**COSC 3P71 Term Project**

**Chess AI System**

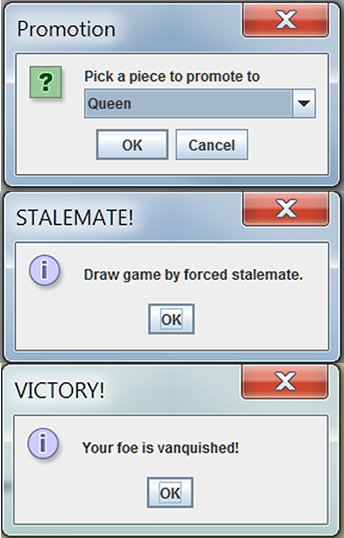
Taras Mychaskiw Lachlan Plant

**1 Overview**

Chess is a two player strategic adversarial game. The two players alternate between each other moving pieces, within certain constraints, in order to capture the opposing player’s king. The goal of this project was to create a chess playing AI program using an adversarial search technique that implemented alpha-beta pruning within the search tree, as well as the creation of a user interface that would allow a player to play against the computer.

**2 User Interface**

The user interface for the Fluffy Bunny Chess System was created through the implementation of a Java Swing graphical user interface (GUI). The main GUI has a graphical representation of the board as well the current game status and move history. During the players turn the game will alert the player if they are in check with a larger banner in the top corner, as well as displaying a move history for the entire game and highlighting for the move the AI took. When the player selects a piece, that piece is highlighted in red and all moves available to that piece are highlighted in blue, to aid the player. During the computers turn a message appears warning the player that the AI is thinking and the player is prevented from selection and moving any pieces at this time. The user interface also allows the user to create a custom board at any time. If this is selected a new window will pop up with options to place pieces this allows for simpler testing. The user also has options to save and load previous games, which have been stored in their FEN format in text files in the saved games folder. On any event such as piece promotion or game ending conditions the program will display a pop up message for the player. Below is an image of the board and customization window, as well as some sample dialogs to the user.

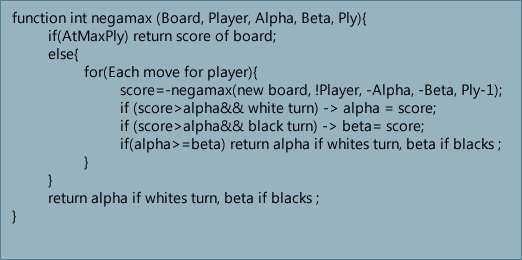


**3 AI System**

**3.1 Adversarial Search**

The search technique used by the AI to select which move to make was a negamax algorithm which implemented alpha-beta pruning. The negamax algorithm is a variation on the Minimax algorithm but it relies on the zero sum property of two player games, that is that the max score for one player is the negative of the minimum score of the negatives of the values for the other player. This allows the algorithm to be simpler then a minimax algorithm since there is no need to differentiate between minimizing and maximizing depending on which players turn it is. When the algorithm is called it begins searching the game tree with alpha and beta values set to unreachably high and low values. It makes each possible move for each player, recursively, until it reaches the maximum specified depth, at which time the board is evaluated and the results returned. If at any point a branch has a result which cannot improve upon the best result already found, alpha or beta depending on which colour is found, it returns out of that branch and does not evaluate the branch further. Below is the pseudo code for negamax with alpha-beta pruning. The search will immediately return a move if that move achieves checkmate. Due to the nature of the heuristic function, as outlined below, better moves have a tendency to be ones that occur in the middle of the board. This was used in the move set part of the alpha- beta pruning. By starting the search with pieces in the center of the board the pruning was able to remove more branches from the tree without sacrificing the results. This created a considerable decrease in the time taken to make each move.

**Negamax Pseudo-Code**



**3.2 Move Set Generation**

To generate the moves available for each player the program is given the colour of the players whose moves it is calculating and scans through the board. When a piece of that colour is found it then creates a list for that piece of all possible moves available to that piece regardless of the moves outcome on the game. The move set creation algorithm various from direct checking of known places for the pawn knight and king, to a more complex check for the pieces that have a much larger move set. Once this list of possible moves has been created each move is made and unmade on a temporary local board. If the move results in that colour’s king being put into check the move is discarded from the list. Special case moves are also checked; en passant, piece promotion, and castling.

**3.2.1 En passant**

The ability for a pawn to capture another pawn through en passant is stored within the board itself. The location of the en passant square is stored within the board as an x,y coordinate of where the pawn would move to if an en passant attack were to occur. The pawn move generator checks for this location as a possible attack and adds it to the move set if the move can be taken.

