26-05-2014

Board game - CHESS **Department of Electrical Engineering and Information Technology**

Danmarks tekniske universitet Ballerup campus

Lautrupvang 15, 2750 Ballerup

Board Game CHESS

62526 Artificial Intelligence in Computer Games F14

26-05-2014

Board game – CHESS

Student name	Student number	Signature
Vilius Maskovas	s107154	
Justas Mikalajunas	s107083	
Jacob Lambert	s136215	

Contents

1.Introduction

Although the concept of chess programming has existed in theory since the early 20th century, computer chess first became popular in mainstream media in 1989, with the famous match between the reigning world champion Garry Kasparov and IBM's Deep Thought. Deep Thought was a chess computer with hardware specifically designed for playing chess, and was able to reach a move search depth of up to 11 moves ahead. However, even with the computational power of Deep Thought, Gary Kasparov was able to defeat it, and chess programmers discovered that creating a more powerful chess computer relied not only on computational power, but also powerful applications of artificial intelligence and heuristics.

In 1996, IBM held a rematch with Gary Kasparov, this time using the chess computer Deep Blue, and for the first time a chess program won a game against a chess world champion. Finally in 1997, an upgraded version named Deeper Blue won an entire match against Kasparov, although the games were very close. This event was monumental to the development of chess programming.

Throughout the late 1990s and early 2000s, multiple matches were held between high-rated chess players and chess computers. As time passed, chess engines and computers with less computational power and smarter artificial intelligence were able to defeat high-rated human players. Today chess engines running even on low hardware are able to defeat grandmasters, and the most powerful chess engines play in entirely separate tournaments, unbeatable by humans.

Our chess applications builds upon the decades of research and testing done by these famous chess programmers. The application relies on the popular alpha-beta algorithm, along with array-based board representation and dynamic board evaluation, and is able to play legal chess using intelligent movements.

1.1. Problem formulation

We chose to program chess opposed to other multiplayer games due its complex nature and large game tree of chess. Chess is easily represented in a computer because it is simply a two player strategy board game played on a board with 64 squares arranged in an eight by eight grid. The rules of chess focus on movement of different pieces, pawns, rooks, knights, bishops, queens, and kings, and the goal is to put the opponent's king in a position where he can make no moves to escape capture, forcing a surrender.

Basic rules were set to make engine work reasonably:

- Standard movements
- End of game – check mate
- Queen side castling and king side castling excluded
- En passant excluded
- Promotion excluded

1.2. Problem analysis

In problem analysis we review design issues and decisions.

1.2.1.Algorithm behind the scenes

The general algorithm used is based on the min-max principle, where possible future moves are analysed. In the algorithm, the friendly team attempts to maximize the score by choosing the most favourable moves, while the enemy team attempts to minimize the score. During further design discussions, we decided to add alpha-beta pruning which improves upon the min-max algorithm. The alpha-beta pruning doesn't search paths that are guaranteed to not produce a minimum or maximum score, thus cutting off branches of the game trees and reducing computational complexity.

1.2.2.Board

We chose the board representation after researching and discussing the 3 most common ways of representing a chess board on a computer:

- 2D array
- 0x88 – hex board
- Bit boards

For simplicity and to allow more time to focus on other elements of our chess application, we chose the least abstract method, array representation. However, this representation is the most memory taxing, so for future improvements we will consider other options.

1.2.3.Evaluation

Many books offer different methods of evaluating a chess board. However, they all rely on the same core principles, especially evaluating the king's safety and the following aspects of each piece:

- Value
- Mobility
- Position on the board.

We have chosen specific values to assign to the board in each possible situation. We have used the books and repeated testing to assign these values. However, as with all chess programs, our evaluation is subject to errors and needs continual improvement.

1.2.4.Move generation

Move generation for our chess application is inspired and governed by the relatively few and simple rules in chess. For each chess piece on the board, we generate all legal positions reachable for that piece. However, there are many underlying considerations that make this process lengthy and computationally expensive.

