Assignment #1: Uninformed and Informed Search

Ty Skelton

CS331 - INTRO TO AI
SPRING TERM

OREGON STATE UNIVERSITY

I. Methodology

A. Overhead

I chose to write this assignment in C, which meant I had to be responsible for implementing and managing my own data structures. The script loads the start and goal states into memory in order to retain a constant reference to them as a way to keep its bearings. A linked-list tree is built from all viable successor states leading from the start state provided and all previously visited states are kept in an array so that it doesn't expand any state more than once. At the end of each search algorithm it will report the number of expanded states, which is purely the size of the visited states array, along with the chain of successor actions from start to finish along the "optimal" path.

Each algorithm makes use of the same suite of functions, which are included in the two different header files. First are *check_action()* and *take_action()*. Both contain switch statements and evaluate whatever action is requested (out of the five different possible ones). If a viable action is found, a new state is created in memory that represents the result of the action. If that state already exists in the state array then it is discarded and the existing state is marked as a child, but if it's a previously unexpanded state the script will mark the parent and build a valid successor to be called in whatever fashion the algorithm prefers. Finally, when the goal state is discovered all of the algorithms will update the correct fields and parse the tree from leaf to root recursively and print out the path, length, and expanded nodes.

B. DFS

I chose to use a recursive implementation of DFS. The script passes the DFS function call two parameters: a pointer to the monitor object and a pointer to the current successor node in the tree. The monitor structure is a major part of this program in all of the different algorithms, as it contains pointers to all of the data structures like the successor tree, states list, start / goal states and the queues (bfs and priority). The current successor node is required for the DFS to work recursively, as it will look at the nodes first (leftmost) viable action and call itself again on it, effectively parsing the tree top to bottom and left to right.

C. BFS

Continuing with the theme of recursive functions, I took that approach with BFS as well. BFS is only passed one parameter, which is a pointer to the monitor object. The reason I pass the monitor object as opposed to a pointer to the queue of states to expand in typical BFS fashion is because the monitor object also retains useful information like queue size, states array & size, and pointers to the start & goal states. All of these things are necessary to evalutate the function, so I opted to pass a single pointer rather than a multitude of parameters.

D. IDDFS

IDDFS takes the same aproach as DFS, save for a few minor details that make it unique. Firstly, it will be called iteratively and each iteration will define the depth to which a DFS (DLS) algorithm will delve to find the optimal solution. This process repeats indefinitely until a solution is found.

E. A*

A* is the only informed search used in this assignment and has the most interesting approach details. It is also recursive in nature, but couples together two different pieces of information to help it decide which successor node to follow. First it will run a heuristic over the resulting node after a potential action and store its value. This heuristic for the script is merely a function of the number of people on the right bank divided by the boat capacity. The lower the value is the closer (most likely) is to the optimal solution. This is then coupled with the second piece of information, which is the depth at which this node is. If two nodes have the same heuristic value, then the one with a lower depth takes priority since it could result in a less expensive path. These successors are stored in a linked-list priority queue, which as mentioned earlier is stored as reference by the monitor structure.

II. Results

Results are presented as 'Path Length / Nodes expanded'

Algorithm	Data Set #1	Data Set #2	Data Set #3
DFS	11/13	43/54	557/558
BFS	11/14	33/95	337/7318
IDDFS	11/13	41/54	555/557
A*	11/14	33/59	337/959

III. Discussion

DFS, BFS, and A* all have results that look either accurate or reasonable. The only issue I have with my results lies with those of IDDFS. IDDFS should be optimal across the board, but for some reason it's reporting results in the DFS ball park for all three data sets. After speaking with the professor about my approach and logic, it seems I have the correct approach and could only be made more optimal by storing the depth of explored nodes and choosing to expand them if found at a higher depth (lower depth value). This would mean it wouldn't mark a node on the bottom left of the graph after the first DFS delve and ignore it if it occured on the top right later on. The problem I had with this implementation was that my program was running out of memory at depths of around 30. This was unsustainable even after spending a lot of time trying to figure out what the issue is, but the fact I had the right idea meant my flaw lied in with an unseen bug in my code. I am happy with how my results came out and I know that at the time of turning this in if I had been able to catch whatever minor logic flaw was leading to DFS results in my IDDFS, then it'd be that much better.

IV. Conclusion

From this assignment I can conclude that while BFS, IDDFS, and A* are all able to find the optimal solution, A* definitely is the best approach. BFS and IDDFS cover a lot of ground and expand a lot of nodes which means they are expensive on memory. DFS is able to find a solution without expanding a lot of nodes, which can be favorable when trying to cut costs. However, A* pulls the best from these approaches. Boasting an optimal path and a low expansion number it can find a smart path through the tree.