Chetan Dindukurthi
WeiHao Kuang

**Methodology:**

**BFS:**
For the experiments that we run on the algorithm, we simply ran them as test cases, we obtained the initial and goal states from the files and passed them to the BFS algorithm. Our algorithm displayed the number of the nodes expanded and the solution path for the optimal solution. For our implementation of this search algorithm, we use the graph version which includes a closed list that checks to see if the nodes are traversed are repeated states. We create a string out of the all of the states and use that as the key and store the nodes into the hash-table that represents the closed list. All of the nodes in the fringe are stored in queue. Whenever we pop a node off of the fringe and expand we check them to see of they already exist in the hash-table and if they do they will not be placed on the fringe. In our algorithm, the queue does all the work with keeping track of the successors states of nodes to be expanded the extra bookkeeping for repeated states be handled with will be closed list.

**DFS:**
For depth first search, we ran the experiments like what we did for breadth first search. We got the initial and goal states and passed them to the DFS algorithm. We display the number of nodes expanded and solution path. For depth first search, our implementation uses a stack for storing the fringe nodes which creates the depth first behavior as opposed to the breadth-first behavior with a queue. We use a hash-table (closed list) to keep track of the nodes that we have visited so that repeated states are ignored and they aren't explored multiple times. So whenever we pop a node off of the fringe we expand it and check that against the hash-table to check for repeats so we ignore repeated states.

**IDDFS:**
For this algorithm, we ran the same experiments that we ran on BFS and DFS. Initial state and goal states are read from a file and passed on to the algorithm. We display the solution path and the nodes expanded. For this algorithm, our implementation is a still the same as the DFS, but the main difference here is that we will have a parameter *l* that limits how deep we go on each iteration of the loop. Once *l* is reached and all the nodes are expanded, *l* is incremented, and the above process is repeated until no more nodes are left to expand or if a goal node is found or the maximum depth has been reached. The closed list for this algorithm will need an extra step tracking section to handle the repeated states of different depth, and we favor the states that were found at a shallower depth and ignore the same states at a deeper depth. For this implementation, we choose the max depth to be 1000 and we initialize *l* as 0 and increment by 1 each iteration. The max depth can be changed for other test cases that require it.

**A*:**
For this algorithm the experiment are conducted in the same manner as DFS, BFS, and IDDFS. For the algorithm, the fringe will be made using a priority queue that will sort the nodes on the fringe in the order of the path cost since they are no longer uniform like it was for BFS, DFS, and IDDFS. The path cost will be determined by the heuristic function which will be $((N-2)*2)+1$. This function will take the current node and compare it to the goal state and estimate the cost that it would take to get to the goal state from the current state. We created this heuristic formula by relaxing the rule of having more wolves than chickens as this is a good estimate of how many moves it takes to get to the goal state and then counting how many moves it would take to bring all the animals to the goal state. This heuristic is admissible as it always underestimates the actual number of moves (the number of moves with the rule not relaxed) needed to get to the goal state.

Chetan Dindukurthi
WeiHao Kuang

**Results:**

| **Algorithm** | **Test Case 1** | **Test Case 2** | **Test Case 3** |
|---|---|---|---|
| **BFS** | Nodes Expanded: 14<br>Path Length: 11 | Nodes Expanded: 66<br>Path Length: 39 | Nodes Expanded: 1,344<br>Path Length: 387 |
| **DFS** | Nodes Expanded: 11<br>Path Length: 11 | Nodes Expanded: 57<br>Path Length: 39 | Nodes Expanded: 1,230<br>Path Length: 953 |
| **IDDFS** | Nodes Expanded: 105<br>Path Length: 11 | Nodes Expanded: 1,691<br>Path Length: 39 | Nodes Expanded: 8,666,888<br>Path Length: 387 |
| **A*** | Nodes Expanded: 14<br>Path Length: 11 | Nodes Expanded: 64<br>Path Length: 39 | Nodes Expanded: 1,340<br>Path Length: 387 |

**Discussion:**
The results that we obtained from the experiment were overall expected from the time and space complexities of the algorithms. We expected that DFS would not always return the optimal solution, especially in larger state spaces. We expected that that IDDFS would generate a lot more nodes than the other algorithms as it is repeating nodes it already created, even for small state spaces. We found that A* is more efficient that BFS, it generates around the same number of nodes, which is interesting but makes sense when you compare the time and space complexities for BFS and A*.

**Conclusion:**
From the results that we gathered from the BFS, DFS, IDDFS, and A* algorithms, we can see that the DFS algorithm expands the least amount of nodes for the 3 test cases. However, this algorithm does not guarantee that it will find the optimal path. You can see this when you compare the results for DFS for test case 3 and compare those results to the results of the other algorithm. Overall, we can conclude that the A* is the best as it expands the least number of nodes (if you exclude DFS). This is expected as this algorithm can "look ahead" in the graph by estimating the potential future path cost and always choose the lowest path cost this way. This is in contrast to the other algorithms as the other algorithms do not "look ahead" in the graph.