1.3. Requirements

Functional requirements

FR1	Chess engine shall make/generate only legal moves
FR2	Chess engine shall evaluate board according to evaluation function
FR3	Chess engine shall make decisions offered by alpha beta algorithm
FR4	Chess engine shall calculate best move

Non-functional requirements

NFR1	Chess engine shall be implemented in C# programming language
NFR2	Chess engine shall use modified rules of chess

2.Chess engine

In the following part we will overview all ideas related to our chess engine, including data structures and design ideas.

2.1. Alpha-beta algorithm

The game of chess can be thought of as a tree of possible future game states, where a state is a snapshot of the positions of the pieces on the board. Our evaluation function can assign a value to each of these states, but this procedure is computationally expensive. With traditional min-max algorithms, we would be forced to search an entire level of the tree in order to determine the best move. For example, the number of moves from a middle-game position could be estimated to 40. With a search depth of 5 ply, this would result in 102 million leaf nodes. With alpha-beta pruning, we are able to reduce the number of branches explored, thus greatly reducing the number of relevant leaf nodes.

2.1.1.Pseudo-code of alpha beta algorithm

The following pseudo code demonstrates the underlying logic of the alpha-beta algorithm.

```
alpha-beta(alpha, beta, depth)
   if(checkmate)
      return score according to which player set piece last
          if(draw)
                    return zero sum
          if(depth == 0)
                    evaluate board
   if(players' turn)
                    children = all legal moves for player from this board
     for each child in children where alpha < beta else cut-off
         append move to board
         call alpha-beta recursively with depth - 1
                                  if (value of child > alpha)
                                            found players best move
   else (opponents' turn)
                     children = all legal moves for player from this board
     for each child in children where alpha < beta else cut-off
         append move to board
         call alpha-beta recursively with depth - 1
```

```
if (value of child < beta)
        found opponents best move
```

2.1.2.Efficiency of algorithm

While analyzing the efficiency of the alpha beta algorithm, we look into growth rates and best case versus worst case cutoffs.

In picture 1:

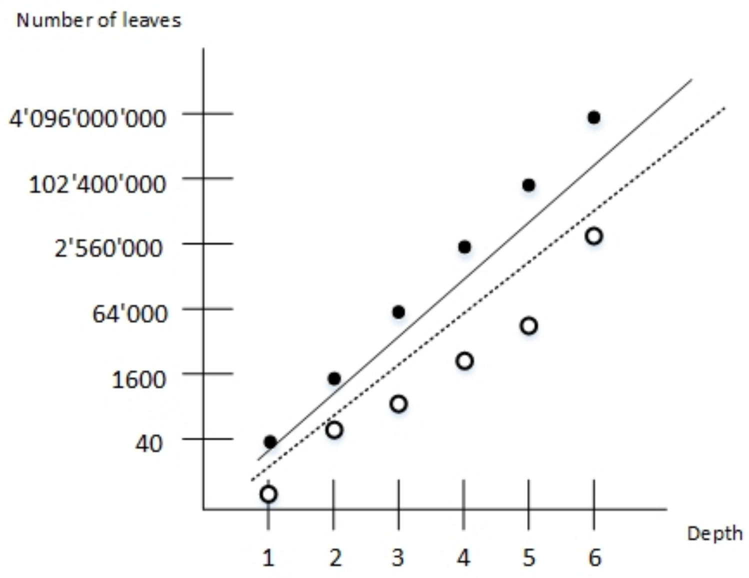
Filled circles – min max algorithm without alpha beta pruning or worst case where no pruning is done. Growth rate is exponential. As we mentioned before branching factor in mid game of chess is 40, so growth rate is $Leaves = 40^d$ where d is depth of game tree.

Non-filled circles - best case with alpha beta pruning - $Leaves = 2b^{d/2} = (2*40)^{d/2}$ (formula taken from Artificial Intelligence by Patrick Henry Winston, 3rd edition).

