HW2 0616086 邱彥慈

Part 0 : data structure (0%)

Whole works are applied in class Graph.

Information of edges.csv and heuristic.csv are stored in graph & heur (in format of dictionary) by calling addEdge() and addheaur().

<table>
<tr><th>Graph structure</th><th>Read data</th></tr>
<tr><td>
<pre>
from collections import defaultdict
from queue import PriorityQueue

class Node:
  def __init__(self, parent = None, id = None, dist = 0.0, speed = 0.0):
    self.parent = parent
    self.id = id
    self.dist = dist
    self.speed= speed

class Graph:
  def __init__(self):
    self.graph = defaultdict(list)
    self.heur = defaultdict(list)

  def addEdge(self,v1,v2,d,speed):
    self.graph[v1].append(Node(v1,v2,d,speed))

  def addheur(self,v,h1,h2,h3):
    self.heur[v]=[h1,h2,h3]

  def bfs(self, src, dst):
  def dfs(self, src, dst):
  def ucs(self, src, dst):
  def astar(self, src, dst, htype):
</pre>
</td><td>
<pre>
import csv
g = Graph() # num(node) = 12326

with open('./edges.csv', newline='') as csvfile:
  rows = csv.reader(csvfile, delimiter=',')
  headers = next(rows)
  for row in rows:
    v1 = (int)(row[0])
    v2 = (int)(row[1])
    dist = float(row[2])
    speed = float(row[3]) * 1000 / 3600 # m/s
    g.addEdge(v1,v2,dist,speed)

with open('./heuristic.csv', newline='') as csvfile:
  rows = csv.reader(csvfile, delimiter=',')
  headers = next(rows)
  for row in rows:
    v = (int)(row[0])
    h1 = (float)(row[1])
    h2 = (float)(row[2])
    h3 = (float)(row[3])
    g.addheur(v,h1,h2,h3)
</pre>
</td></tr>
</table>

Part 1: BFS (10%)

(1) Use set() to store the visited node_ids, which contains only unique elements.

(2) The frontier is applied by queue, thus here apply pop(0) instead of pop() due to FIFO.

(3) What pushed into queue is a tuple with 2 elements: ([path_from_src], distance)

Pseudo code

```
Visited node set: set()
Frontier : queue
Pop the first node in queue:
    # If the node is visited, just pop it
    If the node is not visited:
        Neighbors = expand(node)
        For each neighbor:
            New_path = [path] + child_node
            New_dist = dist(node) + edge(node,neighbor)
            queue.put(new_path, new_dist)
        visited.add(node)
```

code

```python
def bfs(self, src, dst):
  visited = set()
  q = [([src],0.0)]
  while q:
    path, cur_dist = q.pop(0)
    node_id = path[-1]
    if node_id not in visited:
      if node_id == dst:
        return (path,cur_dist,len(visited))
      neighs = self.graph[node_id]
      for neigh in neighs:
        new_path = list(path)
        new_path.append(neigh.id)
        q.append((new_path, cur_dist + neigh.dist))
      visited.add(node_id)

  print("Path not found")
  return
```

The node_id is retrieved from the last element in path.

The distance stored in tuple is calculated from src.

Part 2: DFS (10%)

(1) The frontier is applied by stack.

Pseudo code

Visited node set: set()
Frontier : stack
Pop the node at the top of the stack:
    # If the node is visited, just pop it
    If the node is not visited:
        Neighbors = expand(node)
        For each neighbor:
            stack.push(new_path, new_dist)
        visited.add(node)

code

```python
def dfs(self, src, dst):
  visited = set()
  stack = [([src],0.0)]
  while stack:
    path, cur_dist = stack.pop()
    node_id = path[-1]
    if node_id not in visited:
      if node_id == dst:
        return (path,cur_dist,len(visited))
      neighs = self.graph[node_id]
      for neigh in neighs:
        new_path = list(path)
        new_path.append(neigh.id)
        stack.append((new_path,cur_dist + neigh.dist))
      visited.add(node_id)

  print("Path not found")
  return
```

Part 3: UCS (20%)

(1) The frontier is applied by Priority queue.

