Danmarks
Tekniske
Universitet

01 Department of Applied Mathematics and Computer Science

Board Game Assignment

Thomas Bolander

Made by:
Håkon Collett Bjørgan

1 Prelude

I wrote a paper on Stockfish during the WCC-match between Magnus Carlsen and Fabiano Caruana during the fall of 2018. I chose to continue to explore the topic of computer chess for this exercise. The paper I wrote is included in the zip-file I will hand in.

Exercise 1

The rules of chess

Chess is played on a 8x8 board between two players. At the start of the game each player has eight pawns, two rooks, two knights, two bishops, a queen and a king. The goal of the game is to move your pieces around in such a way that you end up placing your opponents king in checkmate.
The game always starts by a move from the player with the white pieces. The players then alternate between making moves, one after the other. Each type of piece has a predetermined set of legal moves. In addition each piece has a predetermined set of squares they threaten, based on their position on the board. These squares are mostly the same as the ones the pieces are allowed to move to. A rook for instance is allowed to move as far as it wants to either side or forwards/backwards, as long as it does not go through a piece. When a player has placed his opponent's king under a threat the opponent cannot legally move his pieces to get out of, you have placed your opponent in a checkmate, and you have won the game. The players can also agree upon a draw if they think they have no winning chances without risking to lose. The game can also end by a stalemate, which means that a players king is not under a threat, but he has no legal moves. The full rules of chess are easily available online, and can be found here.[1]
In my implementation of the program, all rules except one are consistent with the actual rules. While you should have the choice of selecting which officer you would like to promote your pawn to, my program auto-queens for you. This was done to save some time, as 97% of all promotions are queens [2].

Exercise 2

The game of chess is a two-player, competetive, turn-based, deterministic, zero-sum game with perfect information. As mentioned in Exercise 1, the two players alternate between making moves (one player making one move is know as a ply, and is the term I will be using from here on) and they are both attempting to maximize their own position. With chess being a zero-sum game, maximizing your own position is equivalent of minimizing your opponents position. This property makes the mini-max algorithm a very tempting choice of algorithm to base our AI upon. Furthermore, with chess being fully observable and completely deterministic, the task of finding the best move is purely a task of choosing most wisely while traversing down the game tree. The choices of which branches to explore and where to focus your computing power / brain power is what separates the the average chess player, human or AI-agent, from the good ones.

Exercise 3

From a scientific point of view one can say that chess is a finite and solvable game. If you picture a game of chess as a tree with the initial position as the root node, one can continue to expand the tree for every move made. It does however not take long to realize that this approach yields an unmanageable amount of positions for your program to handle. With an average branching factor of around 35, the tree would quickly become huge. Shannon's number[3] is a conservative lower bound of the number of nodes in a tree representing chess. Its numeric value 10^{120}, is unfathomably large, and rules out the option of brute forcing chess from start to finish. Shannon's number is based on the average number of legal moves in chess game of average length - 80 ply. The actual length of a chess game can be a lot longer, thus the game tree is actually a lot bigger than Shannon's number. the state space complexity, however, is estimated to be smaller than Shannon's number, due to positions occurring multiple times, but with different move orders leading up to them.
With the state space of chess being ridiculously large, it becomes apparent that my algorithm should excel at narrowing down the search as much as possible. After 8 ply from the start of a chess game the number of possible games is 84,998,978,956 [4]. This goes to show that the search tree quickly gets out of hand, and that my algorithm should be selective in which parts of the tree it explores.

Exercise 4

- S_0 : The initial state of the game is as shown on the front page of this paper. Both players have their officers lined up in the back row, with eight pawns in front of them. In more precise terms we have:

  [r,kn,b,q,k,b,kn,r,
  P,P,P,P,P,P,P,P,
  -,-,-,-,-,-,-,-,
  -,-,-,-,-,-,-,-,
  -,-,-,-,-,-,-,-,
  -,-,-,-,-,-,-,-,
  p,p,p,p,p,p,p,p,
  R,Kn,B,Q,K,B,Kn,R]

  which is an array of size 64, representing each tile on the chessboard, and its corresponding .toString() value.

