Methodology:

For all of the searching algorithms we sent the function two game states. The first state represented the initial configuration and the second state represented the goal state. We implemented a game state using a python object, which contained two shores, a pointer to its predecessor, its depth and some functions such as “ok” . Each shore contained a number for wolves, for chickens and for the boat. The “ok” function determined if a state was valid in the game.

For each searching algorithm we tested it on 3 test cases. Each test case increased in size of the problem (increasing number of sheep and wolves).

Test Case 1:

Initial:	Right Bank:	3, 3, 1	(3 wolves, 3 sheep, 1 boat)
	Left Bank:	0, 0, 0	
Goal:	Right Bank:	0, 0, 0	
	Left Bank:	3, 3, 1	

Test Case 2:

Initial:	Right Bank:	9, 8, 1	Goal:	Right Bank:	0, 0, 0
	Left Bank:	0, 0, 0		Left Bank:	9, 8, 1

Test Case 3:

Initial:	Right Bank:	100, 95, 1	Goal:	Right Bank:	0, 0, 0
	Left Bank:	0, 0, 0	Left Bank:		100, 95, 1

Breadth-First Search:

We used a generic breadth-first search implementation, exploring one level before moving on to the next level. This function utilized a LIFO queue to generate new nodes to explore. We checked the validity of the node and if the node was previously visited before adding it to the queue.

Depth-First Search:

We used a generic depth-first search implementation, exploring one path before moving on to another path. This function utilized a FIFO queue to generate new nodes to explore. We checked the validity of the node and if the node was previously visited before adding it to the queue.

Iterative Deepening Depth-First Search:

We used a generic iterative deepening depth-first search implementation, exploring one path to a certain depth before moving on to another path. This function utilized a FIFO queue to generate new nodes to explore. We checked the validity of the node and if the node was previously visited before adding it to the queue. The function also checked to make sure the node was at an appropriate depth before adding it to the queue. Our overall depth limit was infinite.

A-Star Search:

We used a generic A-star search implementation, exploring the node with the lowest heuristic value first. This function utilized a priority queue to generate new nodes to explore. The heuristic function we developed was based off the number of items (wolf, chicken, boat) that were out of place (not matching the goal state). For example, on test case 1, the initial state would have a heuristic value of 7 because all items are out of place. This heuristic is admissible because it will never overestimate. Since the boat has to travel back from the left bank with at least 1 animal, it will take some animals at least 3 moves to end up in the correct spot.

Therefore, a count of how many objects are out of place will underestimate the number of moves left.

Results:

Test Case	BFS	DFS	IDDFS	A*
1	14 nodes expanded 11 nodes in solution	11 nodes expanded 11 nodes in solution	84 nodes expanded 11 nodes in solution	13 nodes expanded 11 nodes in solution
2	54 nodes expanded 31 nodes in solution	42 nodes expanded 31 nodes in solution	889 nodes expanded 31 nodes in solution	36 nodes expanded 31 nodes in solution
3	1344 nodes expanded 387 nodes in solution	900 nodes expanded 485 nodes in solution	350866 nodes expanded 483 nodes in solution	676 nodes expanded 387 nodes in solution

Discussion:

For the most part, our results were as expected. We found that for the first test case, the smallest, all of the methods expanded roughly the same number of nodes (except for the iterative deepening depth first search, which was much higher than the others for each case). Depth first search expanded fewer nodes to find a solution, but it wasn't always optimal. A* seemed to be the best overall pretty much every time, especially as the size of the search space increased. The one interesting thing we noticed with our results was that IDDFS gave us a slightly more optimal solution on the 3rd case than DFS.

Conclusion:

We can conclude that these algorithms are as expected - BFS and A* are optimal, while IDDFS and DFS aren't always. Also, while BFS is optimal, DFS runs faster in most cases. IDDFS is also significantly slower for the size of problems we were computing, but may scale better with problems with greater branching factors. From these results, it is clear that A* is the best search algorithm. Although it performed about as well as the others on the smaller test case, it did vastly better than the other algorithms in larger search spaces. This is pretty much what we were expecting based on lecture.