Methodology:

BFS:

```
static Node* BreadthFirstSearch:    GameState initial,
                                    GameState goal,
                                    map<string, Node*>& explored,
                                    map<string, Node*>& frontier,
                                    int& expandedCount
```

The implementation of the breadth first search included the initial and goal states. These “GameStates” were implemented as a class. This GameState class was defined in terms of two banks, a left bank and right bank. Banks were implemented as a class as well. These classes were defined by the number of chickens, wolves, and the boat. The search functioned by exploring nodes starting with a root node which was derived from the initial state. Nodes are evaluated in a FIFO (first in, first out) order. Nodes that were expanded were added to the explored list, while new child nodes were added to the frontier list during expansion. After the search is complete, the function returns the goal node which contains a way to navigate back through parents via pointers. If there is no solution the function returns a null node (which is a predefined constant node).

DFS:

```
static Node* DepthFirstSearch:
                                    GameState initial,
                                    GameState goal,
                                    map<string, Node*>& explored,
                                    map<string, Node*>& frontier,
                                    int& expandedCount);
```

Depth first search was implemented almost exactly identical to breadth first search. The main difference was that instead of FIFO, a LIFO (last in, first out) method was used to expand nodes. This means that the most recently seen node will be expanded or evaluated first. We can see that the function definition is identical to BFS. Both BFS and DFS make calls to a Expand() function which takes the node to expand and adds the valid children of that node to the frontier.

ID-DFS:
static Node* IterativeDeepeningDepthFirstSearch:

```
                                                    GameState initial,
                                                    GameState goal,
                                                    map<string, Node*>& explored,
                                                    vector<Node*>& nodes,
                                                    int& expandedCount
```

ID-DFS uses a slightly different method of keeping track of nodes. It just uses a simple vector of nodes to keep track of all nodes that are created. It also makes use of an explored map which makes it easy to check if a node has been explored or not. Due to the nature of the search, it was simpler to just wipe out the list each time and then rerun the search. This function makes a call to another helper function which handles the recursion. This helper function takes the depth limit as a parameter. This function makes use of an infinite while loop to simulate the depth limit approaching infinity. It starts with 0 depth limit and then increases as it hits the cutoff. This means that this algorithm is susceptible to potential infinite looping. I used checks for already explored states and for legal moves to avoid infinite looping and to allow it to reach a solution at all.

Note: For this search, due to its nature as inefficient, the final test (#3) took about 6 seconds to complete on the engineering server.

A-Star:
static Node* AStar:

```
                    GameState initial,
                    GameState goal,
                    map<string, Node*>& explored,
                    map<string, Node*>& frontier,
                    vector<Node*>& nodes,
                    int& expandedCount
```

A-Star uses the graph search method to search through the graph. In order to know which node to evaluate and expand it uses a priority queue initialized so that it is a min heap in order to get the smallest value. A heuristic is used to calculate the order in the queue. The heuristic is the sum of the path from the initial state (the node's path cost) and the heuristic function. The heuristic function is calculated by getting the absolute value difference of all the fields of a state. For example, between two states the fields of each corresponding bank are compared to each other. So, left bank chickens are compared and so are right bank chickens. If there is a difference in any of the values (chickens, wolves, boat), then they are added up to a sum difference and added with the path cost to get the total f(n) score for the function.

Common to All: All searches make use of a couple helper functions. These include checking legal actions, checking if a state is explored, and comparing states and nodes. There are a few more, but they perform maintenance tasks that are trivial for searching.

Results:

TEST #1	BFS	DFS	ID-DFS	A-Star
# Nodes	12	12	12	12
# Expanded	13	11	86	13

TEST #2	BFS	DFS	ID-DFS	A-Star
# Nodes	36	36	36	36
# Expanded	58	48	1106	56

TEST #3	BFS	DFS	ID-DFS	A-Star
# Nodes	388	392	578	388
# Expanded	967	856	313,951	960

Discussion:

I expected that DFS would be better than BFS. This is because it seems like it is more likely for the solution to be deeper in the graph with the more wolves and chickens you have. It was interesting to me, however, that in test #3 DFS had more nodes in its solution, but less expansions. It seemed to me that more expansions would correlate with more nodes in a solution. I also expected that ID-DFS would perform the worst out of all the algorithms. This is because we know that DFS is a subroutine of it. Also, it is intuitive that the amount of time it will have to restart in order to find the solution will severely hinder performance. I did expect A-Star to be significantly better at finding solutions. This is perhaps a side effect of my heuristic function. Based on the results it seems to at least be as good as BFS, if not slightly better as the number of wolves and chickens increase.

Conclusion:

From my results we can obviously conclude that ID-DFS is the worst of all of these algorithms. As for which is the best, it would depend on what was being valued more highly. If the number of nodes expanded is valued highest then it seems obvious that DFS is the best algorithm. If the number of nodes in the solution is of the most value then it seems A-Star or BFS are the best.