Spell Chess using the Minimax Algorithm

Braxton Brown
Computer Science Department
Utah Valley University
Orem, USA
10884927@uvu.edu

Nathan Kelley
Computer Science Department
Utah Valley University
Orem, USA
10856968@uvu.edu

Koliniasi Vanisi
Computer Science Department
Utah Valley University
Orem, USA
10723789@uvu.edu

***Abstract*— In this paper we describe our process in creating a program that plays the chess variant spell chess. Humans have been playing chess for thousands of years, computer chess programs have been playing for less than 100 years and can defeat the best human players alive. Using the minimax algorithm we have written an engine to evaluate board states and chess moves using the ruleset of spell chess, as there is no widely available algorithm to play this variant.**

I. Introduction

A. Idea

The game of chess has existed for thousands of years across numerous cultures. Chess is played worldwide even today, boasting massive tournaments a global ranking system and several grandmasters. It is a complex game with billions of possible board states. As complicated as it is, there have also been many variants to the game to add more options for the players, or change the board state in other ways.

Chess.com is a website that hosts online chess games for thousands of players, it also provides lessons and game analysis using its chess engine. After a game is played, a user can go back through the moves played and see the rating of each move. Chess.com also hosts a number of chess variants including spell chess. However, Chess.com currently has no engine for this variant so we decided to write our own.

B. Spell Chess

The Chess variant spell chess deviates from the standard form of chess in the way of "spells" that the players can cast before they move. The jump spell allows the caster to select one of their opponent's pieces that can be jumped over by one of the caster's pieces for a turn. The freeze spell allows the caster to choose a 3x3 area on the board in which no pieces can move or check for a turn. An important rule difference between spell chess and regular chess is that in spell chess, the game ends on either a checkmate, or on a king capture. This means it is possible to win by using the jump spell on a piece in order to capture the opponent's king.

C. Background Research

Computer chess programs have had a significant impact on artificial intelligence development throughout its history. In 1950, Shannon wrote a paper applying the minimax algorithm to Chess, this paper has been hugely influential. This paper defined two main forms of Chess algorithms, type A and type B.

"Type A are those that search by 'brute force' alone, while type B programs try and use some considerable selectivity in deciding which branches of the game tree require searching." (Heath and Allum)

Fig. 1 Our spell chess GUI including the chess board and also the spells available to each player

The majority of popular computer chess programs, including our implementation are type A. Some type A algorithms include minimax, alpha beta pruning and the use of transition tables, this is because they perform a search of all the current possible moves out to a certain point where the possible board states branch out enough to be infeasible to perform an exhaustive search on. Type B algorithms would not necessarily search all of the current possible moves, instead it would prune multiple moves based on a specific criteria and then drill into the remaining moves to find the best one.

II. Writing the Program

A. *The minimax algorithm*

The minimax algorithm is a form of artificial intelligence algorithm that uses an adversarial search to determine the best possible move to make on any given turn.

The algorithm works in this way:

1. First, the algorithm finds every possible move and creates a node for each
2. For each node, the algorithm finds every possible move the opponent can play and creates a node for these moves
3. This repeats out to a certain point at which the game state is evaluated.
4. Assuming that the opponent play optimally choosing the minimum (min) value option for the algorithm each time, the algorithm can choose the maximum value board state for each of it moves.
5. This continues until the top node has a value which is the maximum value of each minimum values the opponent has chosen.

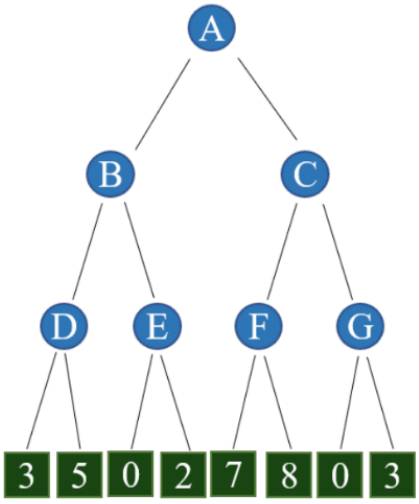
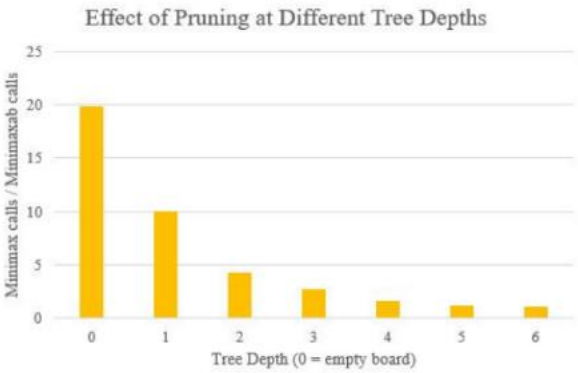


