Introduction to Artificial Intelligence

HW2-Report

0716325-曾正豪

Part.0: load data

```python
import csv

def edgeloader(filename):
    edge_dict = {}
    with open(filename, newline='') as csvfile:
        rows = csv.reader(csvfile)
        for row in rows:
            if row[0] == 'start':
                continue
            if row[0] in edge_dict:
                edge_dict[row[0]][row[1]] = {'distance':float(row[2]), 'speed_limit':float(row[3])}
            else:
                edge_dict[row[0]] = {row[1] : {'distance':float(row[2]), 'speed_limit':float(row[3])}}
    return edge_dict


def heuristic_loader(filename):
    heuristic_dict = {}
    with open(filename, newline='') as csvfile:
        rows = csv.reader(csvfile)
        go_list = []
        for row in rows:
            if row[0] == 'node':
                for go in row[1:]:
                    go_list.append(go)
            else:
                heuristic_dict[row[0]] = {}
                for i in range(len(go_list)):
                    heuristic_dict[row[0]][go_list[i]] = float(row[i+1])
    return heuristic_dict
```

This is how I load the data from the files. I use the csv module to load the edges and heuristics.

For loading edges, I use a dictionary of dictionaries to save the data since the ID of nodes isn’t start from 0 and not continuous. For each line in the csv file, I read the first element as the first layer key, the second element as the second layer key. Rest of the elements would be saved into it. When we want to access the information of some edge, just use edge_dict[from][to].

For loading heuristics, I use a dictionary of dictionaries to save the data, too. When reading the first line of the csv file, I will construct a list to save the possible end points. Then, for the rest rows, I save the heuristics to each node in the list. When we want to access the heuristics of some node, just use heuristic_dict[node][destination].

Part.1

```python
def bfs(start, end):
    edges = edgeloader('edges.csv')

    bfs_path = []
    bfs_dist = 0
    bfs_visited = 0

    node_info = {}
    for node_idx in edges:
        node_info[node_idx] = {'distance': 0, 'discover': 0, 'pre' : 0}
    queue = deque()
    queue.append(str(start))
    node_info[str(start)]['discover'] = 1

    while len(queue) > 0:
        bfs_visited += 1
        current_node = queue.popleft()
        if current_node == str(end):
            break
        for next_node in edges[current_node]:
            if next_node in node_info and node_info[next_node]['discover'] == 0:
                queue.append(next_node)
                node_info[next_node]['distance'] = node_info[current_node]['distance'] + 1
                node_info[next_node]['pre'] = current_node
                node_info[next_node]['discover'] = 1
        node_info[current_node]['discover'] = 2

    #back tracing
    cur = str(end)
    bfs_path.append(end)
    while cur != str(start):
        bfs_dist += edges[node_info[cur]['pre']][cur]['distance']
        cur = node_info[cur]['pre']
        bfs_path.append(int(cur))
    bfs_path.reverse()
    return bfs_path, bfs_dist, bfs_visited
```

First, I load the edges from data. Then, I initialize the data to return to 0. Then, I create a dictionary called node_info. It saves the required data of each nodes for the searching algorithm, including distance from the start point, has it been discovered, and its previous node while searching. To implement the BFS algorithm, it is need to use queue. So, I use the deque module in python.

Firstly, I enqueue the start node into the queue and set its ‘discover’ to 1. Then, the loop will be running until it’s empty or the end node is reached. In each iteration, I will dequeue a node, and add 1 to the visited count. If it is the end node, the search is ended. Then, for all the adjacent nodes to the dequeue node, if it has not been discovered yet, I will enqueue it into the queue and set its distance to its previous node’s distance+1, its previous node to the dequeue node, set its discover to 1. In the end of one loop I will set the discover to 2 to indicate that it has totally discovered.

After the searching has completed, I backtracking from the end node. I hold a pointer, and I will trace the information I record in the node_info to the start point, in each iteration, I append the node in the route into the path list, and add the distance to the total distance. Finally, I return the required data.

