[now]["dist"]
    for v, dist, _ in graph[now]:
        if v not in open:
            open[v] = {}
            open[v]["nodes"] = float('inf')

        if nodes + 1 < open[v]["nodes"]:
            open[v]["nodes"] = nodes + 1
            open[v]["dist"] = dist_now + dist
            last[v] = now
            que.put(v)
```

part 2. DFS stack

```python
from utils import read_edges, get_path
from collections import deque


def dfs(start, end):
    """
    dfs is done with stack.
    In each time, we update stack when we find neighbors of node now
    , putting neighbor v into stack
    Then we take out the arbitrary neighbor of now from stack
    until we find destination

    """
    # Begin your code (Part 2)
    graph = read_edges()
    stack = deque()
    open, close, last = {}, {}, {}

    now, dfs_visited = start, 0
    open[now], open[start]["dist"] = {}, 0

    while now != end:
        dfs_visited += 1
        update(now, graph, stack, open, close, last)
        close[now] = open[now]
        now = stack.pop()


    dfs_path = get_path(last, start, end)
    dfs_dist = open[end]['dist']

    return dfs_path, dfs_dist, dfs_visited
    # End your code (Part 2)


def update(now, graph, stack, open, close, last):
    if now not in graph:
        return

    for v, dist, _ in graph[now]:
        if v in close:
            continue
        elif v not in open:
            open[v] = {}

        open[v]["dist"] = open[now]["dist"] + dist
        last[v] = now
        stack.append(v)
```

part 3. UCS

```python
from utils import read_edges, get_path
import heapq


def ucs(start, end):
    # Begin your code (Part 3)
    """
    ufs is done with priority queue (pq)
    In each time, we update pq if dist of path to v < open[v]["dist"]
    , putting neighbor v into pq
    Then we take out the node with smallest dist from pq
    until we find destination

    """
    graph = read_edges()
    pq = []
    open, close, last = {}, {}, {}

    now, ucs_visited = start, 0
    open[start], open[start]["dist"] = {}, 0

    while now != end:
        ucs_visited += 1
        if now not in close:
            update(now, graph, last, pq, open, close)
            close[now] = open[now]
        now = heapq.heappop(pq)[-1]

    ucs_path = get_path(last, start, end)
    ucs_dist = open[end]["dist"]

    return ucs_path, ucs_dist, ucs_visited
    # End your code (Part 3)


def update(now, graph, last, pq, open, close):
    if now not in graph:
        return

    for v, dist, _ in graph[now]:
        if v in close:
            continue
        elif v not in open:
            open[v] = {}
            open[v]["dist"] = float('inf')

        if open[now]["dist"] + dist < open[v]["dist"]:
            open[v]["dist"] = open[now]["dist"] + dist
            heapq.heappush(pq, (open[v]['dist'], v))
            last[v] = now
```

part 4. A*

```python
from utils import read_edges, get_heuristic, get_path, Nearest


def astar(start, end):
    # Begin your code (Part 4)
    """
    astar is done with dictionary (open).
    In each time, we update open if dist of path to v < open[v]["g"]
    , putting neighbor v into open
    Then we take out the node with smallest f from open
    until we find destination

    """
    graph = read_edges()
    h = get_heuristic(end)
    open, close, last = {}, {}, {}

    now, open[start] = start, {}
    open[start]["f"], open[start]["g"] = h[start], 0

    while now != end:
        update(now, graph, last, h, open, close)
        close[now] = open[now]
        open.pop(now)
        now = Nearest(open)

    astar_path = get_path(last, start, end)
    astar_dist = open[end]["g"]
    astar_visited = len(close) + 1

    return astar_path, astar_dist, astar_visited
    # End your code (Part 4)


def update(now, graph, last, h, open, close):
    if now not in graph:
        return

    g_now = open[now]["g"]
    for v, dist, _ in graph[now]:
        if v in close:
            continue
        elif v not in open:
            open[v] = {}
            open[v]["g"] = float('inf')

        if dist + g_now < open[v]["g"]:
            open[v]["g"] = dist + g_now
            open[v]["f"] = open[v]["g"] + h[v]
            last[v] = now
```

bonus A* time

