OPERATION FLAMING CYRIL

Introduction

We intended to look at AI techniques for game-playing through checkers. While we had already explored the techniques that exist for perfect information games in class, we wanted to look at how they behave under different conditions. Specifically, we took the minimax procedure, which uses a static evaluation function to evaluate possible future game states and choose a move, and looked at how it worked for different heuristic functions on different checkers board sizes.

Task Definition

Examine how various heuristics perform within checkers at various board sizes. First, implement checkers and the minimax algorithm (with alpha-beta pruning) so that a player can be controlled by a human or a computer player using one of the heuristics. Starting with the American version of the game (8x8 board), have the computer play games against itself on other board sizes and record how each heuristics performed as board size grew larger. As time allows, look at variants on checkers (such as Suicide Checkers or Cheskers) and how the same heuristics perform on them at various board sizes.

Methodology

We used a standard implementation of minimax with alpha-beta pruning to allow heuristics to govern a computer player's behavior. A pseudocode implementation:

```
alphabeta(Board b, int depth, int alpha, int beta, Side s):

      moves = get legal moves for board b and side s

      shuffle moves

      if depth = 0 or moves is empty, return heuristic(b)

      if(side is black)

            for move in moves

                  board = apply move to b

                  alpha = max(alpha, alphabeta(board, depth-1, alpha, beta,
                  side.opposite)

                  if(beta <= alpha) break
```

```
        return alpha
    if(side is red)
        for move in moves
            board = apply move to b
            beta = min(beta, alphabeta(board, depth-1, alpha, beta,
            side.opposite)
            if(beta <= alpha) break
        return beta
```

The heuristic function called in the base case is one of many implemented heuristics for checkers. All heuristics counted number of pieces on the board, some weighted towards the edges, one weighted towards the middle and some weighted towards preserving kings. The start of the program gives the user a choice of heuristics for the computer player(s).

We allowed for boards to be arbitrarily large in our implementation, and gave users an option for the board size on startup. All boards have three rows of checkers on each side, staggered so that they are on the right squares of the board. Boards with size less than 7 are trivial - there are no legal moves, since the pieces cover all possible spaces, and boards with size greater than 56 are useless, since a draw will be reached before either player can make a capture. Most board sizes larger than 16 will lead to early draws as well, unless both players spend every move getting their pieces close enough to allow for captures.

We also had an implementation for “suicide” checkers. This variant on the game reverses the win condition: the first player to take all of their opponent’s pieces loses. We attempted to allow our heuristics to adapt to these new rules by negating the heuristic value, however, as documented in the literature, this almost exclusively leads to draws, since a win usually only occurs if one player has many more pieces than the other. We didn’t manage to do any data analysis of suicide checkers due to time constraints.

System Design

Our system was programmed using plain Java 1.8 without any external libraries. We used git for version control, and Eclipse as our code editor and IDE. Our use of Java informed our program structure, to a certain extent, although a number of different coding styles are present in the program. In its current state, the program is a console application taking input from STDIN and printing results to STDOUT.

Overall, our project had 4 main parts: the engine, human player, computer player with minmax algorithm, and the heuristics. The engine orchestrated the entire operation. It kept track of the players, sides, board, and win/loss/draw conditions. It abstracted the specifics of each player away, allowing for human vs. human, computer vs. computer, and human vs. computer matches to be easily assembled. Players are presented with their side, the board and a list of possible moves, and asked to present their choice of move.

The human player very simply presents the user with the board and a list of possible moves, and takes input from the player. The computer player, however, makes use of the minmax algorithm (with alpha-beta pruning), with the “help” of the heuristics, to select what it believes is the best move. In our quest for additional variables to tune, and for other obvious reasons, the maximum depth our minmax algorithm explores is configurable.

The heuristics (which were implementations of the abstract class ComputerPlayer), were, without exception, static evaluation functions over arbitrary boards. We implemented a variety of heuristics, including one that does nothing to help the minmax algorithm and one that actively steers the algorithm towards disadvantageous positions.

We built two interfaces to the engine as a whole. The first, which we showed in the demo accompanying the our presentation, presents an interactive console interface for running a single game to completion. The user is asked to select from the first and second players from a list containing the human player and the various heuristics. The second is a bulk runner, which runs a large number of games, varying the presets sequentially while suppressing board output. This was developed for our bulk analysis, and is not entirely fit for external consumption.