**3.2.2 Pawn Double Advancement**

If the pawn is located on its starting square the move set algorithm checks an additional square, two squares in front of the pawn. If that square and the square directly in front of the pawn are free then the move is added. This also flips the Boolean which signals that en passant is possible and stores the place skipped over as the en passant square, if the move is taken.

**3.2.3 Piece Promotion**

If a pawn reaches the end of the board, the computer will first check if a promotion to knight will result in an immediate checkmate. If this is the case then the pawn is promoted to a knight, otherwise it is promoted to a queen. If it is the player who promotes a pawn, a pop up will open and allow the player to pick any piece, with the exception of king and pawn, to promote to.

**3.2.4 Castling**

In this program castling is considered a move in the king’s move set. The program checks if the king and rook of whatever side castling is being called on are in their original position, It then checks if all squares between them are empty. If this is the case the castling move is added to the move set.

**3.3 Evaluation**

The evaluation of the board is an eight part function which is comprised of; material score, positional score, mobility score, outposts, rook scoring, pawn scoring, and king scoring. In order to improve scoring time due to the large nature of the evaluation function, a hash table was implemented.

**3.3.1 Material Score**

Each of the individual pieces is assigned the following values; Pawn: 100, Knight: 300, Bishop: 325, Rook: 500, Queen: 900, and King: 32767. White is assigned positive values, and black is assigned negative values, with empty squares having a zero score. The materiel score is calculated by scanning the board and taking the sum of all pieces on the board.

**3.3.2 Positional Score**

With the exception of rooks, pieces tend to perform better when placed closer to the center of the board. That being said they do not perform as well when placed too close to opposing pieces. The positional score therefore awards points to pieces placed near the center of the board, these points however decrease the when the piece is too far into opposing territory.

**3.3.3 Mobility Score**

The more moves which are available to a piece the more likely it is to become useful in the game, each piece is therefore given points for each move that is available to it, regardless of the safety of that move.

**3.3.4 Outposts**

Central pieces that are being protected by pawns are always a threat. They are very strong pieces and inhibit the opponent’s ability to move safely or capture. Bishops and knights that are on outposts are therefore given a small bonus towards the heuristic score, while knights are given a higher bonus to their increased versatility in outpost positions.

**3.3.5 Rook Scoring**

Rooks are given a slightly different evaluation then other pieces to their increased performance when in an offensive role, and at the same time their ability to be strong defensively and to protect pawns. Due to this, unlike most pieces they do not perform as well when placed near the center of the board. Their performance is more heavily based on having open columns that allow them to easily move to attack and defend, allowing them to become one of the larger threats. Rooks are therefore given an additional bonus if they are in an open column. They are given a smaller bonus if the column they are in only has enemy pawns (half open). Additionally a rook placed on the 7th rank has a much greater ability to attack pawns, so a bonus is added if the rook is on this rank.

**3.3.6 Pawn Scoring**

Due to the nature of the AI the value of pawns will generally not be noticed until later in the game, at which the pawn is most likely nearing promotion. To push pawns towards the center the pawns are given an additional score if they advance. Central pawns being given a higher bonus then the pawns on the edges to promote taking the center of the board, and since the pawns near the edges are the ones that tend to contribute most to stalemates in the endgame. Pawns have other scores associated with them. If a player has two pawns in one file, they are called doubled pawns. Doubled pawns tend to be weaker because the pawns no longer support one another. To try to prevent the AI from doubling pawns up when it has other options, a penalty is attached to doubled pawns. Similar to doubled pawns, there is what's known as an isolated pawn. An isolated pawn has no friendly pawns in either of the columns neighboring it. So, the isolated pawn can never be defended by a friendly pawn. Isolated pawns tend to become key defensive points, and also a large weakness that the opponent can hone in on. In order to help the AI prevent isolating it's pawns, a penalty is attached to any isolated pawn.

**3.3.7 King Scoring**

For most of the game, up until the endgame, the king has very little use. The best move for a king during the early game is to castle and remain protected. In the early game the king is given a bonus if it has castled in order to get the AI to castle. In the endgame, where king position is critical the king is given very large positional scores, depending on where the pawns are located.

**3.3.8 Hashing**

Due to the very high branching factor of chess there is an extremely large amount of boards to be for all possible moves, many of which would have already been evaluated. In order to speed up the process, a hash table was introduced. The hash table uses Zobrist keys. If a board is hashed and the board is already been stored, then the score for that board is taken from the hash table and not from the evaluation function in order to save time. A hash table size of 5,000,000 was used, as any larger did not improve the time in any considerable way.

