ECSE 526
Assignment 1

Sean Stappas
260639512

September 28, 2017

1 For each of these configurations, graph the total number of states explored by your program when using depth cutoffs of 3, 4, 5 and 6, both with minimax and alpha-beta. Assume it is white's turn to play.

The board states for configurations A, B and C can be seen in Figures 14 to 16 of the Appendix. The total number of states explored by minimax and alpha-beta for configurations A, B and C can be seen in Figures 1 to 3. Note that the implementation of minimax was inspired from pseudo-code on Wikipedia [1]. Also, the negamax algorithm was used to implement alpha-beta pruning, also adapted from Wikipedia [2].

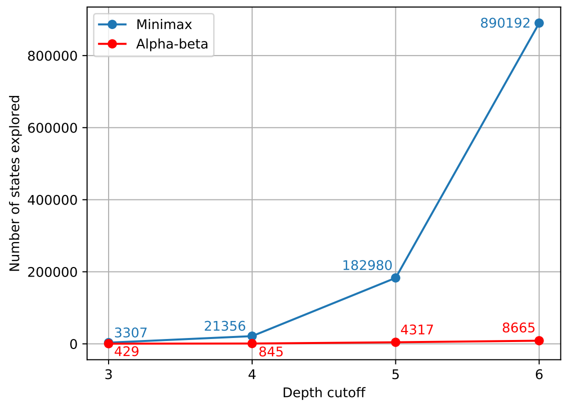


Figure 1: The number of states explored by minimax and alpha-beta starting at the state given by configuration A.

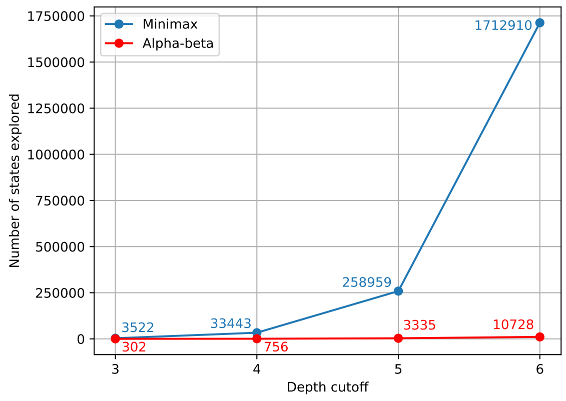


Figure 2: The number of states explored by minimax and alpha-beta starting at the state given by configuration B.

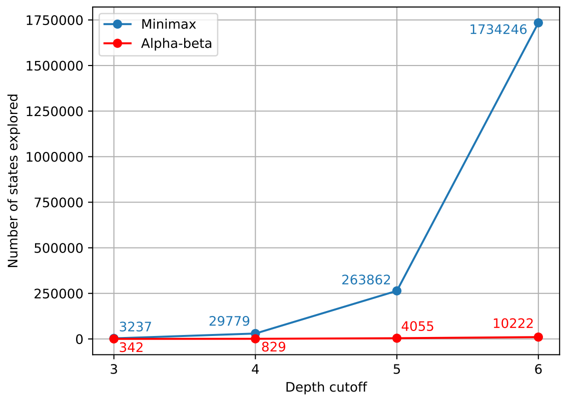


Figure 3: The number of states explored by minimax and alpha-beta starting at the state given by configuration C.

2 Estimate a formula that relates the depth cutoff to the number of states visited for each of minimax and alpha-beta algorithms.

It is expected that the number of states visited should be approximately equal to b^d, where b is the branching factor and d is the cutoff depth. However, the branching factor b is not the same for every state. Indeed, even though there are four directions pieces are allowed to move in, most of the time these directions will be blocked either by other pieces or by the edge of the board. Therefore, the branching factor must be approximated.

Using the results from Question 1, a formula was estimated using a MATLAB script to fit an exponential curve. All the points from configurations A, B and C were combined to fit this curve for both minimax and alpha-beta, as shown in Figures 4 and 5.