Full line – shows approximately our implementations growth rate as there is no specific ordering of moves.

Dashed line – shows wanted and achievable growth rate using alpha beta algorithm.

Growth rate of alpha beta		
Depth	Worst case ●	Best case ○
1	40	9
2	1600	80
3	64000	716
4	2560000	6400
5	102400000	57243
6	4096000000	512000



Picture 1

Because our move generation has no associated heuristics for choosing moves, the cutoff can be considered random, sometimes experiencing near best-case performance, and sometimes near worst-cast. The unpredictable nature of the cutoff with our current implementation makes it difficult to measure the exact growth rate. However, implementing heuristics to predictably order the moves will decrease the growth rate and improve the computational efficiency of the application.

2.2. Board

To represent board we took 2D array of 8x8. The indexing starts at top left corner with coordinates (0,0). Pieces are represented as follows:
Capital chars for black pieces
Lower case chars for white pieces

The downside of array representation stems from the additional calculations needed to guarantee a piece moves to a position on the board. Also, we need to store and transfer two values, both a row and a column. However, as mentioned previously, this implementation comes very naturally from traditional chess.

```
'R',  'N',  'B', 'Q', 'K', 'B',  'N',  'R'
'P',  'P',  'P', 'P', 'P', 'P',  'P',  'P'
'-',  '-',  '-', '-', '-', '-',  '-',  '-'
'-',  '-',  '-', '-', '-', '-',  '-',  '-'
'-',  '-',  '-', '-', '-', '-',  '-',  '-'
'-',  '-',  '-', '-', '-', '-',  '-',  '-'
'p',  'p',  'p', 'p', 'p', 'p',  'p',  'p'
'r',  'n',  'b', 'q', 'k', 'b',  'n',  'r'
```

In improvements we discuss which board presentation we would choose for improving the performance and its advantages and disadvantages compared to the 2D array.

2.3. Evaluation

Our chess board evaluation function consist of 2 parts:

- Evaluation tables
- Threatening status

2.3.1.Evaluation tables

This part evaluates the board according to pieces' places on the board. To do that each piece has an associated value:

Pawn = 100

Knight = 320

Bishop = 330

Rook = 500

Queen = 900

King = 20000

These values were found in chess book and may differ from book the book. Because the king value is normally represented by infinity, we simply chose a sufficiently large value for the discrete representation.

In addition to specific piece values, we are using evaluation tables. There is one table for each piece, made according to piece's movements and optimal placements on the board.

Pawn

```
 0,  0,  0,  0,  0,  0,  0,  0,
50, 50, 50, 50, 50, 50, 50, 50,
10, 10, 20, 30, 30, 20, 10, 10,
 5,  5, 10, 25, 25, 10,  5,  5,
 0,  0,  0, 20, 20,  0,  0,  0,
 5, -5,-10,  0,  0,-10, -5,  5,
 5, 10, 10,-20,-20, 10, 10,  5,
 0,  0,  0,  0,  0,  0,  0,  0
```

Knight

```
-50,-40,-30,-30,-30,-30,-40,-50,
-40,-20,  0,  0,  0,  0,-20,-40,
-30,  0, 10, 15, 15, 10,  0,-30,
-30,  5, 15, 20, 20, 15,  5,-30,
-30,  0, 15, 20, 20, 15,  0,-30,
-30,  5, 10, 15, 15, 10,  5,-30,
-40,-20,  0,  5,  5,  0,-20,-40,
-50,-40,-30,-30,-30,-30,-40,-50,
```

Bishop
```
-20,-10,-10,-10,-10,-10,-10,-20,
-10,  0,  0,  0,  0,  0,  0,-10,
-10,  0,  5, 10, 10,  5,  0,-10,
-10,  5,  5, 10, 10,  5,  5,-10,
-10,  0, 10, 10, 10, 10,  0,-10,
-10, 10, 10, 10, 10, 10, 10,-10,
-10,  5,  0,  0,  0,  0,  5,-10,
-20,-10,-10,-10,-10,-10,-10,-20,
```

