

CS76: AI - 21F - PA1 - Chickens & Foxes Report - Julian Grunauer - 9/21/21

Evaluation

- Do your implemented algorithms actually work? How well?
 - Yes, all the algorithms work! They run in the way they are supposed to and there are no bugs.

Breadth-First Search

Implementation

My BFS algorithm is written in the traditional manner. The root node is placed in a SearchNode, added to a frontier queue, and a visited set is created. The program loops while there are elements in the frontier, and for each cycle, an element is popped off, added to the visited set, and its possible successors are generated. These successor states are looped through and if they haven't been seen before, they are checked to see if they're the goal state and if not, appended to the frontier. If the goal state is not found when the frontier is empty, then the problem has no solution. This runs in $O(b^d)$ time and $O(b^d)$ space.

Results

Chickens and foxes problem: (3, 3, 1) Attempted with search method: BFS Number of nodes visited: 15 Solution length: 12 Path: [(3, 3, 1), (3, 1, 0), (3, 2, 1), (3, 0, 0), (3, 1, 1), (1, 1, 0), (2, 2, 1), (0, 2, 0), (0, 3, 1), (0, 1, 0), (1, 1, 1), (0, 0, 0)]

Chickens and foxes problem: (5, 5, 1) Attempted with search method: BFS no solution found after visiting 13 nodes

Chickens and foxes problem: (5, 4, 1) Attempted with search method: BFS Number of nodes visited: 30 Solution length: 16 Path: [(5, 4, 1), (5, 2, 0), (5, 3, 1), (3, 3, 0), (4, 3, 1), (3, 2, 0), (3, 3, 1), (2, 2, 0), (3, 2, 1), (2, 1, 0), (2, 2, 1), (0, 2, 0), (0, 3, 1), (0, 1, 0), (1, 1, 1), (0, 0, 0)]

Depth-First Search (Path-Checking)

Implementation

My DFS algorithm is recursive and saves memory by utilizing path-checking instead of memoization. In the starting case, a search node is created with the start state and a Search Solution is created to keep track of node-depth. The algorithm then checks if the node state is the goal state and returns the solution if it is. If it is not, then successors to the node are generated, looped through, and checked to see if they have already been seen in the path they are a part of. If they node has not been seen before, it is passed into dfs. If the goal state is found, then it triggers returns all the way up the stack. If the algorithm makes it back to the start state, then their is not solution. Path checking DFS runs in $O(\min(m^n, mb^m))$ time and $O(m)$ space.

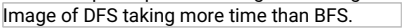
Results

Chickens and foxes problem: (3, 3, 1) Attempted with search method: DFS Number of nodes visited: 12 Solution length: 12 Path: [(3, 3, 1), (3, 1, 0), (3, 2, 1), (3, 0, 0), (3, 1, 1), (1, 1, 0), (2, 2, 1), (0, 2, 0), (0, 3, 1), (0, 1, 0), (1, 1, 1), (0, 0, 0)]

Chickens and foxes problem: (5, 5, 1) Attempted with search method: DFS no solution found after visiting 24 nodes

Chickens and foxes problem: (5, 4, 1) Attempted with search method: DFS Number of nodes visited: 25 Solution length: 18 Path: [(5, 4, 1), (5, 2, 0), (5, 3, 1), (3, 3, 0), (4, 3, 1), (3, 2, 0), (3, 3, 1), (2, 2, 0), (3, 2, 1), (2, 1, 0), (2, 2, 1), (0, 2, 0), (0, 3, 1), (0, 1, 0), (2, 1, 1), (1, 0, 0), (1, 1, 1), (0, 0, 0)]