Part.2

```python
44 def dfs(start, end):
45     edges = edgeloader('edges.csv')
46 
47     dfs_path = []
48     dfs_dist = 0
49     dfs_visited = 0
50 
51     node_info = {}
52     for node_idx in edges:
53         node_info[node_idx] = {'distance': 0, 'discover': 0, 'pre' : 0}
54     stack = deque()
55     stack.append(str(start))
56     node_info[str(start)]['discover'] = 1
57 
58     while len(stack) > 0:
59         dfs_visited += 1
60         current_node = stack.pop()
61         node_info[current_node]['discover'] = 1
62         if current_node == str(end):
63             break
64         for next_node in edges[current_node]:
65             if next_node in node_info and node_info[next_node]['discover'] == 0:
66                 stack.append(next_node)
67                 node_info[next_node]['distance'] = node_info[current_node]['distance'] + 1
68                 node_info[next_node]['pre'] = current_node
69         node_info[current_node]['discover'] = 2
70 
71     #back tracing
72     cur = str(end)
73     dfs_path.append(end)
74     while cur != str(start):
75         dfs_dist += edges[node_info[cur]['pre']][cur]['distance']
76         cur = node_info[cur]['pre']
77         dfs_path.append(int(cur))
78     dfs_path.reverse()
79     return dfs_path, dfs_dist, dfs_visited
```

From line 45 to line 53, they are the same as in part.1. To implement the DFS algorithm, it is need to use stack. So, I use the deque module in python. Firstly, I push the start node into the stack and set its ‘discover’ to 1.

Then, the loop will be running until it’s empty or the end node is reached. In each iteration, I will pop one node, and add 1 to the visited count. If it is the end node, the search is ended. Then, for all the adjacent nodes to the pop node, if it has not been discovered yet, I will push it into the stack and set its distance to its previous node’s distance+1, its previous node to the pop node. In the end of one loop I will set the discover to 2 to indicate that it has totally discovered.

The way how I do the back tracing in line 72 to line 78 is the same as in part.1

Part.3

```
82  def ucs(start, end):
83      edges = edgeloader('edges.csv')
84  
85      ucs_path = []
86      ucs_dist = 0
87      ucs_visited = 0
88  
89      node_info = {}
90      for node_idx in edges:
91          node_info[node_idx] = {'distance': 0, 'discover': 0, 'pre' : 0}
92      priorirty_queue = []
93  
94      heapq.heappush(priorirty_queue, (0, str(start)))
95      node_info[str(start)]['discover'] = 1
96      while len(priorirty_queue) > 0:
97          ucs_visited += 1
98          current_node = heapq.heappop(priorirty_queue)[1]
99          if current_node == str(end):
100             break
101         for next_node in edges[current_node]:
102             if next_node in node_info and node_info[next_node]['discover'] == 0:
103                 node_info[next_node]['distance'] = node_info[current_node]['distance'] + edges[current_node][next_node]['distance']
104                 node_info[next_node]['pre'] = current_node
105                 node_info[next_node]['discover'] = 1
106                 heapq.heappush(priorirty_queue, (node_info[next_node]['distance'], next_node))
107             elif next_node in node_info and node_info[next_node]['discover'] == 1:
108                 for item in priorirty_queue:
109                     if item[1] == next_node:
110                         new_distance = node_info[current_node]['distance'] + edges[current_node][next_node]['distance']
111                         if item[0] > new_distance:
112                             node_info[next_node]['distance'] = new_distance
113                             node_info[next_node]['pre'] = current_node
114                             item = (node_info[next_node]['distance'], next_node)
115                             heapq.heapify(priorirty_queue)
116                         break
117         node_info[current_node]['discover'] = 2
118 
119     cur = str(end)
120     ucs_path.append(end)
121     while cur != str(start):
122         ucs_dist += edges[node_info[cur]['pre']][cur]['distance']
123         cur = node_info[cur]['pre']
124         ucs_path.append(int(cur))
125     ucs_path.reverse()
126     return ucs_path, ucs_dist, ucs_visited
```

From line 83 to line 91, they are the same as in part.1. To implement the UCS algorithm, it is need to use priority queue. So, I use the heapq module in python. I define its key to be the distance from the start point, the value is the ID of that node. Firstly, I insert the start node into the priority queue and set its 'discover' to 1.