Rook
```
  0,  0,  0,  0,  0,  0,  0,  0,
  5, 10, 10, 10, 10, 10, 10,  5,
 -5,  0,  0,  0,  0,  0,  0, -5,
 -5,  0,  0,  0,  0,  0,  0, -5,
 -5,  0,  0,  0,  0,  0,  0, -5,
 -5,  0,  0,  0,  0,  0,  0, -5,
 -5,  0,  0,  0,  0,  0,  0, -5,
  0,  0,  0,  5,  5,  0,  0,  0
```

Queen
```
-20,-10,-10, -5, -5,-10,-10,-20,
-10,  0,  0,  0,  0,  0,  0,-10,
-10,  0,  5,  5,  5,  5,  0,-10,
 -5,  0,  5,  5,  5,  5,  0, -5,
  0,  0,  5,  5,  5,  5,  0, -5,
-10,  5,  5,  5,  5,  5,  0,-10,
-10,  0,  5,  0,  0,  0,  0,-10,
-20,-10,-10, -5, -5,-10,-10,-20
```

King middle game
```
-30,-40,-40,-50,-50,-40,-40,-30,
-30,-40,-40,-50,-50,-40,-40,-30,
-30,-40,-40,-50,-50,-40,-40,-30,
-30,-40,-40,-50,-50,-40,-40,-30,
-20,-30,-30,-40,-40,-30,-30,-20,
-10,-20,-20,-20,-20,-20,-20,-10,
 20, 20,  0,  0,  0,  0, 20, 20,
 20, 30, 10,  0,  0, 10, 30, 20
```

The algorithm checks square by square and if it finds the piece, it adds its value with value the from table according to piece's position, either adding or subtracting from the total based on the color of the piece.

The formula is as follows:

board score = board score +- (Piece score + Piece board(coordinates))

2.3.2. Threatening

When the algorithm evaluates a piece, it checks all directions it can move. If the piece can take over an opponent's piece, additional points are added or subtracted of the board according to the color of the piece. Additionally, we tried assigning different values for threatening, and we saw that AI plays best with these threatening values:

Pawn = 70;
Knight = 150;
Bishop = 200;
Rook = 300;
Queen = 500;
King = 350;

The king has 350 value so that AI would go straight to threaten the king and sacrifice its pieces only for check, but still would check the opponent's king if there is a safe way.

2.4. Move generation

To avoid excessive conditional clauses within our move generation function, we divided our most outer function level into two separate functions, one for black and one for white. Therefore, for every function relating to the black team, we also have written a parallel function for the white team, taking into account the differences in legal moves. Our alpha-beta algorithm calls the appropriate team's function depending on player turn and minimizer and maximizer roles.

The move generation functions iterate through every position on the board, calling the appropriate piece-specific move generation function. These functions return a list of legal moves to the alpha-beta algorithm, forming the next level of the search tree.

2.4.1.Piece-specific Moves

Generation of moves for each piece consists primarily of two steps, generating plausible moves of the piece based on the governing rules in chess and validation that these moves are legal. For a move to be considered legal, it must not cause the piece to leave the boundaries of the board, and it must move only to an unoccupied or enemy-enemy occupied square. To guarantee legal moves, we must also guarantee that the king piece is not in check after the move, which is discussed in detail below.

Generation of moves for the pawn pieces proved to be the most straightforward due to their limited range of motion. We need only to generate moves forward one square if the next position is unoccupied, forward two squares if the next two positions are unoccupied and the pawn is in the starting position, and moves for capturing. For capturing, we allow the pawn to move forward one and left or right one if these positions is occupied by an enemy piece.

Similarly for the knights, we generate the 8 possible movement positions, verifying only that the knight remains on the board and moves to an enemy or unoccupied square. Like the functions for the rook, bishop, king, and queen, the functions for the movement of the white and black rooks are very similar. The main differences between the two sets of functions are the classification of enemy and friendly pieces.

