CS6380 Artificial Intelligence

Assignment - 3

General instructions:

1. You have to turn in the well-documented code along with a report of the results of the experiments.

2. Submission Guidelines: Create a folder named 'GroupNo RollNo1 RollNo2' if it's a group or 'GroupNo RollNo' if doing individually. In this folder, you should have your report and a subfolder 'codes' containing the python file corresponding to the algorithm that was assigned to you. Upload this folder(.zip) on moodle. Please follow the naming convention strictly.

3. Any sort of plagiarism/cheating will be dealt with strictly. Acknowledge any source used for performing the experiments.

Submission Deadline: - Saturday, 26 December 2020, 11:59 PM

Search Algorithm (SA)

The program shows stepwise progress of the algorithm along with displaying the relevant nodes as per the algorithm. The end path found is highlighted.

1. **SA1: A* and Weighted A* .**

- Find and mark the least cost path on the generated graph.
- Your program should be able to use a different value of weight and re-run the algorithm on the same graph.
- Colour the open and closed nodes in each iteration.
- The new run should color the nodes in open and closed again and, mark the optimal path on the graph.

2. **SA2: DFID and Iterative Deepening A* (IDA*) .**

- Perform a sequence of deepening searches.
- Color the open and closed nodes in each iteration. In the next iteration color the nodes again. Show the final path found and the cost of the final solution.

3. **SA3: Recursive Best First Search (RBFS).**

- Color the open, closed and the rolled back nodes in each iteration.
- Show the final path found and the cost of the final solution.

4. **SA4: Breadth-First Heuristic Search (BFHS).**

- Colour the following nodes: open and closed, in each iteration.
- Show the final path found and the cost of the final solution.
- Tip: To find the upper bound U, use beam search with f-values and always move to the best neighbour even if worse than the current (refer to sir’s lecture for more details).

5. **SA5: Divide and Conquer BFHS (DCBFHS) .**

- Colour the following nodes: open,closed and relay, in each iteration.
- Show the final path found and the cost of the final solution.
- Tip: To find the upper bound U, use beam search with f-values and always move to the best neighbour even if worse than the current (refer to sir’s lecture for more details).

6. **SA6: Beam Stack Search.**

- Colour the following nodes: open,closed and deleted nodes, in each iteration.
- Since BSS is an anytime algorithm, show the cost of the best solution found every time the upper bound U is refined.
- Show the final path found and the cost of the final solution.

Game Algorithms (GA)

1. **GA1: Alpha-Beta .**

- Your program should be able to run Alpha-Beta from left to right as well as right to left.
- Compute the minimax value and trace the path from where the minimax value comes.
- Colour the nodes and edges visited by the algorithm - Give one colour to the nodes while traversing top down and once the minimax value for a node is computed, colour it using another one.

2. **GA2: SSS* .**

- Compute the minimax value and trace the path from where the minimax value comes.
- Colour the nodes and edges visited by the algorithm - Give one colour to the nodes while traversing top down and once the minimax value for a node is computed, colour it using another one.

You can change the **depth** and **branching factor** of the game tree and your algorithm should run on values set on the leaf nodes(see API specification to understand how to do this).

To verify the correctness, compute the minimax value of the game tree using a minimax algorithm and compare it with the one computed using your algorithm.

For the game algorithms, extra points will be given if you are able to colour nodes having an alpha/beta cutoff with a different colour, compared to a node where there is no cutoff.

API Specification

The algorithm that you code should be written as a Python class, by extending the **AlgorithmBase** class. The AlgorithmBase class provides the following methods which you can use to manipulate and visualize various aspects of the graphs and trees:

1.self.alg_iteration_start(): This is used to denote the start of an iteration of the algorithm.

2.self.alg_iteration_end(): This is used to denote the end of an iteration of the algorithm. This function should be called only after calling alg_iteration_start(). This is used to let the framework update the colouring for the nodes and edges that were updated in the lists (obtained from the call to self.get_list() method), in the current iteration.

3.self.gen_path(): This is used to get the nodes in the path from the start node to the goal node, once the goal node has been added to the 'closed' list. (Refer BFS implementation)

4.self.get_edge_attribute(u:int,v:int,key:string): Returns the value corresponding to the property 'key' for the edge connecting node with index u to the node with index v, provided it has been set previously.

5.self.get_edge_weight(u:int,v:int): Returns the weight for the edge connecting node with index u to the node with index v.

6.self.get_list(name:string,color:list): Get a reference to a list which will be used to track the progress of the algorithm (like ‘open’, ‘closed’, ‘relay’ etc.). Color is an optional parameter to specify what color you want to use for the nodes in the list, which is to be passed as a list of 4 normalized values (between 0 and 1), denoting an RGBA code.

Eg: self.get_list(‘customList’,[1,1,0,1])

Note: You can use any name for the list you are fetching with this method. But we’d suggest you stick with the conventions for the algorithm you’re implementing.

Important: DO NOT overwrite/re-initialize the list obtained after calling this method since it contains a reference to the original list. You may append/remove elements from the lists as needed.

7. self.get_nodes(): Returns a list of the nodes in the graph (same as Graph.nodes in Networkx, refer to the official documentation for details).

8. self.get_node_attribute(u:int,key:string): Returns the value corresponding to the property ‘key’ for the node with index ‘u’. For example, this can be used to get the minimax value of nodes, provided it has been set previously.

9. self.get_tree_params(): Returns the branching factor and height associated with the current tree (only to be used for the game algos).

Note: The tree nodes are indexed from 0 to n-1, where n is the total number of nodes in the tree. Use the branching factor and height of the tree to set the minimax values in the leaf nodes before running the game algorithms.

10. self.get_parent(u:int):Returns the index of the parent of the node ‘u’, returns -1 if the parent of the node is not set, or if it is the root.

11.self.heuristic(u:int,v:int,fn:Python function,f_name='euclidean'): Get the heuristic value associated with the distance from node with index u to the node with index v. ‘fn’ is a function (optional parameter) that can be passed in to calculate heuristic value according to it. f_name is ‘euclidean’ by default, ‘manhattan’ can also be passed to calculate the corresponding heuristic values.

12. self.neighbors(u:int): Returns a list of neighbours of the node with index ‘u’.

13. self.set_edge_attribute(u:int,v:int,key:string,value:any): Set a property ‘key’ for the edge connecting node with index u to the node with index v, using the value ‘value’.

14. self.set_node_attribute(u:int,key:string,value:any): Set a property ‘key’ of the node with index ‘u’, using the value ‘value’. For example, this can be used to set the minimax value of nodes.

15. self.set_parent(u:int,v:int): Set the parent of the node with index ‘u’ as the node with index ‘v’.

16. self.show_info(info:string): This can be used to display the message ‘info’ on the screen. Use this to display the final cost of path found, minimax values and to debug variables with respect to your algorithm.

17.self.show_path(path:string): This can be used to highlight the edges connecting the nodes in the list ‘path’ passed as an argument. Use this to highlight the path from where the minimax value comes in game algorithms.