Independent Study End of Semester Report

Aaron Jencks

11/17/2020

Contents

1 Background

Reversi is an age old game that was invented in 1883 and later became known as Othello in 1971. Othello differs a little from Reversi in the way that the game starts; in Reversi, the players take turns placing the first four pieces, but in Othello, the first four pieces are already placed.

1.1 Gameplay

The game proceeds as players take turns placing new pieces onto the board, any disks of the opponent's color that are in a straight line and bounded by the disk just placed and another disk of the current player's color are flipped to the current player's color. Whichever player has the most disks of their color when either the board is filled, or there are no more legal moves, wins. There is only one additional rule, each play must capture/flip at least one piece.

2 Introduction

Many advances have been made in the world of chess computers since the late 1900s, but less have been made in the field of Othello computers. In order to decide whether creating an Othello AI is feasible, I've decided to improve on the statistics of the game. It is estimated that the number of legal positions in the game is $O(10^{28})$ but I hypothesize that the number is much less. So, I spent the semester making a graph walker to count the number of legal final positions that exist in the game. Where a final position is the state of the board when the game ends. For more information about Reversi you can visit `https://en.wikipedia.org/wiki/Reversi`.

3 Methodology

3.1 Theory

With the search space of the problem being so large, I needed to find ways to reduce it. So far I've found two ways to do this. Both of them have to do with the symmetry of the game.

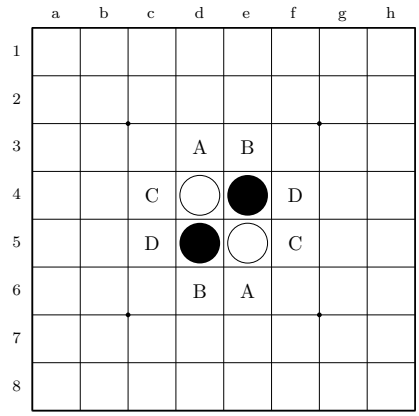
3.1.1 Color Reversal

Any state in the game can be switched to to the corresponding state in a game where the opposing color had started first by simply flipping all of the disks on the board. This is equivalent to rotating the game board by 90 degrees.

3.1.2 Rotation

There are four possible starting positions for each color. Any game started from any of the 2 positions from one corner can be moved to a corresponding move

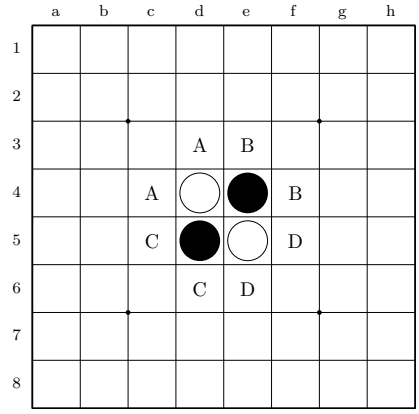
started from a different starting position by rotating all of the disks on the board 180 degrees. See the figure below:



Where each letter can be rotated to by the other square of the same letter.

3.1.3 Reflection

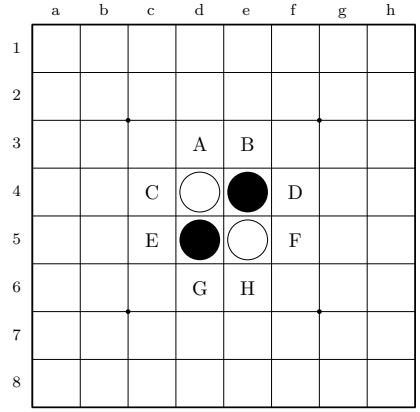
The last way that there is symmetry in the board is by using reflection across the central diagonals. Each game started from a corner of the initial four pieces can be converted to a game that started on the other square that shares the same corner, by reflecting the game board across the diagonal. See the figure below:



Where each letter can be rotated to by the other square of the same letter.

3.1.4 Full Circle

Using these 3 methods, I can show that, I can convert any game from any corner of the initial position to any other corner of the initial position, see below:



For simplicity's sake, I'm only going to use a single corner as an origin

- $A \to B$ Color Reversal, Reflection
- $A \to C$ Reflection
- $A \to D$ Color Reversal
- $A \to E$ Color Reversal, Reflection, Rotation
- $A \to F$ Rotation, Reflection
- $A \to G$ Color Reversal, Rotation, Reflection
- $A \to H$ Rotation

This also shows that there are set properties at work here. Because any position from A can be mapped to B and any position from B can be mapped to A, then that means that A and B are fully mappable, meaning that there aren't any games in B that I'd be missing if I only counted A

$$\forall b \in B \to \exists a(a \in A \wedge a \equiv b) \wedge \forall a \in A \to \exists b(b \in B \wedge a \equiv b)$$

All of this means that I can do a walk of all games starting at position A and if I multiply the resulting count by 8, then I will have the total count of all legal positions.