The fitting function for minimax is given by the following, where N_m is the number of states explored by minimax and d is the depth:

$$N_m = 24.76e^{1.829d} = 24.76(6.228)^d$$

The fitting function for alpha-beta is given by the following, where $N_{\alpha\beta}$ is the number of states explored by alpha-beta and d is the depth:

$$N_{\alpha\beta} = 20.42e^{1.031d} = 20.42(2.804)^d$$

If we look at the overall complexity of minimax and alpha-beta for the program, we can estimate as follows:

$$N_m = O((6.228)^d)$$

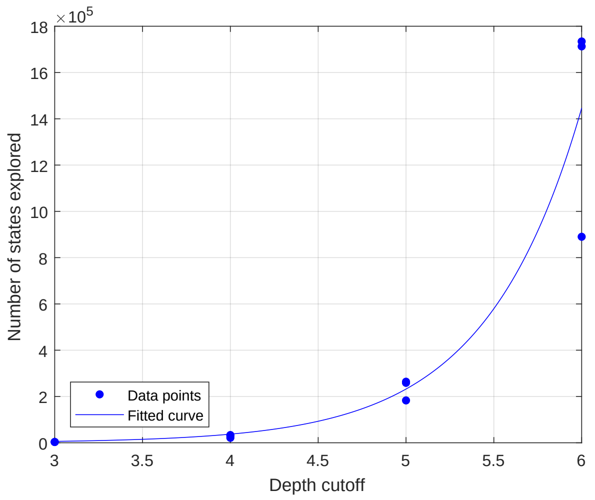


Figure 4: The number of states explored by minimax, with an exponential fitting curve.

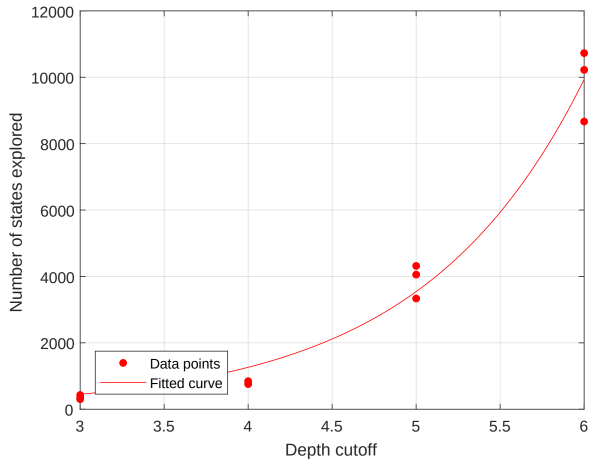


Figure 5: The number of states explored by alpha-beta, with an exponential fitting curve.

$$N_{\alpha\beta} = O((2.804)^d)$$

We can therefore see that the effective branching factor of minimax is $b_m = 6.228$ and that of alpha-beta is $b_{\alpha\beta} = 2.804$, showing that alpha-beta explores much fewer nodes on average.

3 Explain whether the number of states explored depends on the order in which you generate new states during the search. Justify your response using results from your program.

The order in which one generates new states during search affects the number of states explored. The reasoning behind this is that, if one visits states with higher heuristic value first, these states will likely lead to alpha-beta pruning in the next states (assuming the heuristic approximates the true utility accurately enough). Therefore, it is most advantageous to sort the successor states to be generated by the best heuristic value for the current player. In theory, this technique can lead to a complexity of $O(b^{m/2})$ instead of $O(b^m)$ for the search, where b is the branching factor and m is the maximum depth of the search. Even if the ordering is random, a complexity of $O(b^{3m/4})$ can be achieved [3].

