Zeyad Shureih
4/20/2020

Implementation Assignment 1

Methodology

All of my algorithms return a list of explored states. To determine the path from start to goal, I iterate through the list of expanded nodes. If I find two adjacent states where the boat is on the same bank, I check to see if the node after the second is in the frontier of the second. If it is, I add the second node to the path. For any given node, if none of the adjacent nodes have the boat on the same bank, they are added to the path.

- BFS
    - For test cases 1-3, I ran a standard breadth-first search algorithm by instantiating the game with the respective start and goal states, then running the algorithm. The algorithm outputs the number of expanded nodes, and a list of those nodes; the last node being the goal state. I then use my path finding algorithm to get the path of the result. Expanded nodes are counted as soon as the frontier of a node is found. Because the program is written in Python 3, I use a dequeue as a standard queue.
- DFS
    - For test cases 1-3, I ran a standard depth-first search algorithm by instantiating the game with the respective start and goal states, then running the algorithm. The algorithm outputs the number of expanded nodes, and a list of those nodes; the last node being the goal state. I then use my path finding algorithm to get the path of the result. Expanded nodes are counted as soon as the frontier of a node is found. I use the Python list object to emulate a stack for my frontier.

- IDDFS
    - For IDDFS, I began by selectively choosing a max depth based upon the results of DFS and BFS tests. This became tedious so I set my algorithm to have a max depth of 1000 nodes (chickenWolfSolver.py:308) for all test cases. The number of expanded nodes per iteration is added to a running total, and output once the algorithm has found the goal state.

- A*
    - I defined the heuristic h(n) as $\text{sqrt}((n_{ik} - g_{ik})^2 + (n_{iw} - g_{iw})^2 + (n_{ib} - g_{ib})^2)$ where n is the next state, g is the goal state, and k, w, and b denote the chicken, wolf, and boat counts of the state on each bank i, respectively. I then defined g(n), similarly, as $\text{sqrt}((c_{ik} - n_{ik})^2 + (c_{iw} - n_{iw})^2 + (c_{ib} - n_{ib})^2)$ where c is the current state and n is the next state. This was done to represent g(n) as the change in score between nodes, while h(n) is the overall progress towards the goal node. There were also experiments where g(n) = 1, which indicates the cost of expanding the node *n*, though this was decided against because I felt it didn't convey enough information. In both cases of g(n), the algorithm produces a solution.

Results

Table entries are in the following format:
(number of states in path including start/final, total expanded states)

	Test Case 1	Test Case 2	Test Case 3
BFS	(12, 14)	(36, 60)	(388, 970)
DFS	(12, 13)	(36, 54)	(392, 856)
IDDFS	(12, 77)	(36, 1224)	(392, 367652)
A*	(12, 14)	(36, 69)	(388, 1153)

Discussion

At first, my results were not as expected. IDDFS and A* performed exponentially worse than DFS with regard to path length. That was when I realized I was tracing paths from the starting state to the goal state incorrectly. Once I fixed this with some much needed refactoring, I got the results as they are shown in the table above.

The results above are much more in line with how I expected them to be initially. BFS expands more nodes than DFS, but actually generates a shorter path than DFS in the third test case. I found this a little interesting. If I am understanding the algorithms correctly IDDFS should have also found the 388 step path. There is probably an issue with how I am tracking depth that is causing this. If I were to guess, it is probably because I am increasing depth by 1 every time I expand a node, but not subtracting by however much when I jump back up to an earlier depth. If I had more time, I would refactor both DFS and IDDFS in to recursive solutions so that path and depth can be tracked properly.

A* expands more nodes than I expected it to, but performs as expected, because in the 2nd and 3rd test cases it finds the shortest path.

Conclusion

From these results, for large state spaces (like test case 3) I would use A* or BFS, and for small state spaces (like test cases 1 and 2) I would use DFS or IDDFS.

DFS is not optimal, but if space and time are concerns (measured by the number of nodes expanded), it is the best choice for small state spaces because it expands the least amount of nodes across the board. A second choice would be BFS, because of the optimal paths,

If space and time are not concerns, I would use IDDFS (not mine, an actual correctly implemented version), or A*. Probably A*, because as expected, given the right, admissible heuristic, it performs optimally.