Then, I the loop will running until it's empty or the end node is reached. In each iteration, I will extract_min a node, and add 1 to the visited count. If it is the end node, the search is ended. Then, for all the adjacent nodes to the pop node, if it has not been discovered yet, I will insert it into the priority queue and set its distance to its previous node's distance plus the distance of that edge, its previous node to the extract_min node. If the adjacent node is in the priority queue, I will do the Relax and Decrease_Key operations. In the end of one loop I will set the discover to 2 to indicate that it has totally discovered.

The way how I do the back tracing in line 119 to line 125 is the same as in part.1

Part.4

```
129 def astar(start, end):
130     edges = edgeloader('edges.csv')
131     heuristic = heuristic_loader('heuristic.csv')
132 
133     astar_path = []
134     astar_dist = 0
135     astar_visited = 0
136 
137     node_info = {}
138     for node_idx in edges:
139         node_info[node_idx] = {'distance': 0, 'discover': 0, 'pre' : 0}
140     priorirty_queue = []
141 
142     heapq.heappush(priorirty_queue, (0, str(start)))
143     node_info[str(start)]['discover'] = 1
144     while len(priorirty_queue) > 0:
145         astar_visited += 1
146         current_node = heapq.heappop(priorirty_queue)[1]
147         if current_node == str(end):
148             break
149         for next_node in edges[current_node]:
150             if next_node in node_info and node_info[next_node]['discover'] == 0:
151                 node_info[next_node]['distance'] = node_info[current_node]['distance'] + edges[current_node][next_node]['distance']
152                 node_info[next_node]['pre'] = current_node
153                 node_info[next_node]['discover'] = 1
154                 heapq.heappush(priorirty_queue, (node_info[next_node]['distance'] + heuristic[next_node][str(end)], next_node))
155             elif next_node in node_info and node_info[next_node]['discover'] == 1:
156                 for item in priorirty_queue:
157                     if item[1] == next_node:
158                         new_distance = node_info[current_node]['distance'] + edges[current_node][next_node]['distance']
159                         if item[0] > new_distance + heuristic[next_node][str(end)]:
160                             node_info[next_node]['distance'] = new_distance
161                             node_info[next_node]['pre'] = current_node
162                             item = (node_info[next_node]['distance'] + heuristic[next_node][str(end)], next_node)
163                             heapq.heapify(priorirty_queue)
164                         break
165         node_info[current_node]['discover'] = 2
166 
167     cur = str(end)
168     astar_path.append(end)
169     while cur != str(start):
170         astar_dist += edges[node_info[cur]['pre']][cur]['distance']
171         cur = node_info[cur]['pre']
172         astar_path.append(int(cur))
173     astar_path.reverse()
174 
175     return astar_path, astar_dist, astar_visited
```

From line 130 to line 139, they are the same as in part.1. To implement the A* algorithm, it is need to use priority queue. So, I use the heapq module in python. I define its key to be the distance from the start point plus the heuristic function of that node, the value is the ID of that node. Firstly, I insert the start node into the priority queue and set its ‘discover’ to 1.

Then, the loop will be running until it’s empty or the end node is reached. In each iteration, I will extract_min a node, and add 1 to the visited count. If it is the end node, the search is ended. Then, for all the adjacent nodes to the pop node, if it has not been discovered yet, I will insert it into the priority queue and set its distance to its previous node’s distance plus the distance of that edge, its previous node to the extract_min node. The element I insert into the priority queue is “(the distance from the start point plus the heuristic function of that node, the value is the ID of that node)”. If the adjacent node is in the priority queue, I will do the Relax and Decrease_Key operations. In the end of one loop I will set the discover to 2 to indicate that it has totally discovered.

The way how I do the back tracing in line 167 to line 173 is the same as in part.1

Part.5

Screen shot:

NYCU to Big City:

```
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: 4267

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: 4376

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: 5075

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: 256
```

Hsinchu Zoo to COSTCO:

```
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: 4604

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: 8618

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: 7311

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: 1297
```

National Experimental High School at Hsinchu Science Park to Nanliao Fishing Port:

```
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: 11227

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: 3364

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: 11909

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: 7000
```

Discussion:

According to the experiment result, I found some features of different searching algorithms.