Results

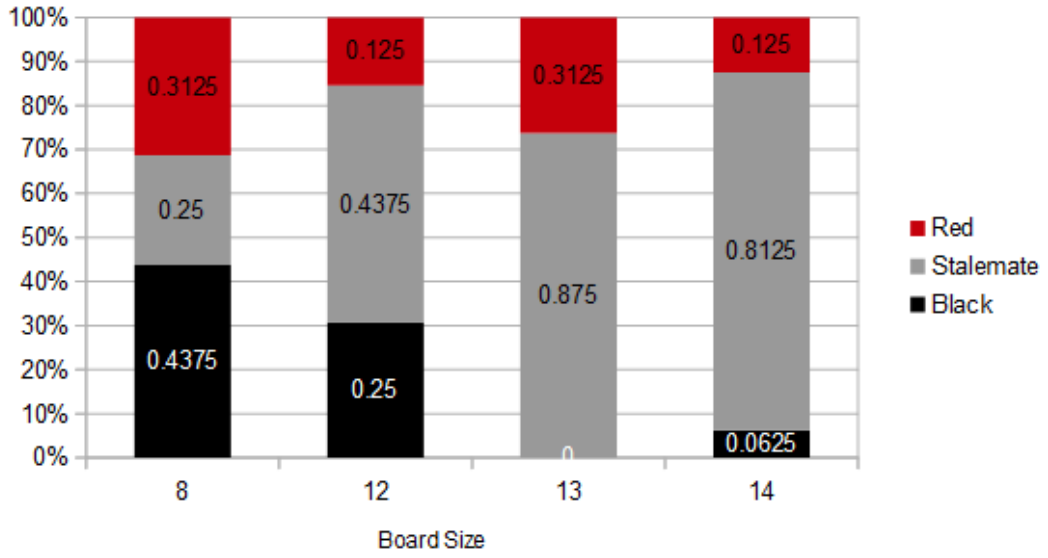

Figure 1: Percentage wins for Black/Red/Stalemate at different board sizes

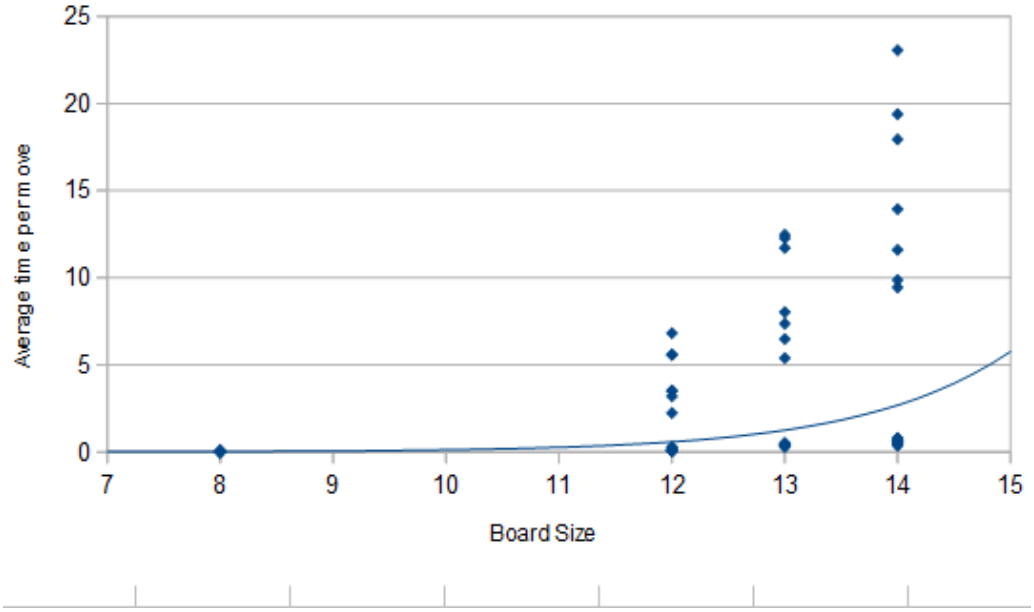

Figure 2: Average move times (in seconds) for each game at different board sizes with exponential trend line.

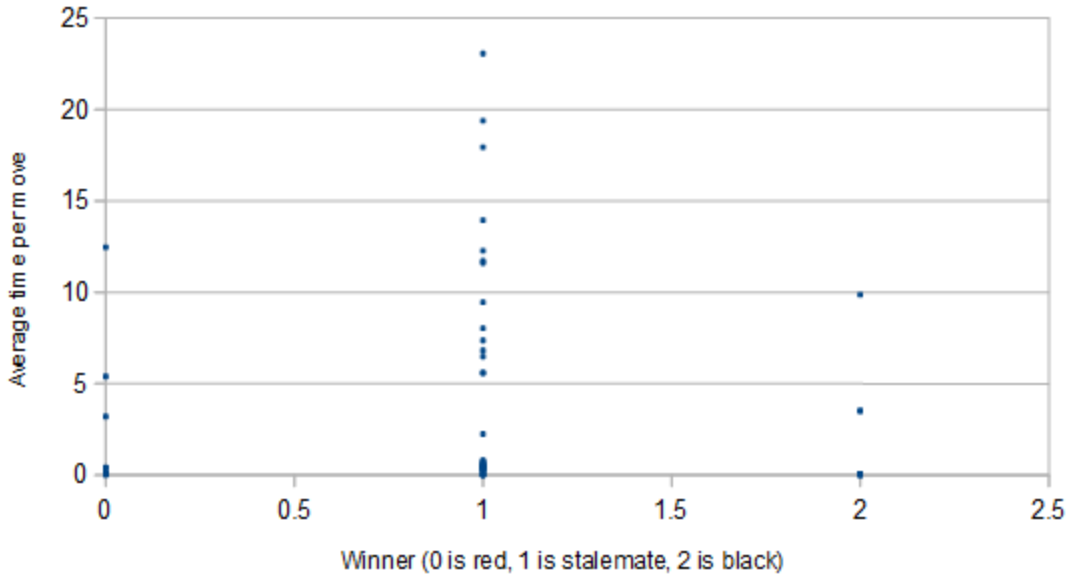

Figure 3: Average time per move based on game result.