In the case of this program, new states are generated in order of best heuristic first. This corresponds to descending heuristic order for the MAX (white) player and ascending for MIN (black). To validate that this order does in fact affect the total number of states explored, three different orderings were applied on the alpha-beta search: *No Sorting*, *Sorting by Heuristic*, and *Random Order*. These three orderings were applied when searching on state configurations A, B and C, as seen in Figures 6 to 8.

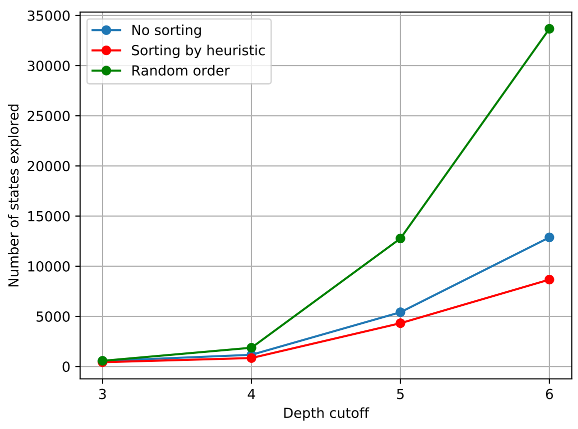


Figure 6: The number of states explored by alpha-beta using various successor sorting strategies starting at the state given by configuration A.

One can see that, despite the theoretical performance increase of random ordering, in practice the random order performs the worst, with sorting by heuristic showing the best performance. Indeed, for all depth cutoffs from 3 to 6 and for configurations A, B and C, sorting by best heuristic value leads to fewer states explored because of pruning.

4 Explain the heuristic evaluation function you used and provide a clear rationale for the factors you included.

Many heuristic functions were tested during the development of the program. Here are some of these functions, with the reasoning behind them:

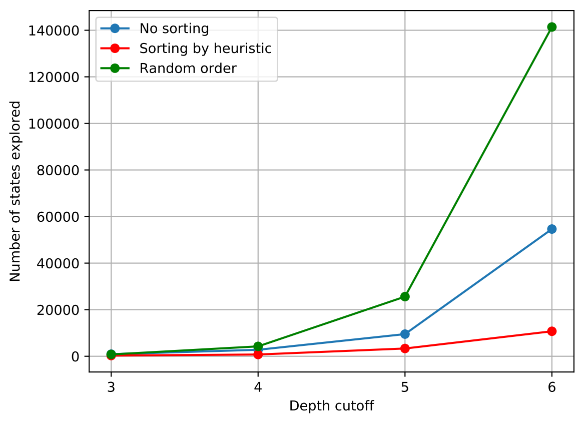


Figure 7: The number of states explored by alpha-beta using various successor sorting strategies starting at the state given by configuration B.

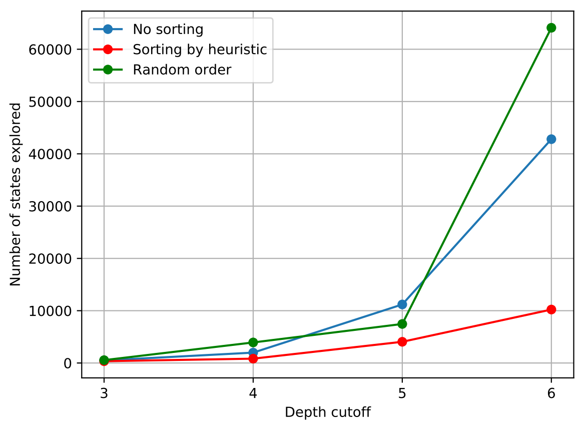


Figure 8: The number of states explored by alpha-beta using various successor sorting strategies starting at the state given by configuration C.

Random returns a random number. This heuristic is used as a benchmark for others.

Win/Loss checks if the player has won or not (four in a row). This is the simplest heuristic, and is only useful if the agent can search deep enough to find a win.

Three in a Row checks if the player has three pieces lined up. Three in a row is the best arrangement besides the winning condition of four, so the agent should strive to achieve these when possible.

