CS331, Alaric Hartsock: hartsoca@oregonstate.edu

Programming Project 1 Report

Methodology

Experiments

I used three test cases, represented as (chickens, wolves). The three test cases were (3,3), (11,7), and (100,97). Each algorithm produced different results, except on the first test case which was too simple to highlight any nuance in performance that each search algorithm could bring. In terms of setting up the experiment I created a class called River that held each of the simulation's states. Without getting into too much detail it contained all of the functionality for producing valid iterations of itself, without repeating states and also keeping track of all of its children, its depth, and its parent. Then I implemented breadth first search (BFS), depth first search (BFS), and iterative deepening depth first search (IDDFS) in one function called triple because they are all very similar. In each case we keep a list of the frontier, but the nuance comes with BFS popping off the first element of the list while DFS and IDDFS search pops off the last element of the list. This is, in effect, is a FIFO queue and a LIFO queue respectively. There is also a depth limit for which IDDFS won't expand child nodes beyond. A* search works similarly to BFS, except it sorts its list every time there is a new iteration.

Designing an Effective Depth Limit for Iterative Deepening Depth First Search

The trees that my DFS was generating were incredibly deep. Even for the starting problem state of 3 chickens and 3 wolves, it was generating a depth of 1 with only 4 leaf nodes. Meaning that the vast majority of the nodes had singleton children. I noticed that on average, for every 2 animals added to the initial problem the nodes expanded increased by 6-7. However, the goal depth expanded by approximately 2 nodes per animal added to the initial state of the problem. The equation $D = 2A$ (N = depth of goal state, A = # of animals in starting state). holds true even for a problem of 100 chickens and 99 wolves (395 depth expected, 395 depth received). For a problem of 1000 chickens and 999 wolves, the equation only misses by 3 nodes (depth of 3998 expected, the actual depth of 3995 was actually observed). Because anything larger than this depth limit would be excluding the goal node, the optimal solution is to define the depth limit by an equation rather than a constant. A constant depth limit would not be optimal nor complete for larger problem sets, and as I'll discuss later the IDDFS algorithm doesn't really make sense for this problem set anyways.

Designing an Admissible A* Heuristic

My first attempt at making an admissible A* heuristic were overly optimistic and I experimented with adding constants, but I found a better solution. It is surprisingly easy to accurately estimate the depth of the goal node for a problem set. As mentioned before in Designing an Effective Depth Limit for Iterative Deepening Depth First Search, multiplying the number of animals in the initial state by two provides an accurate estimation of how deep the goal node will be. So I simply subtracted the current nodes depth from this value to get an A* heuristic which is admissible and accurate. I also subtract approximately 2% from the heuristic just to be sure that it doesn't overestimate the cost to the goal and remains optimistic/admissible. It should be noted that this simple change, subtracting 2% from the overall heuristic value, decreased nodes expanded by as much as 25%.

Results

BFS Results

Case	Nodes Expanded	# Nodes in Solution Path
(3,3)	15	11
(11,7)	33	98
(100,97)	982	391

DFS Results

Case	Nodes Expanded	# Nodes in Solution Path
(3,3)	15	11
(11,7)	37	94

(100,97)	1163	395

IDDFS Results

Case	Nodes Expanded	# Nodes in Solution Path
(3,3)	15	11
(11,7)	37	90
(100,97)	979	395

A* Results

Case	Nodes Expanded	# Nodes in Solution Path
(3,3)	15	11
(11,7)	33	73
(100,97)	784	391

Discussion

Originally I was surprised because my results were coming in remarkably close. A*, and BFS were both producing similar results. Then I modified by heuristic by reducing it by 2% and my number of nodes expanded shrank by almost 25% immediately. It turns out that I was slightly overestimating the heuristic, and that was enough to make my search algorithm not competitive with the others. Once I made that change, my results make a lot more sense. BFS, DFS, and IDDFS are all relatively similar in finding the goal state but A* expands far fewer nodes.

Another thing that I found interesting was just how linear the trees were. When I visualized the trees generated by these search algorithms very few had many leafs. This most likely reduced the amount of divergent performance that each search algorithm displayed, as each search algorithm was more limited and following a discrete and optimal path to the goal node. Never the less, performance changed were still observed in each of the searching algorithms.

Conclusion

What can you conclude from these results?

I can conclude that there are many ways to search the trees for the chickens and wolves problem, each with their own nuances. While I'll discuss which algorithm had the best performance in the next section of this conclusion, I think it's interesting how tree structure relates to the optimal searching algorithm to use. Because the trees generated in this problem are mainly linear with not many leaf nodes, one might assume that DFS is the best searching algorithm for the problem. However, BFS consistently outperforms DFS in this situation. Nothing is absolute but we can conclude that in this case, out of BFS DFS and IDDFS that BFS is probably one of the best searching algorithms for this problem.

Which search algorithm performs the best? Was this expected?

The best performing algorithm is by far A*. In each test case, A* consistently outperformed its peers in reducing both the number of nodes expanded and number of nodes in the solution path. This wasn't unexpected by any means as the lectures described A* as one of the best searching algorithms, but what was surprising to me was just how much a small change in the heuristic could change A*'s performance. Simply reducing its value by 2% so that it wasn't overestimating goal node cost improved performance by 25%, which was a shocking reduction. If I ever have to do a similar search then I'll definitely consider using A* to search a tree.