```python
from utils import read_edges, get_heuristic, get_path, Nearest


def astar_time(start, end):
    # Begin your code (Part 6)
    """
    astar time is done with dictionary (open).
    In each time, we update open if dist of path to v < open[v]["g"]
    , putting neighbor v into open
    Then we take out the node with smallest f from open
    until we find destination
    Note that : we set heuristic function = 0
    """
    graph = read_edges()
    h = get_heuristic(end)
    open, close, last = {}, {}, {}

    now, open[start] = start, {}
    open[start]["f"], open[start]["g"] = h[start], 0

    while now != end:
        update(now, graph, last, h, open, close)
        close[now] = open[now]
        open.pop(now)
        now = Nearest(open)

    time_path = get_path(last, start, end)
    time = open[end]["f"]
    time_visited = len(close) + 1

    return time_path, time, time_visited
    # End your code (Part 6)


def update(now, graph, last, h, open, close):
    if now not in graph:
        return

    g_now = open[now]["g"]
    for v, dist, speed_limit in graph[now]:
        if v in close:
            continue
        elif v not in open:
            open[v] = {}
            open[v]["g"] = float('inf')

        # set heuristic function = 0
        # h(x) = 0 is admissible because 0 smaller than travel time definitely
        if g_now + dist * 36 / (10 * speed_limit) < open[v]["g"]:
            open[v]["g"] = g_now + dist * 36 / (10 * speed_limit)  # turn km/hr to m/s
            open[v]["f"] = open[v]["g"] + 0
            last[v] = now
```

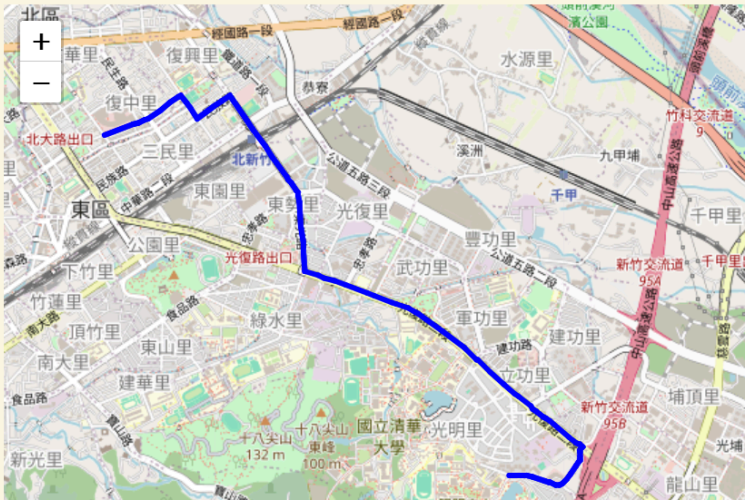
Part II. Results & Analysis

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: 4380
```

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: 5231
```

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
```

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: 311
```

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: 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: 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: 7453
```

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
```

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: 1235
```

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: 3370
```

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: 12311
```

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: 7073
```

A* time

```
The number of nodes in the path found by A* search: 209
Total second of path found by A* search: 779.527922836848
The number of visited nodes in A* search: 11552
```

Part III. Question Answering (12%):

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

In the begining, I have no idea how to find a optimal solution of A* time function. However, after checking the meaning of addmissible function, I realized that I could find a optimal path by setting a addmissible function.
I set h(x) = 0 eventually.

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 congestion may be a arrtribute for route finding. In fact, congestion may affect the speed of driving, it is a more significant factor than speed limit.

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

Well, we could use camera and sensor to map the environment, and with dfs, we could record all the roads on the map.
Latter when we are localizing, we would do object detection on the surroundings. We could find where we are with the records obtained beforehand.

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.

The formula

\[ETA_t = prep\ time_t + driving\ time\ to\ customer_c + other\ factor\]

Definition

\[prep\ time_t\] it is the preperation time of restaurant.
At time t, when prep time is a little too long, it will increment the expected time

\[driving\ time\ to\ customer_c\] it is the driving time from restaurant to customer
When deliver change © the path, it will change the expected time in the same time

\[other\ factor\] if there are multiple orders, it would take the travel time from restaurants to restaurants take into account.
if there are some orders with higher priority than this order, deliver have to finish those order first

Explain

estimated time of arrival equals the sum of three factor aboves. Every order must consider preperation time and driving time.
Sometimes we may consider other factors like there are order with higher priority than this order or there are multiple order
Besides, each order will adjust with schedule