X in a Row checks if the player has any pieces lined up, with four being more valuable than three, and three more than two. It is of course necessary to build two in a row before three, so this heuristic helps the agent build up to the winning four.

Manhattan Distance to Center returns the Manhattan distance between each piece and the center of the grid, where distance is given by $|x_1 - x_2| + |y_1 - y_2|$. Center presence is important to establish control on the board. The reasoning is that, once enough pieces are close to the center, it should be easier to search for a win, so the early game priority should be to move towards the center.

Weighted Distance to Center returns the weighted distance between each piece and the center of the grid, where distance is given by $(x_1 - x_2)^2 + (y_1 - y_2)^2$. This is an approximation of Euclidean distance, without the square root because of its computational complexity.

Default is the final heuristic used in the program and is a combination of the *X in a Row* and the *Weighted Distance to Center* heuristics. After much testing, these two heuristics were found to be the most valuable. Center presence is crucial to being in control of the game. The *X in a Row* heuristic gives value to three or two pieces in a row, which are necessary to build up to the winning condition of four in a row. The weight given to each heuristic is such that, in the early game, it is more important to go to the center of the board than to form two pieces in a row. Also, forming three or four in a row should be more valuable than simply moving towards the center.

For most of the heuristics above, the heuristic value of a state is given by the white player's value minus the black player value ($white - black$), since white is the maximizing player. However, for the distance heuristics, it is the opposite ($black - white$), since smaller distance to center is better.

5 A more complex evaluation function will increase computation time. Since each move is time constrained, explain whether this favours the use of simpler evaluation functions so that your agent can evaluate nodes deeper in the game tree.

While a complex heuristic evaluation function can enable much more accurate estimations of a state's utility, it can make the search slower when evaluating all the cutoff states. This can be seen in Figures 9 and 10, which show the number of states and maximum depth reached when using different heuristic evaluation functions with different time limits, starting at configuration B.

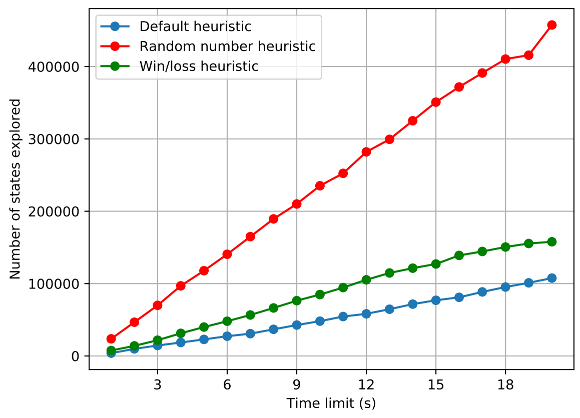


Figure 9: The number of states explored for different heuristics and different time limits, starting at configuration B.

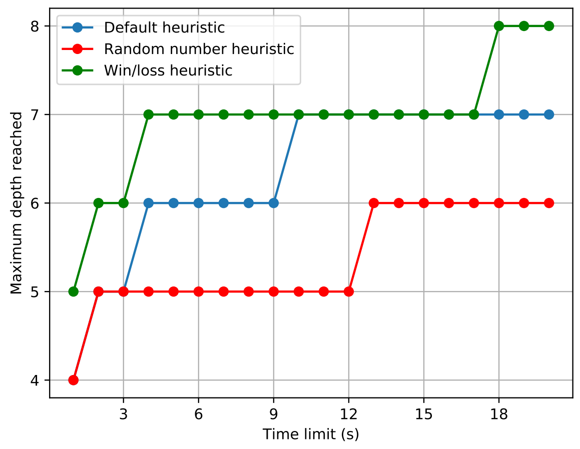


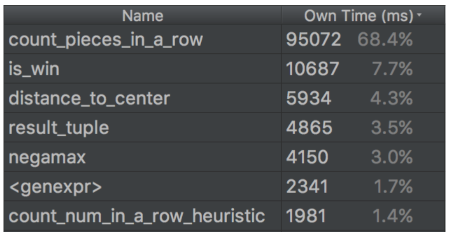
Figure 10: The maximum depth reached during search with different heuristics and different time limits, starting at configuration B.