Discussion Questions

- Does path-checking depth-first search save significant memory with respect to breadth-first search? Draw an example of a graph where path-checking DFS takes much more run-time than breadth-first search; include in your report and discuss.
 - Yes, there is a significant benefit to implementing path-checking DFS over BFS. BFS has to store all of its nodes at every level which creates a memory complexity of b^d (where b is the branching factor and d is the depth). Path checking DFS only has to store the path that it is currently on and thus only has a space complexity of $O(m)$ (m being the maximum depth).
 - Here is an example of path-checking DFS taking much longer than BFS. Path checking DFS runs in $O(\min(m^n, mb^m))$ time while BFS runs in $O(b^d)$ time. 
- Does memoizing DFS save significant memory with respect to breadth-first search? Why or why not? As a reminder, there are two styles of depth-first search on a graph. One style, memoizing keeps track of all states that have been visited in some sort of data structure, and makes sure the DFS never visits the same state twice.
 - No, memoizing DFS does not save significant memory in comparison to BFS. Memoizing DFS runs in $O(\min(n, b^d))$ space while BFS runs in $O(\min(n, b^d))$ space. Since m is the max depth, it can be much larger than d and thus in the worst case, BFS is better for space complexity.

Iterative Depth Search

Implementation

My IDS algorithm uses the previously written DFS algorithm by looping up to a depth limit and passing this value into DFS every cycle. If the depth limit is reached and the answer has not been found, IDS will explore the next level. This makes sure long vines that may not have the goal state in them are not explored fully which avoids a large time cost. Exploring in a DFS manner, level by level, allows for $O(\min(d^n, mb^d))$ time and $O(d)$ space.

Discussion Questions

- On a graph, would it make sense to use path-checking DFS, or would you prefer memoizing DFS in your iterative deepening search? Consider both time and memory aspects. (Hint. If it's not better than BFS, just use BFS.)
 - In a comparison between path-checking IDS, memoizing IDS, or BFS, I think path-checking IDS would make the most sense. In terms of memory, path-checking IDS is by far the best with a space complexity of only $O(m)$, while both BFS and memoizing DFS would be $O(b^d)$ in the worst case. For time complexity, memoized IDS runs at $O(\min(n, b^m))$ which is larger than BFS's $O(b^d)$, but smaller than path-checking IDS's $O(\min(m^n, mb^m))$ time. So if one is looking to prioritize space complexity, path-checking IDS is the clear choice which if one favors reducing space complexity, BFS is the winner.

Results

Chickens and foxes problem: (3, 3, 1) Attempted with search method: IDS Number of nodes visited: 162 Solution length: 12 Path: [(3, 3, 1), (3, 1, 0), (3, 2, 1), (3, 0, 0), (3, 1, 1), (1, 1, 0), (2, 2, 1), (0, 2, 0), (0, 3, 1), (0, 1, 0), (1, 1, 1), (0, 0, 0)]

Chickens and foxes problem: (5, 5, 1) Attempted with search method: IDS no solution found after visiting 2300 nodes

Chickens and foxes problem: (5, 4, 1) Attempted with search method: IDS Number of nodes visited: 1090 Solution length: 16 Path: [(5, 4, 1), (5, 2, 0), (5, 3, 1), (3, 3, 0), (4, 3, 1), (3, 2, 0), (3, 3, 1), (2, 2, 0), (3, 2, 1), (2, 1, 0), (2, 2, 1), (0, 2, 0), (0, 3, 1), (0, 1, 0), (1, 1, 1), (0, 0, 0)]

Final Question

- Lossy chickens and foxes: Every fox knows the saying that you can't make an omelet without breaking a few eggs. What if, in the service of their faith, some chickens were willing to be made into lunch? Let us design a problem where no more than E chickens could be eaten, where E is some constant. What would the state for this problem be? What changes would you have to make to your code to implement a solution? Give an upper bound on the number of possible states for this problem.
 - State = (C, F, E, B)
 - The successor function would have to be changed to allow checking for states in which there are more foxes than chickens (until the E limit has been reached).
 - An upper bound of the amount of possible states is $= (C+1) \times (F+1) \times 2 \times (E+1)$
 - Simply multiply the max possible amount of states for each argument to find the upper bound of possible states

Extra Credit

- For extra credit I implemented a memoized version of dfs. This version is very similar to BFS, but targets successors in a very different order. Memoized DFS runs at $O(\min(n, b^m))$ time complexity and $O(\min(n, b^m))$ space complexity.