Matt DeMartino

Jonathan Schoeller

Othello Writeup - Team Saint Inferno

Goal: Write a program to play Othello as well as possible. We did this by creating all possible moves and picking a move that will most likely bring us to the board position most in our favor. In other words, we followed the minimax algorithm with alpha-beta pruning (for efficiency). Our minimax algorithm used an original utility function to calculate the utility of the leaves of the state-space tree created.

Java Files:

- **Board.java**
    - Represents the board and pieces in the Othello game
    - This file stored the helper functions and all of the minimax functions related to the AI and storing the state of the game
- **Othello.java**
    - This file helped us actually play the game and loop through as if we were actually playing.

Utility Function: We calculated how much a board position was in our favor by using a utility function that considered:

- how many pieces of each color were on the board
- how late in the game we were
- The number of moves the opponent could make from a given position.

General Program: Our program starts by taking in the string by that's given to us and decoding it for use in our program. We then emulate the game internally. The computer takes in moves

from the opponent and makes moves based on that. We continue to play until someone wins.

Data Structures: Our data structures are pretty simple.

- We use a Board class (acts like a C struct) to hold variables that represent facts about the current board. As we look through different possible boards, delving deep into possible game states, we also use the board class to represent these possible boards.
- The physical board itself is stored simply as a 2 dimensional char array. We could not use a boolean array because there are three possible states for each position of the board. It could be occupied by a white piece, a black piece, or it could be unoccupied.
- Searching for the next possible move acts like a tree but isn't physically stored like one. We use a recursive minimax function (the one whose pseudocode was given to us) that creates the tree with its call stack where the root is the current game state, and this branches down for each next possible move. The height of the tree is how far we're looking ahead (default is 4)

An English description of your program's static evaluation function

We evaluate the utility of any state by considering a few important things. These are: the number of pieces on the board, the positions of these pieces (some positions are better than others), and the number of available moves the opponent had. Also, we considered how late into the game it was and weighted "number of pieces we controlled" higher as the game went on.

- Number of pieces on the board

We took the number of pieces on the board that we controlled - the number of pieces on the board that the opponent controlled and multiplied this by our time function. If the total number of pieces on the board is "t" then our time function f(t) is $f(t) = \log_2(t) - 1$. The minus one comes

from the fact that there are 4 pieces on the board to start with so we want to multiply by 1 on the first turn.

- Positions of pieces

We considered pieces that are on the side to have more value and pieces in the corner to have even more value. If a piece was on the side, we added or subtracted 3 utility if it was our piece or the opponent's piece, respectively. If it was in the corner, we added or subtracted 6 utility.

- Number of opponent's moves

We wanted to minimize the number of moves an opponent could make so we subtracted 3 utility for every move they could make.

To find the utility of any state, we added these 3 factors together and gave that game state that utility.

Justification of your design decisions, including your choice of programming language:

Frankly, we chose java because we both knew the language already. Java isn't necessarily the fastest language (which is very important for phase 2) but it is an easy language to use for object oriented programming which we are both used to. As for the choice of classes, having a Board class and an Othello class allows for an organization of the different functions. Those that were for the board itself, be it inputting moves or looking at future moves, were put in the Board class. While those that were necessary to run the game were kept in Othello. It allowed for easier understanding of the code on our end, and it made finding bugs a lot easier.

Any special features of your program

The program also has an ASCII board that we used for testing but it's nothing really special

How you tested your program

We tested the program by playing and seeing if it messed anything up. At first it was just making simple tests with known outcomes to make sure that little things worked, but once we had the basic structure of the computer-player, we just played the game to make sure it would work as expected. Considering I (Jonathan) lost a lot once we had the MinMax set up, I'd say it works just fine.

Citations to any resources you used in developing your program, such as articles about Othello strategy.

One of us (Jonathan) has plenty of experience playing Othello. This means that I know the general idea of strategy behind the game. You want to have as many pieces as possible on the board, while controlling what moves your opponent can have. This means controlling the walls/corners, putting huge emphasis on them, and keeping on eye on the number of moves you have access to verses the moves your opponent has access to.

Phase II

State Space Search changes.

To accommodate different levels of search based on time input, we had two options to choose from. One was the simpler, static experimental option where we would run some experiments and see how long it took to search down to each level then convert the time to a depth level at the very beginning of the program and simply always search down to that level. The other was a more dynamic approach where the alpha-beta optimized minimax algorithm would constantly

keep track of time and return when there wasn't enough left to keep searching. We went with the dynamic approach of always keeping track of the time we had left. We had to change the minimax algorithm to check the time remaining and exit early if necessary, and changed the make move function to use the minmax function in iterations.

Time Calculations

We changed our alpha-beta optimized minimax algorithm to now pass along 3 time related variables:

- Time Remaining
- Start Time
- Time Limit

Time was stored as a 64-bit "long" integer and the `System.currentTimeMillis()` method to calculate the current time as a long integer in milliseconds. From there, we could test how much time had elapsed by simply subtracting the current time from the start time. When that result was too close to the time we started at, we would return right away.

Phase III

A very simple update to the program to work for this: when given the total time, divide it by 30(the largest number of moves a side can make in a game), and make that the move time limit. Then run as in Phase II.