For the rooks, we generate moves corresponding to four directions on the board, north, east, south, and west. In each direction, we generate a move for every unoccupied position in that direction until we reached an occupied square or the edge of the board. If the next position is occupied by an enemy piece, we generate one additional move, capturing that piece.

Like the rooks, we also generate moves in four directions for the bishops, north-west, north-east, south-west, and south-east. The other details of generation for bishops are similar to rooks. For queens, we simply call the move generation functions for both rooks and bishops. Finally, for the kings, we simply check the 8 surrounding squares and generate one move for each position that lies within the boundaries of the board and is unoccupied or enemy-occupied.

Verifying that the king is not threatened for each move uses logic that mirrors the piece-specific move generation. First, we execute the move to yield the state of the board after the move has been made. Then we conceptually replace the friendly king with each type of piece, and then test to see if that piece can capture an enemy piece of the same type. For example, to see if the white king is threatened by a black knight, we replace the white king with a white knight and test to see if this white knight is able to capture any black knights. See figure (a).

(a)

```
'-', '-', '-', '-', '-', '-', '-', '-'
'-', '-', '-', '-', '-', '-', '-', '-'
'-', '-', '-', '-', '-', '-', '-', '-'
'-', '-', '-', '-', '-', '-', '-', '-'
'-', '-', '-', '-', 'N', '-', '-', '-'
'-', '-', '-', '-', '-', '-', '-', '-'
'-', '-', '-', 'k', '-', '-', '-', '-'
'-', '-', '-', '-', '-', '-', '-', '-'
```

After replacing the king with a knight, we can see that he is threatened because he is able to capture an enemy piece of the same type.

```
'-', '-', '-', '-', '-', '-', '-', '-'
'-', '-', '-', '-', '-', '-', '-', '-'
'-', '-', '-', '-', '-', '-', '-', '-'
'-', '-', '-', '-', '-', '-', '-', '-'
'-', '-', 'x', '-', 'N', '-', '-', '-'
'-', 'x', '-', '-', '-', 'x', '-', '-'
'-', '-', '-', 'n', '-', '-', '-', '-'
'-', 'x', '-', '-', '-', 'x', '-', '-'
```

(b)

```
'-', '-', `-', '-', '-', '-', '-', '-'
 '-', '-', '-', '-', '-', '-', '-', '-'
 '-', '-', '-', 'R', '-', '-', '-', '-'
 '-', '-', '-', '-', '-', '-', '-', '-'
 '-', '-', '-', '-', '-', '-', '-', '-'
 '-', '-', '-', '-', '-', '-', '-', '-'
 '-', '-', '-', 'k', '-', '-', '-', '-'
 '-', '-', '-', '-', '-', '-', '-', '-'
```

By the same process we can see the king is threatened by a rook.

```
 '-', '-', `-', '-', '-', '-', '-', '-'
 '-', '-', '-', '-', '-', '-', '-', '-'
 '-', '-', '-', 'R', '-', '-', '-', '-'
 '-', '-', '-', 'x', '-', '-', '-', '-'
 '-', '-', '-', 'x', '-', '-', '-', '-'
 '-', '-', '-', 'x', '-', '-', '-', '-'
 'x', 'x', 'x', 'r', 'x', 'x', 'x', 'x'
 '-', '-', '-', 'x', '-', '-', '-', '-'
```

Finally, we restore the board to its previous state. Using this structure, we were able to reuse the logic written for the move generation functions. This function is required to ensure only legal moves are generated and contributes significantly to the computational expense of move generation.

3.Conclusion

3.1. Improvements

3.1.1.Board

To improve the board representation, we would like to implement bit boards instead of arrays. Bit boards are able to take advantage of essential logical bitwise operations, available on nearly all CPUs, that complete in one cycle and are fully pipelined and cached. Another improvement comes with 64 bit machines as 64 bit operation can occur in one operation. This would greatly speed up operations. However it is not that easy to debug as it is hard to read and most of operations are atomic.