It can be seen here that the *Win/Loss* heuristic allows for more states to be explored, and thus a deeper search, because of its computational simplicity. The *Random* heuristic also allows for exploration of many more states, because it takes very little time to execute. However, the random heuristic does not lead to deeper search in this case. This is most likely because it does not help when sorting successor states and pruning the game tree.

However, both the *Win/Loss* and *Random* heuristics offer little insight when the search is cut off and the agent has to make a decision on what move to make. Indeed, even though the *Default* heuristic leads to fewer explored states (and less depth), it offers the agent valuable information about which states are likely to lead to a win when search is cut off.

A local AI vs AI game was recorded to show the quality of the *Default* heuristic. The initial state of the game is the default one, shown in Figure 17. The resulting state after 12 moves by each player was saved using the *Default*, *Win/Loss* and *Random* heuristics. The result of using the *Default* heuristic can be seen in Figure 17, where both players have good board presence and can make good moves to lead to a win. When using the *Win/Loss* heuristic, however, both agents exhibited oscillatory behavior, simply picking the first move out of the successors and returning back, since they could not search deep enough to find a winning condition. This led both agents to end up in the same state as the starting position, as shown in Figure 13. Finally, the result of using the *Random* heuristic can be seen in Figure 18, where the pieces for both players are scattered and a win looks distant. These results show that using the more competent *Default* heuristic leads to better performance, despite the shallower search.

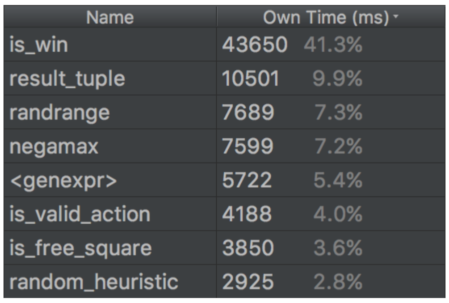
There is additionally a trade-off between memory and time, since a simple fast heuristic will go deeper in the tree and therefore require more memory to store its recursive path when using depth-first search. This depth can however be controlled by using iterative deepening search, which is what is used in the program. Also, faster heuristic evaluation can lead to more states being stored in the transposition table, which increases memory usage. This can be addressed by limiting the number of entries in the table, by using a LRU cache, for example.



Name	Own Time (ms)	
count_pieces_in_a_row	95072	68.4%
is_win	10687	7.7%
distance_to_center	5934	4.3%
result_tuple	4865	3.5%
negamax	4150	3.0%
<genexpr>	2341	1.7%
count_num_in_a_row_heuristic	1981	1.4%

Figure 11: Result of profiling the program in AI vs AI *mode with the default heuristic after a couple minutes of play. It can be clearly seen that most of the execution time is spent in the count_pieces_in_a_row method, which is used to compute the* X in a Row *heuristic.*

The program was also profiled to see what bottlenecks exist when using the *Default* heuristic, as shown in Figure 11. This shows that the main bottleneck is the heuristic function, which takes up approximately 68% of the execution time of the program. If a simple heuristic is used, however, the program doesn't spend as much time computing



Name	Own Time (ms)	
is_win	43650	41.3%
result_tuple	10501	9.9%
randrange	7689	7.3%
negamax	7599	7.2%
<genexpr>	5722	5.4%
is_valid_action	4188	4.0%
is_free_square	3850	3.6%
random_heuristic	2925	2.8%

Figure 12: Result of profiling the program in AI vs AI *mode with the random heuristic after about a minute of play. With a simple heuristic like the random number generator, the program no longer spends most of its time computing the heuristic.*

the heuristic. This can be seen in Figure 12, where the program was profiled using the fast *Random* heuristic, and most of the program execution time (41%) is spent checking for a win. This shows that a more complex heuristic function leads to slower execution.