- Player(s): For each state you have a whitePlayer and a blackPlayer. The two alternate between being set as the currentPlayer and opposingPlayer.

- Action(s): For each state the the game iterates over all the pieces of the player set as currentPlayer and calculates the legal moves for each piece. These moves are then added to Collection<Move> legalMoves. This collection then contains all possible actions in the current state.

- Result(s,a): When a player makes a move a TransitionBoard transitionBoard is created, on which the move is made. This board is then passed to the constructor of Board, making a new Immutable board on which the move is made.

  If an action is applied to a state in which it is not a legal action Result(s,a) $\rightarrow$ s.

- Terminal-Test(s): There are two tests implemented in my program: isInCheckMate() and isInStaleMate(). The two serve as the terminal tests to check whether or not a game has reached its end. isInCheckMate() checks if a king is threatened and that there are no legal moves that can prevent the threat. isInStaleMate() checks that your king is not threatened, but all your legal moves leave your king threatened (placing yourself in check).

- Evaluation(s,p): There are several evaluation functions implemented in my program. They will be described later on in this paper.

Exercise 5

The state is mainly captured in the Board class through the use of a list containing 64 tiles and a collection of pieces for each player. The pieces are mapped to their tile by the use of a HashMap ¡Integer, Piece¿. The Board class also keeps track of the currentPlayer and enPassantPawn. Each piece also captures part of the state as they record whether or not their move is their first move, in order to allow/disallow pawn jumps and castling.
Moves are represented as a board + a moved piece and its destination coordinate.
Both moves and state are in other words represented in the same way. The state consists of a board (hashmap with tiles and pieces) and to represent a move we add the moved piece to the equation. In order to develop our AI this is all we need. In order to choose the best move, what we need to know is the state captured in the board and piece class, and that is it. Our AI will then behave as follows: $AI(State) \rightarrow Move$.

Exercise 6

The minimax algorithm lies as a foundation of most chess engines, including mine. It lies as a foundation of many turn-based two player games, in fact. It utilizes the concep tof zero-sum games, which, as the name suggests, are games in which the total score always adds up to zero. In chess for instance, a player can lead by three pawns if and only if the opposing player is down three pawns. The zero-sum property renders the minimax algorithm an effective decision-making tool for making the most advantageous choice. The algorithm calls one player the maximizer and the other the minimizer. They have the corresponding goals of maximizing and minimizing the score. The challenge minimax solves is that of choosing the optimal choice for yourself, given the fact that you know your opponent will attempt to counteract your choices. The game tree in figure 1 shows us how the minimax algorithm makes it's decisions.The algorithm explores nodes down to the leaves or an otherwise predetermined depth. Once these are explored it will maximize or minimize, depending on which depth it is on, and pass its value upwards towards the root node. In the figure you can see the maximizing player being at depth zero, the root node, and having to choose between two paths. In order to determine the best choice a minimax algorithm probes the game tree. At depth one, it would be the minimizing players