In Zobrist hashing, a set of different board elements are initialized to random 64bit numbers. The algorithm then goes through each of the elements in the actual board. If the board is in some particular state, the key is XOR'd with the random number. For example, if (x,y) contains a white rook, the algorithm XOR’s the current key with the random value given to piece[white][rook][x][y]. Doing this for all game aspects gives a unique enough key for any position. 64bits isn't quite enough to create a unique key for every position, but collisions are so rare that the effect is negligible to the results. It is very unlikely that two keys will collide when going only 6 ply.

**4 Stalemate**

There are multiple ways which the game can end in a stalemate. The first being no available moves for a player, given the player is not in checkmate. The AI recognizes this and attempts to avoid it if possible, but still values stalemate much greater than losing it will stalemate if it is necessary. The second way stalemate can be achieved is if there has been a total of 50 moves without a piece taken or a pawn moved. The AI returns a 0 in this case and it is deemed a draw. A third way that stalemate can be achieved is by a lack of material to checkmate. For example, a bishop and king cannot checkmate a lone king. If there is a pawn on the board, then a checkmate is still possible. The final way for stalemate to occur is if the same board is repeated three times. In order to check for this a text file is kept containing all the FEN's of the current game. When a move is made it checks if this FEN is in the file twice already and if it is it ends the game in a stalemate.

**5 Results**

**5.1 Algorithm Speed**

The algorithm was tested in various plys to determine its effectiveness. At four ply the algorithm performs at a rate consistent to a game of speed chess, with moves taking less than 10 seconds to be performed. At six ply the program performs closer to tournament chess. Moves can take considerably longer but still result in game lengths less than that of a full tournament game. The algorithm was slowest when the queens were in open positions, as the amount of possible moves for the queens resulted in a much higher branching factor for the search. Due to the nature of the heuristic function the speed of the algorithm suffers in speed when searching for deeper moves. However the strength of the heuristic function compensates for the ability to search less moves. There was generally a negligible difference between moves made between the different plys. This enables a lower ply to be used without sacrificing the strength of the moves made. Another aspect of the program that resulted in higher speeds was the hash table**.** For plys greater than 5 the number of boards evaluated through a hash table lookup surpassed the number of boards evaluated through the evaluation function, as demonstrated in the graph below. The hash table look ups accounted for between the hash table was able to significantly reduce the number of board evaluations necessary. Since the board evaluation was by far the most time consuming aspect of the algorithm, this had a considerable impact on the over time taken by the algorithm.

**5.1.1 Boards Evaluated**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 Ply | 2 Ply | 3 Ply | 4 Ply | 5 Ply | 6 Ply | 7 Ply | 8 ply |
|  |  |  |  |  |  |  |  |  |
| Eval Score | 20 | 380 | 5381 | 49980 | 447298 | 2469438 | 28034164 | 1.49E+08 |
|  |  |  |  |  |  |  |  |  |
| Hashed | 0 | 0 | 715 | 25993 | 509335 | 3417124 | 45753792 | 2.23E+08 |
|  |  |  |  |  |  |  |  |  |
| Total Boards | 20 | 380 | 6096 | 75973 | 956633 | 5886562 | 73787956 | 3.72E+08 |
| Percent Hashed | 0 | 0 | 11.729 | 34.21347 | 53.24247 | 58.04957 | 62.00713 | 59.99296 |

**5.2 Competitive performance**

The system was tested against various preexisting chess programs with the following results.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Wins | Losses | Draw |
| Chess Titans Level 5 | 6 | 3 | 1 |
| Chessmaster ELO 1000-1300 | 3 | 3 | 4 |
| Chessmaster ELO 1300+ | 0 | 3 | 0 |

In the games against the Chessmaster program, where the game resulted in a stalemate, the stalemate was caused by threefold repetition while the fluffy bunny chess system was wining. Overall the system performed quite well against the Microsoft Chess Titans Program on its medium difficulty and was able to beat it twice as often as it lost. The performance of the system would put it at a Class D by the United States Chess Federation (Classes Range from A-J As well as Senior Master, National Master and Expert).

**6 Conclusions**

The use of alpha beta pruning on the negamax algorithm greatly improved the speed of the program. However the algorithm is not suited for a grandmaster level chess program such as Rybka. The branching factor of chess moves requires a technique beyond that of a simple adversarial search in order to produce results adequate for truly high level play. The adversarial search technique is however a much simpler algorithm to implement and is therefore suited well to a mid level chess program such as Fluffy Bunny Chess.