3.2 Implementation

3.2.1 Traversal

In order to traverse the game space, Depth-First search was used. This because the stack memory usage would be lower, on average there would be a little more than 64 entries in it at any time.

3.2.2 Revisitation

In order to prevent recounting of the same board state twice, a hashset was used. Unfortunately to limit memory consumption, only the final board states were inserted, all of the other states can be revisited as needed. The hashfunction I used was the Bob Jenkin's hash[1]. I also had to use nested arraylists, since the number of elements I was going to try and insert was going to be in the trillions.

3.2.3 Memory Efficiency

Originally when I was going to try and use Breadth-First seach, I needed to find ways to make the board memory usage as low as possible. I found that I can save the board in 128 bits using 2 bits per space. I originally thought that it could be done in 3 bits per every 2 spaces, but after looking at the truth table, I have realized that, that isn't true.

3.2.4 Parallel Execution

To speed up the search, I also implemented the search using the pthread library. The way that I got this to work is to start the search by using Breadth-First search to find $2n$ boards, and then assign each of those $2n$ boards to $2n$ pthreads, where n is the number of cores in the cpu. The hashset and the total final board state count are shared among all of the threads and protected with mutex locks, everything else is duplicated.

3.2.5 Checkpoints

I also created a method for saving the walker's progress in a binary file every hour so that I could stop and resume the program as needed. The syntax of the file is the threads' search stacks are saved with null bytes in between them, and then the cache is saved, along with the total count.

3.3 Challenges

There were a lot of challenges in this project, from data structure design to debugging.

3.3.1 Python

This project was actually originally started in Python, for ease of coding as well as debugging. But because of speed and memory limitations, it had to be converted to C. Which was a real challenge, because abandoning almost a full month's worth of work is always disheartening. I still wish I could switch back to using the multiprocessing library and having my memory managed for me automatically, but I'll also admit that the benifits are crazy.

[1] Source code from `http://www.burtleburtle.net/bob/c/lookup3.c`

3.3.2 Data Structures

Pointers are slow, and I've learned that the hard way, I've ended up having to reimplement much of this project multiple times because of that. In the beginning I used mainly linked lists because of the memory advantage, but quickly came to realize that I was going to have to use arrays instead. That wasn't the end of it, though, because *malloc* in C can only accept a *uint*32 as the size specifier, I ended up running into issues where I was trying to allocate arrays that were far to large, so in the end, I actually had to end up moving to a mix between the two, using nested arrays.

Hash tables are hard, finding ways to take an object and convert it into a key manually isn't easy and I went through several iterations of functions to get where I'm at now. There were also pointer issues as mentioned above, in fact my hashset for keeping track of state visits was the reason I had to start using nested arrays. I ended up using a bob-jenkins hash function in multiple takes, I needed to be able to use keys that were at least 98 bits if I wanted to guarantee that I wouldn't have collisions, so I used the bob-jenkins hash function to hash an upper and lower 64 bit sub key and then combined them into a 128 bit integer which was used as the key.

Parallelization was a challenge because it required rethinking how to approach the problem, instead of being able to just use DFS or BFS, I had to end up using both, in order to ensure that there wouldn't be significant overlap between the different pthreads.

3.3.3 Debugging

Debugging in C is always a challenge, even with tools like gdb and valgrind.

Unit Testing There is no unit testing framework in C, unless you want to integrate a third party library, so I had to build my own. It was mostly painless, because of function pointers, unit testing in C is actually quite easy, but that doesn't mean that the problems I was solving with it were. Most of my unit testing came down to fixing required functionality: testing array lists, linked lists, hashing functions, hashtable insertions, etc... Just searching for seg faults.

Memory Leaks The majority of my time debugging was spent tracking down memory leaks, which was no small task because even the smallest leak would cause a catastrophic hardware and software failure after a few days, which meant that valgrind needed to come back absolutely spotless if I had any chance in succeeding. Even now, there is a 2 byte leak that I can't seem to find, but at least it's not continuous, so until it actually crashes, I'm going to ignore it.

Multithreading I ran into several issues once I began multithreading the project. Debugging a multithreaded project is much harder than a single threaded one, and it turned out that one of the libraries I was using to help me detect errors wasn't thread safe, which caused me all kinds of headaches, until I figured out what was going on. Sometimes I still find myself converting my program to run on a single thread, just so that I can debug it easier. Handling locks and shared memory was also a new experience, and I'm glad that it went "mostly" smoothly.

4 Concluding Remarks

I'd like to remind everybody that this project started in Python, but my hand was forced into converting into C. There was almost a full month of development on this project in Python, and because of memory limitations, it had to be migrated.

Overall I've had a lot of fun working on this project, it's really opened my eyes to new experiences with C that I've never gotten a chance to do before, even if I didn't end up completing my research at this time, that doesn't mean that I'm going to stop, and I look forward to completing this project in the future.