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().

|  |  |
| --- | --- |
| Graph structure | Read data |
|  |  |

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 | |
|  | 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 |
|  |

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 | |
|  | 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)

1. 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 |
|  |

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 :   Overall, A\* cost the least especially in short route (blue and orange).  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 | result |
|  |  |
| compaison |
| 1. Figure 1 :   the number of nodes exist in the path A\* < A\*\_time  , but there’s no significant difference.   1. Figure 2:   Skipped since one is evaluated by dist yet the other  is evaluated by time.   1. 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.

1. 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.