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

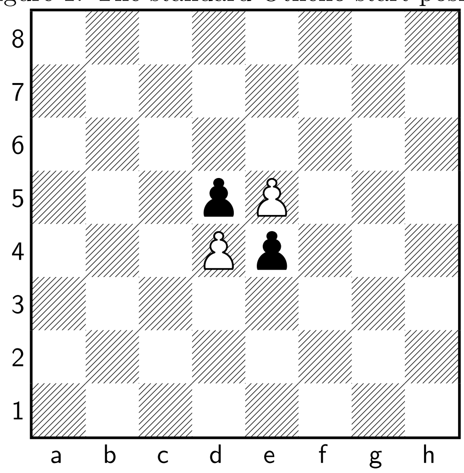
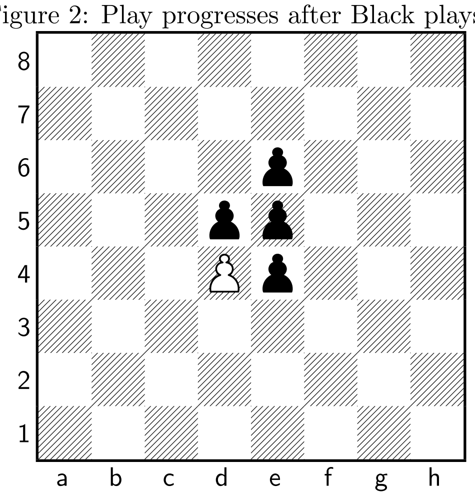


Figure 2: Play progresses after Black plays e6



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).

```
procedure Minimax(node, depth)
    if depth = 0 then
        return Eval(node)
    else
        score ← −∞
        for all c, children of node do
            score ← − Max(score, Minimax(c, depth − 1))
        end for
        return score
    end if
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.

```
procedure AlphaBeta(node, depth, α, β)
    if depth = 0 then
        return Eval(node)
    else
        score ← −∞
        for all c, children of node do
            score ← − Max(score, AlphaBeta(c, depth − 1, −β, −α))
            α ← Max(score, α)
            if α ≥ β then                                    ▷ β cutoff
                Break
            end if
        end for
        return score
    end if
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.

procedure PVS($node, depth, \alpha, \beta$)
    if $depth = 0$ **then**
        return EVAL($node$)
    else
        for all c, children of $node$ **do**
            if c is the "Principal Variation" **then**
                $score \leftarrow -$ MAX($score$, PVS($c, depth-1, -\beta, -\alpha$))
            else
                $score \leftarrow -$ MAX($score$, PVS($c, depth-1, -\alpha-1, -\alpha$)) ▷ Scout window
                if $\alpha < score$ **then**
                end if
                $score \leftarrow -$ MAX($score$, PVS($c, depth-1, -\beta, -\alpha$))
            end if
            $\alpha \leftarrow$ MAX($score, \alpha$)
            if $\alpha \geq \beta$ **then** ▷ β cutoff
                Break
            end if
        end for
        return α
    end if
end procedure

3.4 Shared Hashtable

Shared hashtable is a naive approach to adding parallelism to Game-tree search. We send of multiple agents to search the space, and cache results in a table. This allows for parallelism without the need for any added implementation complexity.

3.5 Jamboree

Jamboree is a parallel version of PVS. It utilizes the "Young Brothers Wait" concept, the idea that parallelism should be saved until it is needed and that searching the principal variation is inherently serial [**?**, jamb] Parallelism is used for scouting once the principal variation is searched, and any later variations that fail the scout are searched in serial as well.

procedure JAMBOREE($node, depth, \alpha, \beta$)
    if $depth = 0$ **then**

return EVAL(*node*)
else
score $\leftarrow -$ MAX(*score*, JAMBOREE($pv, depth - 1, -\beta, -\alpha$)) $\triangleright$ Search the "Principal Variation"
for all c, children of *node* **do** $\triangleright$ In parallel
score $\leftarrow -$ MAX(*score*, JAMBOREE($c, depth - 1, -\alpha - 1, -\alpha$)) $\triangleright$ Scout window
if $\alpha <$ *score* **then**
Wait for all previous parallel iterations to finish
score $\leftarrow -$ MAX(*score*, JAMBOREE($c, depth - 1, -\beta, -\alpha$))
end if
$\alpha \leftarrow$ MAX(*score*, α)
if $\alpha \geq \beta$ **then** $\triangleright$ β cutoff
Break
end if
end for
return α
end if
end procedure

References

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

[2] C. F. Joerg and B.C. Kuszmaul. Massively Parallel Chess. *DIMACS '94* October 1994.