(2) The 1st element of tuple is changed to distance, so the priority queue will sort by distance.

Pseudo code

Visited node set: set()
Frontier : priority queue
Pop the node with smallest distance in queue:
  # If the node is visited, just pop it
  # So when the element in the frontier has to update its distance, we just push a new one into the priority queue since the one with smaller value will be retrieved first, and thus the old one will be popped without expanded.
  If the node is not visited:
    Neighbors = expand(node)
    For each neighbor:
        priority_queue.put (new_dist, new_path)
    visited.add(node)

code

```python
def ucs(self, src, dst):
  visited = set()
  pq = PriorityQueue()
  pq.put((0.0, [src]))
  while pq:
    cur_dist, path = pq.get()
    node_id = path[-1]
    if node_id not in visited:
      if node_id == dst:
        return (path,cur_dist,len(visited))
      neighs = self.graph[node_id]
      for neigh in neighs:
        if neigh.id not in visited:
          new_path = list(path)
          new_path.append(neigh.id)
          pq.put((cur_dist + neigh.dist, new_path))
      visited.add(node_id)

  print("Path not found")
  return
```

pq.get() will pop and return the stack element with smallest distance.

Part 4 : A* (20%)

(1) The frontier is applied by Priority queue.

(2) The distance stored in tuple is g+h instead of g in ucs.
(g is referred to distance calculated from src, while h is referred to the heuristic value)

(3) There's an additional parameter h_type in A* function, which is used to identify the heuristic function we should use. (h_type is equal to place _option)

Pseudo code

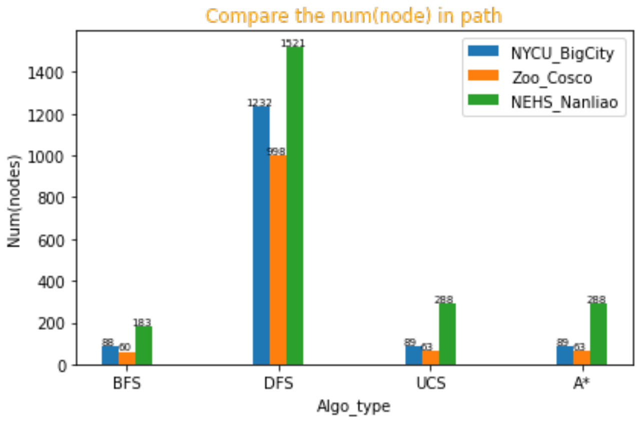
Visited node set: set()
Frontier : priority queue
Pop the node with smallest distance in queue:
  # If the node is visited, just pop it
  # So when the element in the frontier has to update its distance, we just push a new one into the priority queue since the one with smaller value will be retrieved first, and thus the old one will be popped without expanded.
  If the node is not visited:
    Neighbors = expand(node)
    For each neighbor:
      priority_queue.put (new_dist + heuristic(node_id), new_path))
    visited.add(node)