Our raw and adjusted data is available in the repository alongside this file.

Discussion

Hardware and processing times ended up being much more of a limiting factor on our ability to gather data than we expected. Games, even without printouts, could take up to thousands of seconds to run, depending on the board size and depth. There is also a pretty clear exponential increase in runtime as board size increases (Figure 2), which was expected, since there are more pieces which add more branches to the minimax tree at the highest levels. Most of these large runtimes were from draws, because we defined a draw as fifty moves without a capture. As a result, we weren't able to modify our techniques and take more points within the time we had.

We realized while going through our data that we had an error in our bulk runner, so every game with WCCPlayer (the computer player which weighted checkers by their distances from the board edge) was played twice. In order to remove any possible skew that might have had on our results, we threw out every game that WCCPlayer played, which was unfortunate because it was probably the best of the heuristics we used.

All of our heuristics ended up performing worse at larger board sizes, which was not the result we expected, but makes a lot of sense, since they all end up drawing more often (Figure 1). The different heuristics had tendencies to find local board state maxima where every move other than moving a king back and forth generates a board state that is less ideal according to the heuristic. These situations would usually have the players calculating a very similar, complicated board state fifty times in a row, leading to higher than average move

times for stalemates (Figure 3). This tends to happen on larger boards because there are more pieces, and thus, more potential for suboptimal states once one side manages to get a king. For future trials, we can make our computer player generate scores for each move and choose one probabilistically in order to prevent it from staying on these local maxima for enough turns to generate a draw.

Another interesting case is board size 13 - our implementation gave an extra piece to the red side, which put black at a distinct disadvantage. This was reflected in our results, which showed that black won zero times on board size 13.

All in all, we made a lot of embarrassing mistakes with our gathering of data that could be easily fixed in a followup run of the project.

Related Work

Close to home, we implemented similar algorithms as the Reversi Group, although they took their ultimate project in different directions (and they implemented a GUI). Further afield, we encountered Chinook, a checkers AI designed by the University of Alberta. It was rather quickly retired from competitive play after winning too hard, and would later become unable to lose (although it would draw against perfect play).

There are also a large number of smaller commercial and open source checkers games, many of whom contain AI to play against, which range from rather simple to quite complex.

Checkers is ultimately a solved game, and a fair amount of research on it has been produced, which we didn't investigate very carefully.

Future Work

There are always more heuristics, and because we don't play checkers at a professional level, our insights into high level play is limited, which, to a certain extent, limits the play of our checkers engine.

Further, there are a number of quite bizarre variants of checkers which we didn't examine due to time constraints, although their popularity is rather lower than

The current single game interface is console based and quite ugly, compared to other graphical applications, so we could build a graphical interface.

Similarly, the bulk game runner's interface is even worse, as it requires editing its source file to adjust its activity, so that could do with some improvement.

Overall, due to the problem domain, our program runs rather slowly on large boards and large depths. Although we parallelized our bulk runner, our system architecture prevented

us from parallelizing across more physical cores than are present on a single machine, which for us was 4 cores at 3.75 Ghz. On that machine, the bulk processing required for this project still took approximately 15 hours, and ran only 138 games. If we had the chance to redo this portion of the project, we would use a more distributed architecture, and several more machines.

Our current programming style results in a rather memory expensive program, with a fair number of rapidly made and discarded objects. Reducing that number could reduce the number of GC pauses needed while running, but the effect of that would be fractional at best.

At the end of the day, this kind of optimization is outside the purview of this class, and outside our skillset as a whole.

Conclusion

Bibliography

Bodlaender, Hans. "Cheskers." *Chess Varients*. 1 Jan. 1995. Web. 21 Nov. 2014. <http://www.chessvariants.org/crossover.dir/cheskers.html>.

Hibal, Tim. "Playing Checkers with Minimax." *Project Blog & Code Musings*. 7 Feb. 2010. Web. 21 Nov. 2014. <http://tim.hibal.org/blog/?p=92>.

Mandziuk, Jacek, Magdalena Kusiak, and Karol Waledzik. "Evolutionary-based Heuristic Generators for Checkers and Give-away Checkers." *Expert Systems* 24.4 (2007): 189-211.*Wiley Online Library*. Web. 21 Nov. 2014. <http://onlinelibrary.wiley.com/enhanced/doi/10.1111/j.1468-0394.2007.00429.x/>.

Griffith, Arnold. "A New Machine-Learning Technique Applied to the Game of Checkers."*DSpace@MIT*. Massachusetts Institute of Technology, 1 Mar. 1966. Web. 21 Nov. 2014. <http://dspace.mit.edu/handle/1721.1/5896>.

Program

-Included. To run the bulk runner code, check the source file for the main executable (Game.java), adjust the boolean variable `op_nerf_plox` to `false`, and recompile. Compilation is left as exercise for the reader.