Othello and Parallel Game-tree Search

Dane Johnson and Aaron Handlman

12-21-2020

1 Introduction

The game of Chess has always been captivating to game enthusiasts, and doubly so to computer scientists. Aaron and I were the latest in a laundry list of computer scientists who became interested in the game. Given that our research is in parallel computing, and seeing as we needed a project for this course, it was only natural that we decided to investigate how parallelism could be applied to the great game.

Game-tree search soon became our primary area of concern. It is an inherently serial algorithm, challenging to extend to multiple workers. With the intent to focus on search algorithms, we gravitated away from Chess towards Othello, a simpler game with only one type of piece. With this new interest, we decided our project would be one of implementation and analysis, the prospect of creating an entire Othello engine from scratch was a much more tractable problem.

Thanks in no small part to the Chess Programming Wiki [1], we were able to find a wide variety of Game-tree search algorithms to implement, and benchmark against each other for runtime and scalability.

2 Othello

Othello, or Reversi, is a game played by two players on an 8x8 grid, the same dimensions of a Chess board (and I will therefore use a Chess board to demonstrate the game), with disks that are white on one side and black on the other. The game is played by placing a disk of the player's color on the board, then flipping any disks that are sandwiched between and existing player's piece and the new one. With only a single type of move, Othello is a much easier engine to implement than Chess, but Game-tree search algorithms should be just as, if not more, effective at playing the game.

Figure 1: The standard Othello start position

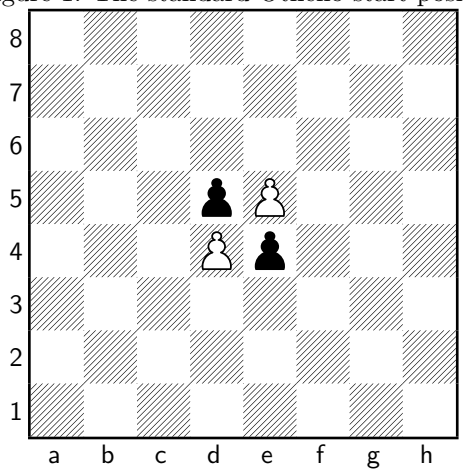
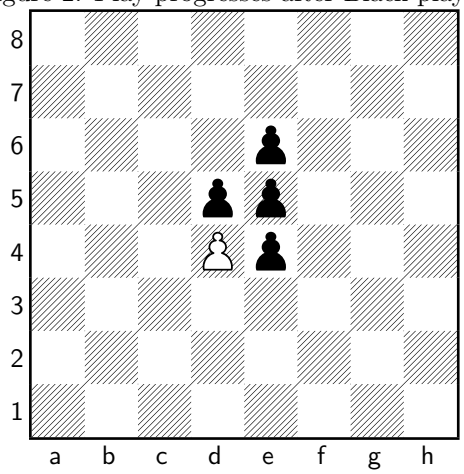


Figure 2: Play progresses after Black plays e6



3 Algorithms

3.1 Game-tree Search (Minimax)

The core of Game-tree search is looking through the search space trying to find the move that maximizes your chances of victory, under the assumption that your opponent will do the same in the opposite direction. Since theses spaces are intractably large, we choose some maximal depth, nodes searched, or time spent looking and then evaluate the board via some criteria (in Othello, how many more moves we have than our opponent).

```
1: procedure MINIMAX(node, depth)
2:     if depth = 0 then
3:         return EVAL(node)
4:     else
5:         score ← −∞
6:         for all c, children of node do
7:             score ← − MAX(score, MINIMAX(c, depth − 1))
8:         end for
9:         return score
10:    end if
11: end procedure
```

3.2 Alpha-Beta Pruning

Alpha-Beta pruning improves Game-tree search by keeping track of the highest and lowest values that are guaranteed to each player. If a move is found to be worse than a score guaranteed by these values, the search of that branch terminates early.

```
1: procedure ALPHABETA(node, depth, α, β)
2:     if depth = 0 then
3:         return EVAL(node)
4:     else
5:         score ← −∞
6:         for all c, children of node do
7:             score ← − MAX(score, MINIMAX(c, depth − 1, −β, −α))
8:             α ← MAX(score, α)
9:             if α ≥ β then                                  ▷ β cutoff
10:                Break
11:            end if
12:        end for
13:        return score
14:    end if
15: end procedure
```

3.3 Pricipal Variation Search

PVS improves further on Alpha-Beta pruning by assuming that the first child of a node is it's "Principal Variation", and that it is likely to be the best choice. It runs a search on the other children with a narrow window (the Scout window), so they will cut off if they generate a value that is smaller than the principle variation. This allows for even more aggressive pruning.

```
1: procedure PVS(node, depth, α, β)
2:     if depth = 0 then
3:         return EVAL(node)
4:     else
5:         for all c, children of node do
6:             if c is the "Principal Variation" then
7:                 score ← − MAX(score, MINIMAX(c, depth − 1, −β, −α))
8:             else
9:                 score ← − MAX(score, MINIMAX(c, depth − 1, −α − 1, −α))
   ▷ Scout window
10:                if α < score then
11:                end if
12:                score ← − MAX(score, MINIMAX(c, depth − 1, −β, −α))
13:            end if
14:            α ← MAX(score, α)
15:            if α ≥ β then                                    ▷ β cutoff
16:                Break
17:            end if
18:        end for
19:        return α
20:    end if
21: end procedure
```

References

[1] Various Authors, *The Chess Programming Wiki*, www.chessprogramming.org