Stockfish? How About Blobfish!

Abstract

In this lab, we were tasked with implementing the basic mechanics for a chess game as well as a H-minimax chess AI to compute moves, which we named Blobfish. We ultimately represented the state of a game using a board containing the positions of the pieces. The state of a board was computed as the difference between the current player's pieces' score and the opponent's pieces' score. We chose two exploration policies: random and score-based, where higher scoring states were visited first. The results showed that the score-based exploration heuristic resulted in much fewer nodes visited, implying that more nodes were pruned off in the search tree using the score-based policy as opposed to the random exploration policy.

Introduction

The high-level problem was to develop an AI that could compute the winningest actions in an adversarial setting. A chess-playing AI must perform adversarial search; an opponent seeks to maximize their score, which directly goes against the AI's goal of obtaining a high score. Thus, we implemented the minimax algorithm for the chess AI, which seeks to perform actions that minimize the maximum loss. It is worthwhile to develop an AI for this task for various reasons. For example, chess players may learn new strategies by studying the moves of a chess-playing AI that consistently wins. Also, the intuition behind exploration policies may be leveraged for AI in other adversarial games/contexts.

Formulation

We represented the problem of implementing a chess-playing AI as an adversarial search problem, which could be "solved" by using an H-minimax algorithm with alpha-beta pruning. The initial state can be any valid configuration of pieces on a chess board. The cut-off tests were 1) an arbitrary depth limit was reached by the minimax algorithm and 2) checkmate/stalemate was reached. The transition model took any valid board configuration as input, *selected a valid action*, and then *output a valid board configuration*. Guaranteeing the validity of actions and successor board states was a point of emphasis for our implementation. We sought to rigorously check that the generated states were valid, thereby improving the time efficiency of our search (less time spent generating successor board states) and improving the branching factor of our search algorithm. In general, the evaluation function is as follows:

$$f(P,\ O)\ =\ (\sum_{x\ \in pieces(P)} x) - (\sum_{y \in pieces(O)} y)$$

where P is the current player (at the current level of the search tree) and O is the opponent (at the same level of the search tree). The Reinfeld scores of the pieces are as follows:

- Pawn: 1 point
- Bishop/Knight: 3 points
- Rook: 5 points
- Queen: 9 points

Additionally, when a king is checkmated, the king becomes worth 1,000,000 points (an arbitrary selection; this number just needed to be higher than the maximum possible number of points one could achieve by capturing all of the opponent's pieces on the board).

The exploration policies we selected were *random* and *score-based*. We believed a random exploration policy could provide a good baseline for comparison for what we hypothesized to be a better exploration policy, the score-based heuristic. In the score-based heuristic, we ordered successor states according to their piece scores in non-ascending order and selected the highest-scoring successor states to visit first. We believed this would result in more states being pruned from the search tree than the random exploration policy would afford.

Additionally, we note that we implemented a variant of H-minimax called H-negamax. The intuition behind this algorithm is that max(a, b) = -min(-a, -b), and the player whose turn it is to move will try to maximize the negation of the value resulting from the move, since the successor position was valued by the opponent. Ultimately, it provided the same result that H-minimax would have, but simply made implementation a bit simpler.

Experiments and Analysis

The random exploration results are averaged over 10 trials.

	Random Exploration	Score-Based Exploration	Optimal Action
Initial State A	4996 nodes	1142 nodes	Rook to E8
Initial State B	1828 nodes	1765 nodes	Bishop to A6
Initial State C	3960 nodes	1419 nodes	Knight to B3

Optimal Successor for Initial State A:

```
_ _ _ _ R _ q k
_ _ _ _ _ _ _ _
_ _ _ _ _ P _ p
_ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _
_ _ _ _ _ _ Q P
_ _ _ _ _ P P _
_ _ _ _ _ _ K _
```

Optimal Successor for Initial State B:

```
_ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _
B _ _ K _ _ _ _
_ p _ _ _ _ _ _
_ _ k _ _ _ _ _
P _ _ _ _ P _ _
_ B _ _ _ _ _ _
N _ _ _ _ N _ _
```

Optimal Successor for Initial State C:

```
_ _ _ _ _ _ _ _
_ _ _ K _ _ _ _
_ _ R _ P _ _ _
_ P _ k r _ _ _
_ _ _ _ p b _ _
_ N _ _ P _ _ _
_ _ _ _ _ _ _ _
_ _ _ _ _ N _ _
```

Regardless of the exploration policy used, since each initial state optimally results in a checkmate for white, the successor state computed by H-minimax was the same. However, for each initial state, the score-based exploration policy had fewer visited nodes (much fewer for initial states A and C) than with a random exploration policy. We believe that this implies many more actions were eliminated by the alpha-beta pruning algorithm when using the score-based exploration policy. This also makes sense, as visiting successor states in which the player whose turn it is to move would have a higher score (more pieces/higher-valued pieces) are intuitively good chess moves to make, and would be harder for the opponent to maximize their score.

Additionally, we verified Blobfish's computed optimal move using the Stockfish chess engine on lichess.com.

Conclusion

We learned that an exploration policy could heavily influence the number of visited nodes in a H-minimax with alpha-beta pruning search. The weakness of our score-based heuristic is that it may perform less effectively in situations where trading a rook (worth 5) for ultimately two knight/bishop pieces (worth 6) is the optimal move (Blobfish would not explore the initial rook trade for a bishop/knight, as it seems unfavorable in the short-term). However, these kinds of situations rarely arise. We believe even better exploration heuristics could be implemented, such as the well-known killer heuristic, which we would have liked to implement given more time. We would also like to have implemented a better-designed codebase if given more time. Besides adding new heuristics, there are other extensions regarding the chess game mechanics that we would like to build, such as using bitboards to represent a board's state.

The extra-credit feature we implemented was to move pawns two spaces under the appropriate circumstances.

Member Roles

David: Implemented logic for determining the validity of king states; namely, whether or not the king was in check and ensuring that a state in which a king moved itself into check could not be generated. Tested and debugged every component. Helped with implementation of negamax. Contributed to representation of States and the board. Also implemented input/output and node counting. Co-wrote the report.

Nick: Implemented move generation for all non-king pieces. Discovered and implemented most of the negamax algorithm. Implemented pruning strategy selection as well as user argument specification and handling. Helped with testing and debugging. Contributed to representation of States and the board. Co-wrote the report. Re-documented all code.