Authors: Juichi Lee and Michael Boly
Spring 2022
CS 331 Intro to Artificial Intelligence
4/18/2020

Programming Assignment #1: Uniformed and Informed Search

Methodology

We ran the following experiments on the three test cases:

	Input	Output
Case 1	(0,0,0) (3,3,1)	(3,3,1) (0,0,0)
Case 2	(0,0,0) (11,7,1)	(11,7,1) (0,0,0)
Case 3	(0,0,0) (100,97,1)	(100,97,1) (0,0,0)

These were the parameters we considered for each algorithm in each experiment:

Algorithm	Parameters
BFS	FIFO queue: Never stop until we find it, or we exhaust all options.
DFS	LIFO queue: Never stop until we find it, or we exhaust all options.
IDDFS	Depth Limit: 0 - INT_MAX
A*	Heuristic: Adding the number of chickens and wolfs left on the Right Bank

<u>BFS:</u> To implement the Breadth-First Search algorithm, we began with the general format of the graph search pseudocode. First we initialize the frontier using the initial state of the program and an empty explored set. Then it performs a loop that does the following: checks if the frontier is empty, chooses a node to remove, checks if it contains the goal state, adds the node to the explored set, expands the chosen node, and adds its child nodes to the frontier if not already in the frontier or explored set. The unique aspect of the BFS algorithm is that it uses a LIFO queue in order to choose the next leaf node from the frontier.

<u>DFS:</u> We implemented the Depth-First Search algorithm by substituting the BFS's FIFO queue for a LIFO queue. This ensures that the nodes that have been recently added to the frontier queue get expanded first, therefore traversing the depth of the tree instead of the breadth.

<u>IDDFS:</u> We implemented the Iterative Deepening Depth First Search algorithm in a similar fashion to the DFS algorithm, but this time, we introduced a depth limit that stops the search. The IDDFS algorithm runs DFS first with a depth limit of 0. After each run, we increment the depth by 1 until the solution is found or the frontier is exhausted.

<u>A*:</u> We implemented the A* algorithm by creating a heuristic, which is a guess of sorts to approximate the cost of navigating from the current node to the goal state. The heuristic we chose was to add the number of chickens and wolves that are left on the Right Bank. We chose this as the heuristic because it is admissible (will never overestimate the cost to the goal node) and it is simple to calculate. The A* algorithm uses this heuristic calculation, as well as a calculation from the start node to the current node (g), to calculate f ($f = g + h$). Lastly, A* uses a priority queue to choose the leaf node from the frontier with the smallest f value as the next node to explore.

Results

The number of nodes on the solution path for each algorithm (includes the initial state):

	BFS	DFS	IDDFS	A*
Case 1	12	12	12	12
Case 2	34	38	38	34
Case 3	392	396	396	392

The number of nodes expanded for each algorithm in each case (not including goal state):

	BFS	DFS	IDDFS	A*
Case 1	14	11	77	14
Case 2	96	58	1769	96
Case 3	980	865	375410	980

Discussion

The results were largely as expected based on the space and time complexity of each algorithm. However, there was some interesting behavior we found in comparing each algorithm on these three test cases. We know that the time complexity is represented by the number of nodes generated during the search, while the space complexity is a function of the maximum number of nodes stored in memory. We expected BFS and DFS to have similar time complexity. This is mostly true although in the largest test cases it becomes clear that DFS is performing better in the number of nodes expanded than BFSs.

We expected DFS to use less memory than BFS. However, for most cases the number of nodes on the solution path was basically the same. In fact, there were even cases where BFS had slightly less nodes on the solution path than DFS. This could be a result of where the goal state was located in the tree that enabled BFS to find it just a few nodes earlier. Furthermore, we expected IDDFS to expand significantly more nodes than the other algorithms, and this was certainly true. This makes sense because IDDFS is running DFS so many times in order to reach the goal node, so many of the same nodes need to be expanded in different iterations. Additionally, we expected A* to expand fewer nodes than BFS. However, A* expanded exactly the same amount of nodes as BFS for all three test cases.

It was really interesting to see that A* was very similar to BFS in terms of the number of nodes on the solution path and the number of nodes expanded. This could be because they both seemed to produce the optimal solution path in each test case.

Conclusion

From our results, we conclude that both the BFS or A-star algorithm are the best for finding the optimal solution path. Both are identical in terms of number of nodes on the solution path and number of expanded nodes. Moreover, they offer the most optimal solution path while only slightly lagging behind the DFS algorithm. These results were slightly surprising since we expected A-star to perform the best, however, BFS matched it in performance.

In terms of speed, however, we conclude that DFS performs the best. Even though it does not provide the most optimal solution out of all 3 test cases, it ran significantly faster than all of the other algorithms in test case 2 and 3 which were more computationally complex. We did not expect DFS to be the fastest and believe its speed may be due to the way the problem is set up.