code

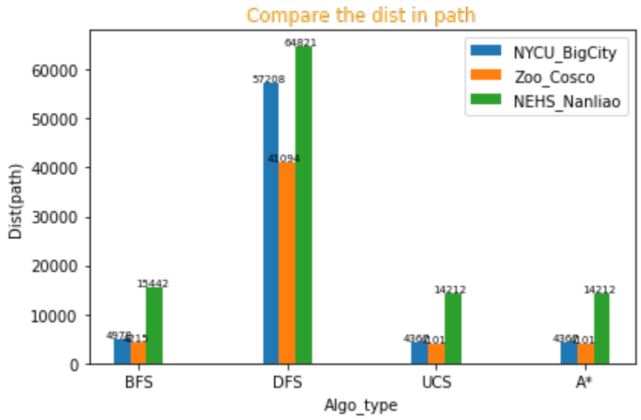
```python
def astar(self, src, dst, h_type):
  visited = set()
  pq = PriorityQueue()
  pq.put((0.0 + self.heur[src][h_type], 0.0, [src]))
  while pq:
    cur_gh, cur_dist, path = pq.get()
    node_id = path[-1]
    if node_id not in visited:
      if node_id == dst:
        visited.add(node_id)
        return (path,cur_dist,len(visited))
      neighs = self.graph[node_id]
      for neigh in neighs:
        if neigh.id not in visited:
          new_path = list(path)
          new_path.append(neigh.id)
          new_dist = cur_dist + neigh.dist
          pq.put((new_dist + self.heur[neigh.id][h_type], new_dist, new_path))
    visited.add(node_id)

  print("Path not found")
  return
```

Part 5: Test your implementation (15%)

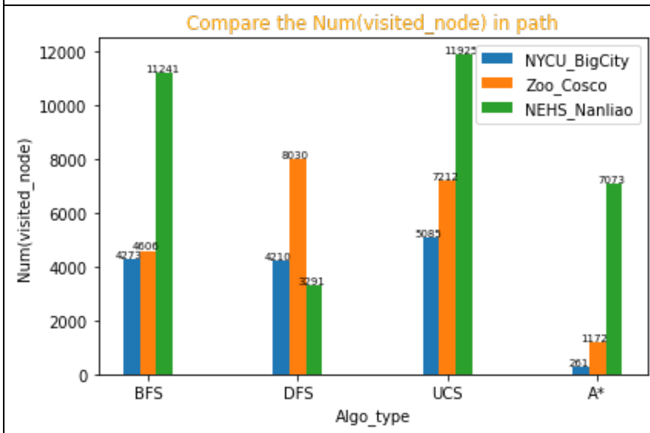
Number of nodes in the found path



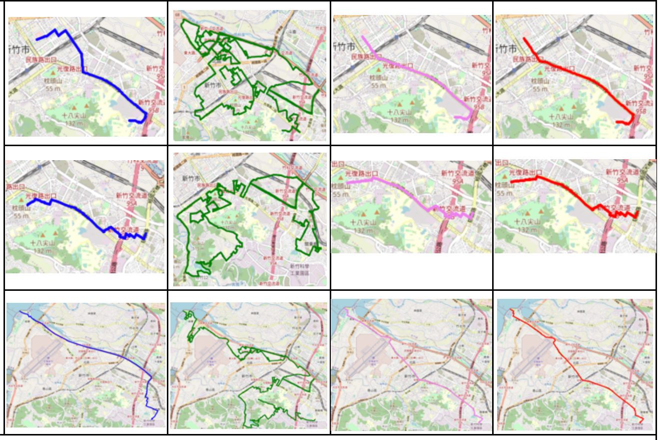
Distance of the found path



Number of nodes expanded



result



Observations

* ** Let the number of nodes visited to be cost ***

(1) Path length : A* = UCS < BFS << DFS

(2) Cost of blue route : A* << DFS < BFS < UCS

(3) Cost of orange route : A* << BFS < UCS < DFS

(4) Cost of green route : DFS < A* < BFS < UCS

Discussion

(1) Both num(nodes) and distance of the found path are quite large in DFS.

(2) BFS contains the smallest nodes in path, but it is not optimal in distance.

(3) UCS and A* found the optimal path among all.

(4) A* visited much less nodes than others in blue and orange routes.

(5) DFS visited the least number of nodes in green case, yet it is not a near path.

(6) The path found by UCS and A* are both small, while A* cost less UCS.

(7) From the observations, we can conclude that :

① Overall, A* cost the least especially in short route (blue and orange).

② Regardless of path lengh, DFS is more efficient in finding long route than short route.