Therefore, a balance must be found to have a heuristic function that is complex enough to offer valuable information at cutoff states, but simple to explore to a reasonable game tree depth, while not consuming too much memory. This was achieved with the *Default* heuristic.

References

[1] Minimax, Sep 2017. https://en.wikipedia.org/wiki/Minimax.

[2] Negamax, Jul 2017. https://en.wikipedia.org/wiki/Negamax.

[3] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach, Global Edition.* Always learning. Pearson Education, Limited, 2016.

A Board Configurations

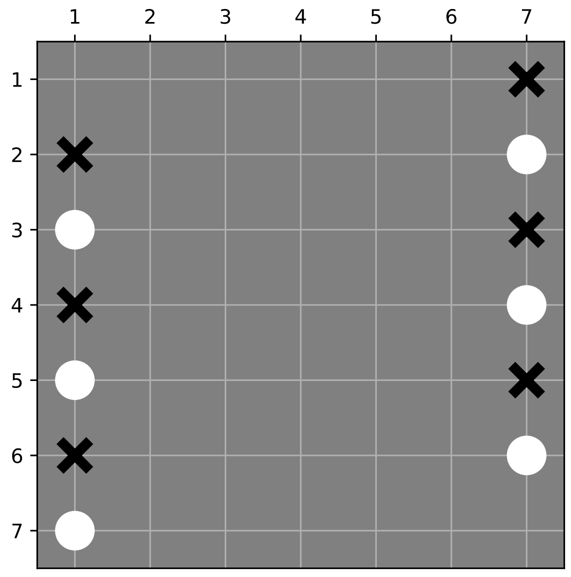


Figure 13: Default initial state.

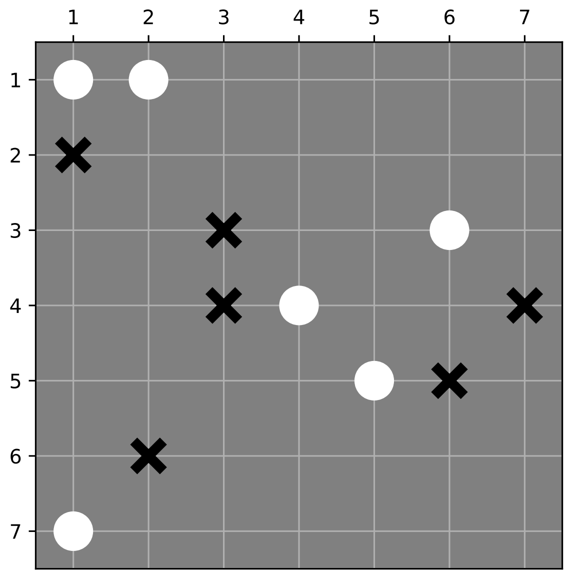


Figure 15: Configuration B.

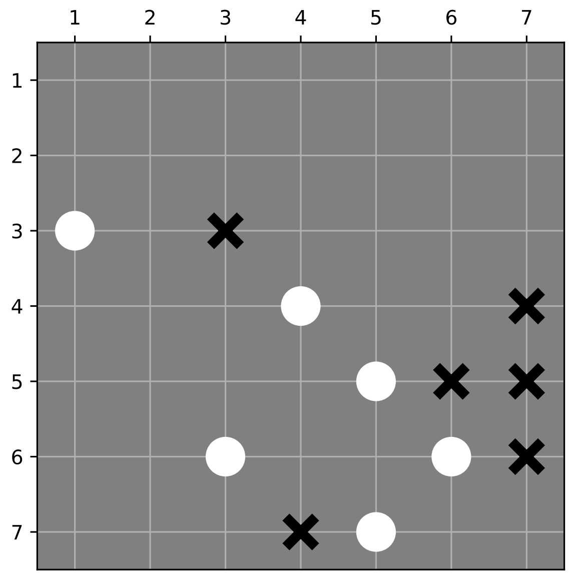


Figure 14: Configuration A.

