Uninformed and Informed Searches: Wolves and Chickens Puzzle

Methodology:

The wolves and chickens puzzle is represented by states (i.e. nodes in the search tree) using C++ classes and structs. A 'state' struct holds information about the state of a game, including the number of chickens and wolves on the left and right banks and the side of the river the boat is on (a boolean - false for the left bank and true for the right). Furthermore, a 'gameState' class is defined by a state, a parent gameState, five children gameStates (as there are potentially five actions from each state), and a depth. With the data structures implemented, the search algorithms utilize these structures and their functionalities. Each search accepts three parameters, the initial state of the game, the goal state of the game, and the file path to output results to - all passed as command-line arguments into the program execution. For all searches, two maps store explored nodes and nodes currently on the frontier, which are checked against before adding nodes to the frontier. Node expansion generates all five successors of the gameState, adding each to the state's 'children' array. Successors will be NULL if either a losing state or invalid.

All search functions follow the traditional graph-search pseudocode. Breadth-first search (BFS) and depth-first search (DFS) do not require any further adjustments other than changing the frontier implementation from a queue to a stack (respectively). For iterative-deepening depth-first search (IDDFS), a "for loop" iterates from zero to the maximum integer value, with each iteration of the loop representing an incremented maximum depth for the DFS being performed within. Upon removing a node from the frontier - if not a winning state, the node's depth is compared against the maximum depth (must be less than) before it is expanded to ensure that depths beyond the limit are not explored in the current iteration. Additionally, non-goal-producing expansions (if a node and all its descendents are non-goal-producing) are removed from the explored and frontier maps (thus eliminating the possibility that a shorter path is not followed as DFS progresses). For A* search, the heuristic is based on the idea that if the rules of the game are relaxed so that the game is won by bringing animals from one side to another - regardless of whether there are more wolves than chickens - the heuristic will underestimate the actual number of moves it takes to win the game. Either the rules of the game force more moves than the heuristic due to the wolves outnumbering chickens restriction, or the heuristic is the optimal value if the wolves and chickens do not restrict the decision (for example if there were only chickens to take across the river). Following this logic, the heuristic calculated how many animals remained to-be moved (as only one animal is taken at a time considering one must be taken back after each trip across the river). To account for the edge case where there are only two animals left, the heuristic returns 1 when there are only two animals remaining. The heuristic utilizes an operator overload function for the "<" operator that executes f(n) = g(n) + h(n) for two different states and compares the values. The overload enables the proper functioning of the priority queue in the A* search function. With all the search algorithms implemented, a function prints the results to the terminal and writes to the output file. The program is executed with <initial state file> <goal state file> <mode> <output file> as command line arguments.

Results:

Nodes Expanded for Each Algorithm:

Test Case	BFS	DFS	IDDFS	A*
1	14	12	85	11

2	96	34	1571	90
3	980	392	208863	974

Solution Path Length for Each Algorithm:

Test Case	BFS	DFS	IDDFS	A*
1	11	11	11	11
2	33	33	33	33
3	391	391	391	391

Discussion:

Through playing the game, it was determined that the optimal path length for the first test case is 11. This matched the output for the bread-first, iterative deepening, and A* searches - and the path length is consistent across the other test cases. However, to see that DFS also arrives at an optimal path length for each test case is surprising. DFS is not an optimal algorithm, as it can expand longer paths before it expands the optimal one. The anticipation was that DFS would act sub-optimally in some of the test cases, but as this is not the case, the test cases giving optimal results is an indication that the goal path traversed first happens to be an optimal path.

As far as the number of expanded nodes for each algorithm, the results mostly mirrored what is expected. How few nodes were expanded during DFS was somewhat surprising, but this relates to the conclusion made in the path length discussion. A* was expected to expand the fewest nodes, and although it expands more than DFS, it performs better than BFS and IDDFS. The number of nodes expanded for BFS is sensible, as it is a little higher than A* but still reasonable given a branching factor of two or three on average and the increasing relationship between branches explored and test case complexity. By far the most nodes were expanded by IDDFS, which is expected as the algorithm has to run DFS at many depths before reaching the depth of the goal. However, it is surprising to see the number grow so quickly relative to the other cases. Given the results, it would unexpectedly appear that an IDDFS becomes infeasible quicker than other algorithms.

Conclusion:

Based solely on results, DFS appears to be the best algorithm. However, the optimal performance of this algorithm is not solely determined by its effectiveness, but also by some degree of luck that we found the optimal path as early on in our search as we did. Thus, when considering the best algorithm, DSF is not ranked as well as the results indicate. On the other hand, BFS expands fewer nodes while maintaining optimality in comparison to the IDDFS. Consequently, iterative deepening search executes slower and is presumed to be less effective with larger sample sizes than BFS is. Space complexity is IDDFS's biggest advantage over BFS, but since there are no observed issues with memory, BFS appears superior to IDDFS. However, the best algorithm is found to be A* search. A* finds the optimal path and expands fewer nodes than BFS or IDDFS. Thus, it runs quicker and is more effective in high complexity test cases. This is expected, as A* is optimally efficient and guarantees that no other optimal algorithm will be better than it for all state spaces - even if there are some edge cases where a certain algorithm performs better, as seen in this experiment with DFS.