Fig. 2 An example tree in the minimax algorithm: the algorithm evaluates all possible moves from the node A then evaluates all possible moves from the new created nodes B and C



# Spaces Filled	No Pruning	Pruning	Multiple
0	549945	27662	19.9
1	61004	6112	10.0
2	7637	1779	4.29
3	997	364	2.74
4	180	110	1.63
5	35.2	29.5	1.21
6	16.0	14.5	1.10

Fig. 3 The effect of pruning on a game of tic-tac-toe: On an empty board, pruning can reduce the number of nodes created by 20x (Felstiner)

B. *Board Evaluation*

The most essential aspect of writing the spell chess engine is a way to effectively evaluate the board. Without a good board evaluation function, it is impossible to have the engine both evaluate, calculate, and output a move in a timely manner.

For our piece placement, we used a number of considerations, including number of open squares, pieces being attacked, and ensuring that the piece is safe from attacks. We also included a board scoring array, as a basis for good moves. When evaluating a board state, the best possible state is a checkmate, both because it is a terminal case for node evaluation and because it results in a victory. The board scoring arrays are provided below.

```
{ -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 }
```

Fig. 4 White King Score Array

```
{ -10,   0,   0,   0,   0,   0,   0, -10 },
{  10,  30,  50,  50,  50,  50,  30,  10 },
{   0,  40,  60,  70,  70,  60,  40,   0 },
{ -10,  30,  50,  65,  65,  50,  30, -10 },
{ -20,  20,  40,  45,  45,  40,  20, -20 },
{ -30,  10,  20,  30,  30,  20,  10, -30 },
{ -40, -20,   0,   0,   0,   0, -20, -40 },
{ -50, -40, -40, -40, -40, -40, -40, -50 }
```

Fig. 5 White Queen Score Array

```
{ 40 },
{ 50 },
{ 40 },
{ 30 },
{ 20 },
{ 10 },
{  0 },
{  0 }
```

Fig. 6 White Rook Score Array

```
{ -50, -40, -40, -40, -40, -40, -40, -50 },
{ -40, -20,   0,   0,   0,   0, -20, -40 },
{ -40,   0,  10,  20,  20,  10,   0, -40 },
{ -40,   0,  20,  25,  25,  20,   0, -40 },
{ -40,   0,  20,  25,  25,  20,   0, -40 },
{ -40,   0,  10,  20,  20,  10,   0, -40 },
{ -40, -20,   0,   0,   0,   0, -20, -40 },
{ -50, -40, -40, -40, -40, -40, -40, -50 }
```

Fig. 7 White Bishop Score Array

```
{ -50, -40, -40, -40, -40, -40, -40, -50 },
{ -40, -20,   0,   0,   0,   0, -20, -40 },
{ -40,   0,  10,  20,  20,  10,   0, -40 },
{ -40,   0,  20,  25,  25,  20,   0, -40 },
{ -40,   0,  20,  25,  25,  20,   0, -40 },
{ -40,   0,  10,  20,  20,  10,   0, -40 },
{ -40, -20,   0,   0,   0,   0, -20, -40 },
{ -50, -40, -40, -40, -40, -40, -40, -50 }
```

Fig. 8 White Knight Score Array

```
{ 0 },
{ 80, 90, 100, 100, 100, 100, 90, 80 },
{ 20, 22,  23,  25,  25,  23, 22, 20 },
{ 10, 11,  12,  13,  13,  12, 11, 10 },
{  8,  8,   9,  10,  10,   9,  8,  8 },
{  6, 6,   7,   8,   8,   7,  6,  6 },
{ 0 },
{ 0 }
```

Fig. 9 White Pawn Score Array

C. FEN Notation

Fen notation is an encapsulated way to read a board position in text. This notation is an easy way to parse a chess board within a program using minimal space. Our engine takes a line of FEN notation and evaluates board states from there. Our GUI returns a FEN notation to make it easy for our AI engine and GUI to interact.