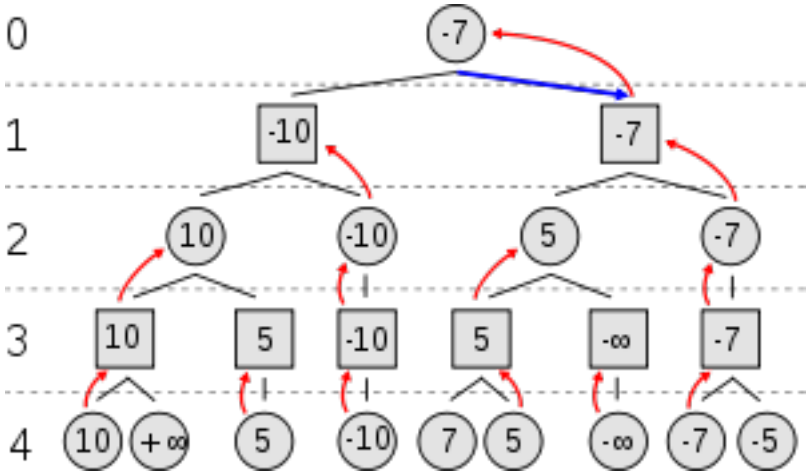

Figure 1: Minimax search tree

turn, hence the nodes at this depth minimize the values from its children at depth two, which have maximized the values of its children at depth three, which have minimized the values of its children at depth four. In the end, the maximizer at our root node is left with a choice between -10 and -7, which he of course maximizes, and chooses the rightmost path of the game tree. The algorithm works in exactly the same way in my game as it does in the example. The states in my implementation are given numeric values by evaluation functions, thus being represented the same way as the nodes in figure 1. However the game of chess has a much larger branching factor than the one on figure 1, so for it to be more realistic, each node should have about 35 children, insted of one or two in figure 1.
I would say the minimax-algorithm is as appropriate as one can find an algorithm to be. It captures exactly the essence of how one should think in order to beat your opponent in the game of chess r any other zero-sum game. It also captures how human players think when playing chess. "Which move can I make that leaves my opponent with the worst available moves?"

Exercise 7

I score the position through several evaluation functions, each of which is added together to form a score for the player. The player's scores are then subtracted $whitePlayer - blackPlayer$ rendering a final score for the board. Score>0 indicates that my AI thinks white has the upper hand, and Score<0 indicates that black is in the lead.
The evaluation functions are as follows:

- pieceValue(player): Iterates over the active pieces for each player and sums up the piece values. The value for the pieces are set as pawn=100, knight=300, bishop=300, rook=500, queen=900, king=10000. These numbers are based on information found at online chess programming communities.

- mobility(player): Returns player.getLegalMoves.size(), which basically gives a small reward to the player who has the most legal moves i. e. the most mobility. this is not weighted, but could

very well be.

- castled(player): returns a bonus of +60 if a player is castled. To castle is generally considered to be a good move as it brings your king to safety.
- check(player): Returns a bonus of +50 for the player having checked the other players king.
- checkMate(player): returns a bonus of +10000 for the player having checkmated the other players king. This is set to be so large that it will trump all other states if found.

Exercise 8 and 9

There are a lot of improvements that can be made to my AI. I have barely made it work on the lowest level of what we can call an AI. All of the parameters can be fine tuned, such as the piece values, check bonus, castled bonus and so on.
There are also a lot of other parameters that can be included into the evaluation functions, that are not included at the moment. Such parameters include weighting the piece values based on the position on the board. A piece is generally more valuable if it is placed in the center, contrary to on the rim of the board. Pawn structure is another parameter that is neglected in my AI. The bishop pair is not given any thought in my AI. Connected passed pawns are not given extra value. In other words the gap between my AI and the top chess AIs is quite vast.

It is however not in the evaluation function the biggest room for improvement is, in my opinion. I have currently only implemented the basic version of the minimax algorithm. In order to improve my engine drastically I at least need to implement Alpha-Beta-pruning and iterative deepening. With this beeing said, I do believe that the foundation is one to continue to work on. To the best of my knowledge, the minimax algorithm is the most suitable for zero-sum-problems. Currently the AI is hardcoded to search at depth 4, but it should be able to adjust its search depth according to the time it has left and so on. Regarding the data structure, I know that the top enigines use bitboards[5] as their data structure to represent the board, so in order to compete with them in the future, I assume that I will have to look into that.

Finally I would like to say that I wish I had more time to actually test the AI and work on that, rather than the implementation of chess itself, which has taken most of the time I have spent on this project. But you live and learn, and the AI I have made did at least beat me, so it's not useless, although I estimate its ELO rating to be no higher than 1200-1300 at the most.

References

[1] Rules of chess. https://www.chess.com/learn-how-to-play-chess.

[2] Pawn promotion. https://en.wikipedia.org/wiki/Promotion_(chess).

[3] Shannons number. http://archive.computerhistory.org/projects/chess/related_materials/text/2-0%20and%202-1.Programming_a_computer_for_playing_chess.shannon/2-0%20and%202-1.Programming_a_computer_for_playing_chess.shannon.062303002.pdf.

[4] Number of games. https://en.wikipedia.org/wiki/Shannon_number.

[5] Bitboards. https://www.chessprogramming.org/Bitboards.