Hus Project Report

COMP 424

Jianhua Li

260351181

April 7, 2016

In this paper I will describe the approach I took to build a strong Hus player. I will likely make many parallels between Hus and chess because I have followed the progress of computer chess since early 2000s.

Choice of Algorithm

We shall first consider the size of the game in order to pick the right algorithm. At the beginning of a game, each player has to make a choice between 24 moves. After the first few moves, the choices of moves quickly decrease to around 12-13 by my rough estimate. By comparison, chess has an average branching factor of around 35, so Hus has a smaller state-space complexity than chess. The state of the art chess engines, which no human can beat, all use some variations of one algorithm, namely MiniMax algorithm with alpha-beta pruning. Other algorithms such as Monte-Carlo tree search, which works better for much larger games such as Go, will simply be counter-productive in this exercise. Additionally, unlike the game of Go where it's nearly impossible to handcraft a good evaluation function, it's much simpler to evaluate a Hus board, so again, using Monte-Carlo tree search probably isn't the best idea. After determining the best algorithm to be used, I started the implementation phase.

History of Implementation

Version 1:

One move look-ahead, choose the move with the highest heuristic value, it's already beating random player.

Version 2:

MiniMax with Wikipedia pseudocode, able to beat version 1 by searching deeper.

Version 3:

Alpha-Beta pruning with Wikipedia pseudocode, searching much faster than Minimax.

Version 4 and beyond:

Various improvements, to be discussed in subsequent sections.

Faster Search

The time complexity of Alpha-Beta at its worst is O(b^d), with "b" being the branching factor and "d" being the depth. The average case is around O(b^(3d/4)). If we simply take the move ordering from the getLegalMoves function, we'd search pits from the lowest numbered to the highest numbered, you'd think that is random which should achieve the average case. But after I simply shuffled the move list, my algorithm would run around 3 to 5 times faster than before! This was an astonishing result to me at the time, searching from lowered numbered pit to higher numbered pit wasn't so random after all. And it turned out that randomizing the move list has a benefit other than speed improvement, we will discuss it in the section of evaluation function.

The best-case time complexity of Alpha-Beta pruning is O(b^(d/2)), this requires perfect move ordering, which means that the best moves need to be search first. I used the heuristic that, if the immediate heuristic value of a move looks good, it'll probably be not too bad for deeper depth either. So at each depth I will sort the moves by the immediate heuristic value of the resulting board state, from higher to lower in a max node and lower to higher in a min node. The benefit of this is not immediately noticeable in low depth search, but once I use it with search depth of 8 and above, the speedup is around 2 times comparing to the random move ordering. Another big improvement I added was using a priority queue instead of sorting the array list of moves. Even though both are O(n*log n) operations, I observed 2 to 3 times speed improvement over list sorting.

Another improvement we can do is that once we approach the terminal nodes, it is not worth much to order the moves considering the overhead it incurs, so I only do move ordering for nodes that are 2 depths higher than terminal nodes, this particular depth was selected based on speed tests results. Now we have achieved speed up of over 30 times comparing to no move ordering; the exact speedup depends on the board configuration and the depth limit, in general I think that the higher the branching factor, the higher the speedup. Below are some benchmark results to illustrate the improvements.

Starting position, searching to depth 9	
No move ordering	72.2 seconds
Random move ordering	19.6 seconds
Move ordering using heuristics	4.6 seconds
Move ordering implemented with priority queue	1.9 seconds

A position that is 4 moves into a game, searching to depth 9	
No move ordering	40.3 seconds
Random move ordering	9 seconds
Move ordering using heuristics	1.1 seconds
Move ordering implemented with priority queue	0.5 seconds

Just having a fast search isn't enough, we also need better time management. In order to fully utilize the two seconds available for each move, I wrote an iterative deepening search starting at depth 8 but skipping depth 9, depth 8 is there to be an insurance in case the computer on which

the program will run is very slow. On my own computer, it pretty much always finishes searching depth 10, sometimes even depth 11 and 12. So I believe that my implementation of Alpha-Beta algorithm is fairly fast.

Evaluation Function

Crafting a strong evaluation function is very important to the strength of a board game engine. One lesson from chess programming is that once your program reached a certain strength level, each patch you make to the program will mostly likely worsen the program rather than improve it. So a good testing framework is very important in chess programming as is in Hus. One problem with a normal Alpha-Beta algorithm is the lack of randomness, which means that each time you let two non random programs play against each other, they will always play the same moves. Running more than once will not reveal any additional information about the strength of an evaluation function. Many moves might return the same heuristic value, with a non randomized move ordering, you can only test one of those moves. Here's where my randomized move ordering comes in to help, now two programs will have non deterministic result so we can take the win percentage over several games in order to have a more accurate picture of how good an evaluation function is.

Coming up with a better evaluation function wasn't an easy process, there were several anomalies that I was able to observe.

- The weirdest one was, evaluation function A was able to beat evaluation function B quite convincingly when running both at depth 6; but somehow when I increase the search depth of A to 7 or 8, it was only able to breakeven against B.