3.1.2.Alpha beta pruning

For improving the alpha beta algorithm we need to use some kind of move ordering heuristics. Our idea to speed up the search was to use an expected move method. We would use a table throughout the entire game where we would save branches according to which we explored in the game tree. For example, suppose we search up to a depth of d, and make some move (move 1). If our opponent responds with a move we expected, then we will have already examined our response position when calculating our previous move, at a depth of d - 2. Since we stored the results in the table, we can skip the searches and start examining the resultant position at depth d - 1.

3.1.3.Move generation

The verification that the king is not threatened is not currently implemented in our chess application. The implementation would require slight reorganization of our main functions.
The reason our program was able to function without application of the check is due to the evaluation strategy. Any move that would result in leaving the king exposed would receive a very negative score and would not normally be chosen.

Another possible improvement would involve the use of static move tables instead of dynamically generating the moves. Each move table would be a 3D array. The rows and columns would represent the board, and each entry would be a list of moves possible from that position. For example, for knights the entry at B1 would contain the list C3 and A3.

c)

```
    a    b    c    d    e    f    g    h
8  '-', '-', '-', '-', '-', '-', '-', '-'
7  '-', '-', '-', '-', '-', '-', '-', '-'
6  '-', '-', '-', '-', '-', '-', '-', '-'
5  '-', '-', '-', '-', '-', '-', '-', '-'
4  '-', '-', '-', '-', '-', '-', '-', '-'
3  'x', '-', 'x', '-', '-', '-', '-', '-'
2  '-', '-', '-', 'x', '-', '-', '-', '-'
1  '-', 'n', '-', '-', '-', '-', '-', '-'
```

The only verification needed for move tables is that the target destination is unoccupied or enemy-occupied. For pieces that can move a variable distance in a direction, implementation of move tables would be more difficult. However, these tables simplify the code from dynamic calculations to static lookups. Implementing move tables would decrease the computational complexity and further compartmentalize the code.

3.1.4.Evaluation function

To improve this algorithm one more table for the king might be used. It would be used in at the end of the game, because the optimal placement for the king in the beginning of the game, near the corners, switch to an optimal placement at the end of the game, which is closer to the middle of the board.

End game determination:
Both sides have no queens or every side which has a queen has additionally no other pieces or one minor piece maximum.

King end game

```
-50,-40,-30,-20,-20,-30,-40,-50,
-30,-20,-10,  0,  0,-10,-20,-30,
-30,-10, 20, 30, 30, 20,-10,-30,
-30,-10, 30, 40, 40, 30,-10,-30,
-30,-10, 30, 40, 40, 30,-10,-30,
-30,-10, 20, 30, 30, 20,-10,-30,
-30,-30,  0,  0,  0,  0,-30,-30,
-50,-30,-30,-30,-30,-30,-30,-50
```

To make the AI better additional rules can be used:

1. Avoid exchanging one minor piece for three pawns.

Bishop > 3P

Knight > 3P

2. Encourage the engine to have the bishop pair.

Bishop>Knight

B > K > 3P

3. Avoid exchanging of two minor pieces for a rook and a pawn.

B + K > R + P

3.2. Overview

In the table below we overview our main parts of chess engine and its status.

Part	Status
Move legitimacy check	Done
Move generation	Done
Decisions upon alpha-beta algorithm	Done
Evaluation function	Done

3.3. Process assessment

The project work was divided between group members in order to work most efficiently. Initially, we could easily spot errors in our algorithms by playing against the chess engine. However, as the engine improved, and because we are not experienced chess players, it became difficult to determine when the engine executed unreasonable moves. After the tournament and commentary from the instructor, who is an experienced chess player, we were able to see additional errors in our AI implementation, which we mention in the report.

4.Bibliography

Chess programming assistance:
http://chessprogramming.wikispaces.com/

History of chess:
http://en.wikipedia.org/wiki/Computer_chess