FEN notation is made up of six strings described as the following:

```
rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1
```

Fig. 10 FEN notation for the start of a game of chess

1. A series of characters separated in columns by '/'. Lowercase letters represent black pieces, numbers represent open spaces and capital letters represent white pieces. See figure BLANK
2. The second string is a single character either 'w' or 'b' to show which player's turn it is.
3. String three represents the castling rule. Capital 'K' and 'Q' represent white's ability to castle on the king side and queen side, lowercase represents black's ability to do so.

.

```
[<frozen> <jumpable> <spells> <waiting_spells>]
```

Fig. 11 Our Spell Chess addition to FEN notation

4. String four represents the halfmove clock, which is the number of moves since a pawn was moved or a piece was captured. This rule exists to allow players to call for a draw if 50 moves have gone by without a pawn moving or a piece being captured
5. Finally, string five represents which turn the game is on

For our engine, we added a couple of strings to FEN notation to allow for variations of spell chess:

- <frozen>: The square with the current usage of freeze (there can only be one square at a time). If no square is frozen, the "-" char will be used. (ex. a2 means the square "a2" and the nearest squares to in all directions are frozen)
- <jumpable>: The square with the current usage of jump. As with the frozen square there can only be one at a time. If no square is jumpable, the "-" char is used. (ex. a2 means the square "a2" is jumpable)
- <spells>: Number of spells each player has: It has freeze followed by jump. White's spells are first. (ex. 5241 means white has 5 freezes and 2 jumps while black has 4 freezes and 1 jump)
- <waiting_spells>: How many half turns until each spell can be used. It goes freeze then jump, where

white is first. (ex. 0000 means all spells are ready to be used while 1300 means white has to wait 1 half turn to use freeze and 3 half turns to use jump while black can use both spells now)

D. Building the GUI

We wanted to develop a working GUI that we could use to provide us with important information that the AI could process in a real game. We decided to go with Unity to program an implementation of spell chess. The game itself is programmed in C#, and is created so that after each turn, it would call our AI engine and give it the current FEN so that it could calculate and output a move.

E. Difficulties and Obstacles

As we started work in having the AI engine find the best move, we first ran into issues with how many moves the AI would have to calculate, and quickly found that going more than 3 levels deep would cause it to take too long to make a move. In order to optimize Alpha-beta pruning, we decided to make a few decisions that would not even be considered, such as freezing your own pieces.

While running the Engine to create a move, we ran into huge memory and time issues where the engine took around 10 seconds and about 6 GB of RAM using alpha beta pruning just to investigate 3 half moves ahead. Found a solution to have it create the move on the stack instead of the heap and copying the whole object to the priority queue to be the solution to this reducing memory usage down to around 5 MB.

III. Conclusion

We were able to program a GUI that can make calls to an AI engine, and have that engine output a move, keeping track of the board using FEN Notation.

Using Minimax, and Alpha-Beta pruning, we were able to get the AI to explore 2 levels deep worth of possible moves in order to make a move it found to be best.

Having the AI explore moves 3 turns out is possible, but with our current implementation, takes around 3 mins to calculate the best move for it to make.

After optimizing our algorithms as best as we could without having our engine take too much time to find a move, we have found that our AI Engine is able to avoid making short term blunders, and could beat an estimated spell chess level 1200 player. Unfortunately, we found that if a player did plan more than about 2 moves ahead, they would be able to beat our AI, as it takes about 3 + mins for it to calculate moves that far ahead.

In conclusion, we found our AI capable of beating beginner level players, taking advantage of player blunders, and avoiding making short term, obvious blunders itself.

References

[1] Heath, David, and Allum, Derek. "The Historical Development of Computer Chess and its Impact on Artificial Intelligence." Deep Blue Versus Kasparov: The Significance for Artificial Intelligence 63 (1997).

[2] "Standard: Portable Game Notation Specification and Implementation Guide". Internet Archive. 12 March 1994.

[3] Wikipedia contributors. (2023, September 21). Forsyth–Edwards Notation. In Wikipedia, The Free Encyclopedia. Retrieved 02:09, December 8, 2023, from https://en.wikipedia.org/w/index.php?title=Forsyth%E2%80%93Edwards_Notation&oldid=1176345997

[4] Felstiner, Carl. "Alpha-Beta Pruning." (2019).