- Another anomaly is similar to the first one but reversed, one evaluation function performs terribly at depth 6, but suddenly starts crushing at depth 8 against opponents searching to the same depth.
- The third anomaly is that I often see my new evaluation function able to beat my previous best evaluation function, but couldn't beat a weaker one. So if A can beat B and B can beat C, it doesn't imply that A can necessarily beat C. This probably stemmed from the fact that none of the 3 is optimal so they exploit each other in different ways, one can sometimes observe this in the world of professional sports.

For a long time, I had no good explanations for them, but one day while actually observing the games, I was able to notice that in some games, the opening stage is so advantageous to one player that the rest of the game doesn't even seem to matter, the advantage gained in the first few moves were so huge that it's almost impossible for the other program to overcome. So a lot of the times when one evaluation function performs better than another, it's not because it actually evaluates the position more accurately, it's simply because it got lucky to get into a very skewed opening. This advantageous position isn't guaranteed to reoccur when playing against a different program. There are two conclusions we can draw from this, one, a good opening is very important, two, it's very hard to judge the strength of evaluation functions by just letting one playing against another. This makes testing extremely difficult, and coming up with a reliable automated testing framework is nearly impossible given the limited time we have. My evaluation function was entirely hand crafted.

I mainly tested the following parameters for my evaluation function:

- Number of seeds
- Number of moves available

- Seeds in outer pits worth more than those in inner pits.
- Seeds on the right hand side being more valuable than those on the left hand side

Eventually, only one of those was admitted into the final program, namely number of seeds. Based on my observations, the evaluation functions that incorporated the other parameters all contained many anomalies presented above, so I decided to go conservative and just use the simplest of them all. Without prior expert knowledge of the game, it's fairly challenging to come up with a robust evaluation function, by robust, I mean one that doesn't exhibit those anomalies I described and can do well against various evaluation functions.

Opening Book

In the previous section I briefly conjectured that the opening has a rather large influence on the outcome of the game. And in chess programming, a good opening book definitely improves the performance of the program during gameplay. The problem with opening books is that it's quite tedious to produce a high quality book beyond the first few moves. The approach I took is very simple, my book only contains the first move for player 0, the reply to the first move for player 1, and the second move for player 0. In order to compute the best opening move, I ran my program up to depth 16 which took about 6 hours on a desktop with a performant CPU (not as good of a searching algorithm back then). For the two subsequent moves I ran them to depth 15, it took one whole night to complete the computation. I don't think it's worth it for me to enlarge even further my book due to the complexity involved.

Approaches That Underperformed

One major implementation that didn't produce the results that I had hoped for was internal iterative deepening, here I will briefly describe what it is. Internal iterative deepening is to do a search to a shallower depth, then use the best move of that search as the first move to consider while doing the deeper search. But just having the one best move isn't quite enough, we need the principal variation. The principal variation is a sequence of moves that the program considers to be best for both players. So during the shallower search, my program will record the principal variation to be used for deeper search. Then during the deeper search, the principal variation will be the first branch searched. After testing, my implementation simply didn't provide the speedup promised, for example, the search to depth 8 using the principal variation from a shallower search, ran slower than my simple move ordering described in section "Faster Search". In my opinion, there can be two reasons for this, one, my implementation was probably far from perfect, two, the simple move ordering was a very good one, so the overhead incurred by internal iterative deepening is too big to overcome. But I still think that internal iterative deepening is something that can be useful in many board game engines, to read more about it, please refer to the references below.

Weaknesses and Future Development

My program does not take into account the stage of the game. For example, the end game where one player has a big advantage over the other should no longer use the same evaluation function as before; in chess, the player with a decisive advantage should be doing some sort of mate search. I have observed end game instances where mate can be achieved with one move, but instead my program dragged on for another 10 moves or so, so maybe something similar to mate

search should be implemented. But on the other hand, it seems almost impossible for a player with significant piece advantage to lose in such a scenario, so perhaps implementing mate search isn't worthwhile.

Improving the evaluation function would be the most beneficial improvement in my opinion, for example, my evaluation function isn't positionally aware, i.e. it doesn't take into account the position of my seeds relative to my opponent's seeds at all.

Using a more efficient data structure to represent the board would also be helpful in terms of speed. Creating a new HusBoard object each time we evaluate a move seems expansive, so keep modifying the same object might be a wiser choice. But if we want to make it even faster, we will have to use more optimized languages such as C; practically all modern chess engines are programmed in C or C++. The best chess engines also have the capability to utilize several CPU cores, so trying to parallelize my alpha-beta search algorithm is definitely on the to do list.

Conclusion

This was easily the most fun school projects I've ever done, I learned a lot even though I did not use any advanced AI techniques. With the future of machine learning, neural networks looking very promising, what does it mean for games like Hus and chess where historically, it's been dominated by hand-crafted evaluation functions. Maybe we will finally see the advent of strong self-taught evaluation functions. Needless to say, very exciting times ahead.

References

https://en.wikipedia.org/wiki/Minimax

https://en.wikipedia.org/wiki/Alpha–beta_pruning

https://chessprogramming.wikispaces.com/Internal+Iterative+Deepening

https://chessprogramming.wikispaces.com/Principal+variation