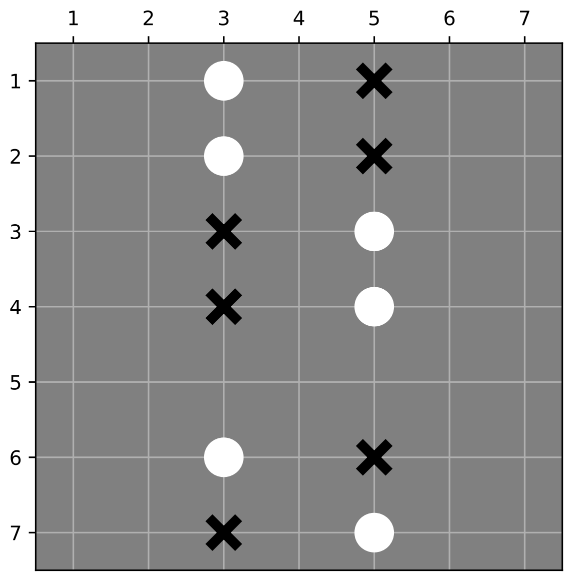


Figure 16: Configuration C.

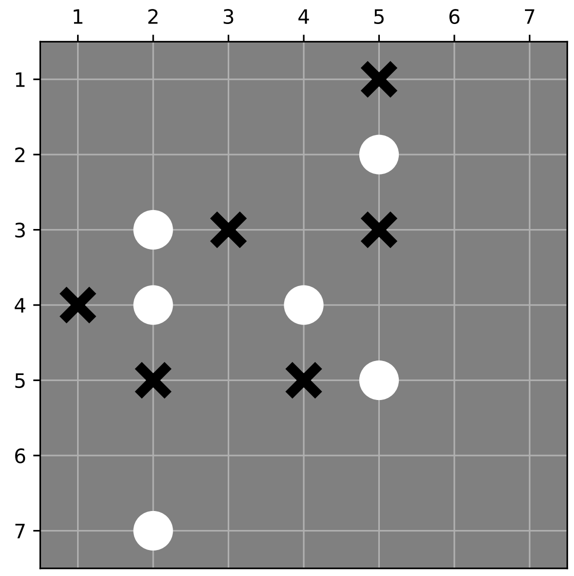


Figure 17: The game state after 6 moves starting at the default initial state, using the Default *heuristic.*

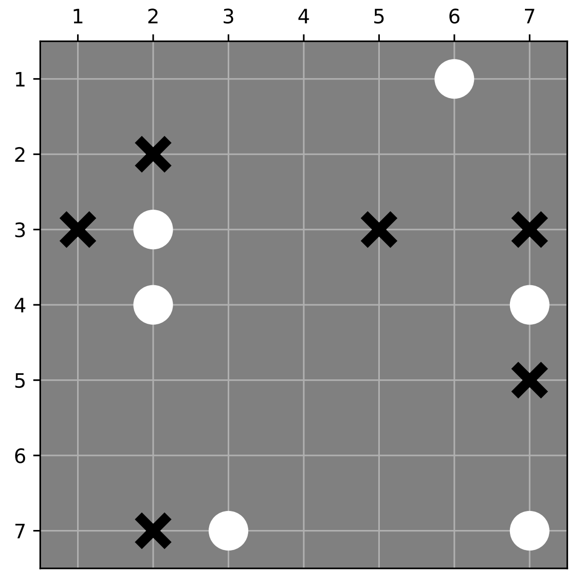


Figure 18: The game state after 6 moves starting at the default initial state, using the Random *heuristic.*