Part 6: Implement another classifier

(1) The speed has been converted to m/s (Part 0).

(2) What pushed into queue is a tuple with 3 elements: (estimated_time, time, [path_from_src])

(3) The priority queue will sort by time+h instead of distance+h.

(4) New_heuristic(node) = heuristic(node)/node.speed.

Pseudo code

Visited node set: set()

Frontier : priority queue

Pop the node with smallest distance in queue:

    If the node is not visited:

        Neighbors = expand(node)

        For each neighbor:

            New time = cur_time + (neighbor.dist / neighbor.speed)

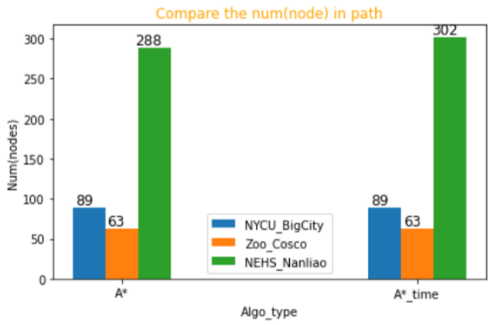
            priority_queue.put (new_time+ heuristic(node_id), new_path))

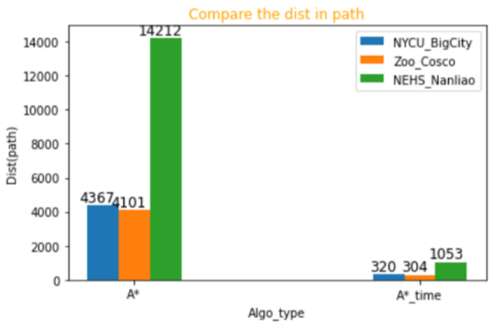
        visited.add(node)

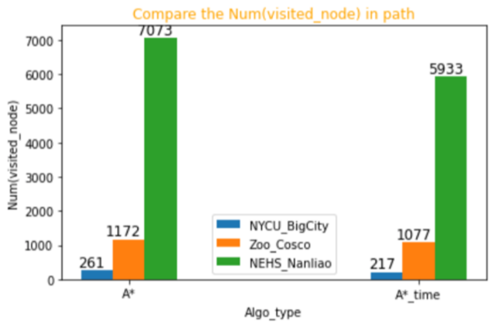
code

```python
def astar_time(self, src, dst, h_type):
  visited = set()
  pq = PriorityQueue()
  pq.put((0.0 , 0.0, [src]))
  while pq:
    cur_gh, cur_time, path = pq.get()
    node_id = path[-1]
    if node_id not in visited:
      if node_id == dst:
        return (path,cur_time,len(visited)) #time_path, time, time_visited
      neighs = self.graph[node_id]
      for neigh in neighs:
        if neigh.id not in visited:
          new_path = list(path)
          new_path.append(neigh.id)
          new_time = cur_time + (neigh.dist/neigh.speed)
          pq.put(( new_time + (self.heur[neigh.id][h_type]/neigh.speed), new_time, new_path))
    visited.add(node_id)

  print("Path not found")
  return
```

result







compaison

(1) Figure 1 :

the number of nodes exist in the path A* < A*_time , but there's no significant difference.

(2) Figure 2:

Skipped since one is evaluated by dist yet the other is evaluated by time.

(3) Figure 3:

The number of nodes explored by A*_time < A*, which means that A*_time is more efficient.

Part7: Describe problems you meet and how you solve them.

(1) The main problems for me are where to store the node_info and what data structure should I apply.
I ended up writing all of them in a class and modify bfs() to g.bfs(), where g = Graph().
Also, the edges and heuristic functions can be searched and accessed by dictionaries stored in Graph.

(2) Another problem is how should I apply heuristic function in Part 6.
I have tried several (a,b) in h(x)= a*new_time + h_dist(x)/b, while the best result still fails to be less than 320.8723(s), which is worse than TA’s result : 320.3284.