BFS tends to find a route that has minimum number of nodes. In some test cases, the BFS would find a route through the high way,

DFS is a disaster. It can find “one” route from start to end, but the path would not been considered by any normal human.

UCS could find the shortest path from start to end.

A* would have less visited node compared with previous algorithms and have a path that is good enough (basically almost the same as the shortest path given by UCS).

Part.6

```
178 def astar_time(start, end):
179     edges = edgeloader('edges.csv')
180     heuristic = heuristic_loader('heuristic.csv')
181 
182     time_path = []
183     time_dist = 0
184     time_visited = 0
185 
186     total_distance = 0
187     total_time = 0
188     edges = edgeloader('edges.csv')
189     for i in edges:
190         for j in edges[i]:
191             edges[i][j]['time'] = edges[i][j]['distance'] / (edges[i][j]['speed_limit'] / 2)
192             edges[i][j]['time'] = max(random.gauss(edges[i][j]['time'], 5), 1)
193             total_distance += edges[i][j]['distance']
194             total_time += edges[i][j]['distance'] / edges[i][j]['speed_limit']
195 
196     avg_speed = total_distance / total_time
197     for h in heuristic:
198         heuristic[h][str(end)] = heuristic[h][str(end)] / (avg_speed - 15)
199 
200     node_info = {}
201     for node_idx in edges:
202         node_info[node_idx] = {'time': 0, 'discover': 0, 'pre' : 0}
203     priorirty_queue = []
204 
205     heapq.heappush(priorirty_queue, (0, str(start)))
206     node_info[str(start)]['discover'] = 1
207     while len(priorirty_queue) > 0:
208         time_visited += 1
209         current_node = heapq.heappop(priorirty_queue)[1]
210         if current_node == str(end):
211             break
212         for next_node in edges[current_node]:
213             if next_node in node_info and node_info[next_node]['discover'] == 0:
214                 node_info[next_node]['time'] = node_info[current_node]['time'] + edges[current_node][next_node]['time']
215                 node_info[next_node]['pre'] = current_node
216                 node_info[next_node]['discover'] = 1
217                 heapq.heappush(priorirty_queue, (node_info[next_node]['time'] + heuristic[next_node][str(end)], next_node))
218             elif next_node in node_info and node_info[next_node]['discover'] == 1:
219                 for item in priorirty_queue:
220                     if item[1] == next_node:
221                         new_distance = node_info[current_node]['time'] + edges[current_node][next_node]['time']
222                         if item[0] > new_distance + heuristic[next_node][str(end)]:
223                             node_info[next_node]['time'] = new_distance
224                             node_info[next_node]['pre'] = current_node
225                             item = (node_info[next_node]['time'] + heuristic[next_node][str(end)], next_node)
226                             heapq.heapify(priorirty_queue)
227                         break
228         node_info[current_node]['discover'] = 2
229 
230     cur = str(end)
231     time_path.append(end)
232     while cur != str(start):
233         time_dist += edges[node_info[cur]['pre']][cur]['time']
234         cur = node_info[cur]['pre']
235         time_path.append(int(cur))
236     time_path.reverse()
237 
238     return time_path, time_dist, time_visited
```

How could we predict the time to go through an edge? Since the traffic in Taiwan is disaster, we never know whether there is a San-Bao(三寶) on the road. So, in line 189 to line 195, I define the time cost on each edge to be (distance / speed limit / 2) and add a random normal number. For the heuristic, I calculate the average speed in the Hsinchu City. Then, I define the heuristic to be (original heuristic / (the average speed – 15)). Rest of the algorithm are the same as in part.4 just replace the 'distance' by 'time'.

Discussion:

Something strange happened, I found that my A* time algorithm sometimes tends to give the route that I would like to ride with my motorcycle in reality. I seldom ride on the route given by BFS, UCS, and A*.