COMP 424 - Hus AI Project Writeup

Kevin Li - 260565522

April 8, 2016

1 Technical Approach

Prepared by creating abstract classes to represent evaluation functions so that they can be easily swapped out in order to test different ones and see which would perform better. All evaluation functions would return a score normalized from 0 to 100, which was important because this way we could easily return scores of 100 for victory and 0 for loss/cancellation.

Began by implementing minimax with a very basic evaluation function that just used the difference in pits between my agent and the opponent. More pits = higher score for me.

In testing it turns out that minimax would only search to a maximum depth of 4 on my machine; any higher depth would result in timeouts. It could beat the random agent around 90% of the time, which was decent, but when searching to a depth of 4, only around 150ms would be used in computation.

Clearly a lot of capacity was being wasted, so I implemented alpha-beta pruning in order to further deepen the search. I noted that in minimax, I was constructing the entire tree first of depth N and then backing up the nodes of that tree. This would not be efficient in alpha-beta pruning, as it would be wasteful to construct a tree and include nodes that would only be pruned off; in other words, it would be ideal to forgo creating nodes (i.e. running moves) where we don't need to. My new implementation would require this.

After implementing alpha-beta pruning, I was able to search a bit deeper (specified at a depth of 5). I had also improved on the evaluation function that I was using to take into account interior and exterior rows along with spaces that only contained one pit. But the depth parameter was exactly the shortcoming of this approach; I had to specify a depth to which we should search, and if we weren't able to discover a move by the time we search to the specified depth then the algorithm would never return a move in time.

After discussing with some colleagues I decided to implement iterative deepening in order to search to the deepest possible degree and thus try and search as deep as possible each time. Of course, in order to prevent the agent from wasting time and repeatedly searching shallow depths, it would need to start by searching to depth 4 or perhaps 5. I would also have to implement a way to determine a clever way of comparing the optimal move chosen from a shallower search with an intermediate move in a deeper search (i.e. in the case where we have completed a search of depth n, and the thread is terminated in the middle of searching in depth n+1). Further care would have to be taken in order to avoid as much as possible re-searching through already searched nodes.

Through observing my agent I noticed that towards the ends of games it would not take a move that would result in its immediate victory, and thus games would break down into these long torturous sessions of the agent chasing down opponent pits. It was through this observation that I discovered that in my alpha-beta pruning recursive function I was not taking into account whether the current state was a victory or loss state, which resulted in the agent searching beyond these terminal states. After having corrected this, along with lowering the computational overhead by reducing the use of unnecessary data structures (mainly Iterators), I was able to vastly reduce the number of turns it took for me to achieve victory against the random agent!

I then noticed an additional problem. As I was using iterative deepening, I was making the decision to throw away the previously obtained optimal solution from the previous depth of search and ended up using the newly generated move from the deeper depth, which theoretically should mean that the solution is better. But if the thread is interrupted before that new depth of search is completed, it is likely (and happens very often) that the 'new' solution is sub-optimal. I fixed this by searching the previous optimal move first, and then searching the remaining moves. This resolved the problem as now, even at a deeper depth with iterative deepening, a newly found optimal move could only be better than the previous depth's optimal move.

Later on, I played my agent several times and managed to win using a strategy that focused on capturing as many pieces as possible. Since I was able to beat my agent, it did bode well for my agent and I thought that I should encode the strategy that I used into an evaluation function.

My agent was still inefficient because when it searched through depth X and found several moves (improving upon moves that were previously thought to be optimal but were not) and then moved on to depth X+1, it would look through those same suboptimal moves in the same order. This meant that occasionally when the thread terminates at the time limit, the agent might consider a move that is known to be sub-optimal as optimal and return that move. In order to fix this, when an optimal move is discovered, the agent will re-order the states through which it searches at the top level. This makes it search the optimal move first, fixing this problem of choosing a suboptimal move first. Of course, if a previously suboptimal move turns out to be better at a deeper depth, this agent will still find it as it is still optimal because it will end up searching through the same moves, just in a better order.

At this point I turned around and tried to test a new technique that wasn't alpha-beta pruning. I implemented Monte Carlo Tree Search with a heavy rollout policy. I had learned from discussions with colleagues that light rollouts did not work well for this game due to the branching factor being low enough that minimax would be better. It turns out that it performed far worse than minimax with alpha-beta pruning, and it was likely because the heuristic I used in the rollout policy converged deterministically towards victory states (when playing against the random player). This was likely due to a complete lack of randomness, which was an error in my implementation.

The capturing focused evaluation function that I was using did not perform much better, and here is when I tried a new approach to try and use some other properties associated with being ahead; I implemented a new evaluation function that, in addition to counting the difference in number of pits between players, also favored states from which you could reach more states (unless it was a victory state). This evaluation function turned out to work better, and I was able to win against the random agent in around 12 moves.

At this point it was getting close to the submission date and I had fewer options left with respect to improvements. I performed one last test, which was to only explore a maximum of X number of nodes in during alpha-beta pruning (after ordering them with my evaluation function). After a small amount of tweaking (exploring only the log of the states, square root of the number of states, and a simple cap on the number of states explored), I settled on a cap on the number of of states explored, which allowed the agent to search quite deeply (up to 12, at least 6 steps ahead) without compromising the strength. The specifications of the finalized agent follow.

2 Specifications

Uses alpha-beta pruning along with an evaluation function that maximizes the number of pits along with the number of legal moves when it is my turn. At each stage in alpha-beta pruning the agent will order the next states by this evaluation function (descending if max, ascending if min) and only take the first 8 states to expand on in order to improve search depth.

Runs the search effort in a thread and waits for 1.9 seconds, then queries for the best move found so far. Best move setting and getting is done synchronously.

3 Advantages and Disadvantages

One great advantage is that for equivalently complex evaluation functions, this agent will be able to search deeper due to its limitation on the number of states expandeded. Although from a theoretical perspective this does not allow for optimal play, the evaluation function is good enough that it will be able to mostly eliminate the clearly bad moves quickly and not waste time expanding on those.

There are numerous disadvantages of my approach though. Due to my experimentation with heavy rollout MCTS, I wasn't able to parameterize and then tune my evaluation function, forcing me to stay with a relatively simpler one. Furthermore, there are no prepared moves, so if my agent encounters any other agent that has a lookup table of optimal moves, it will always be inferior when those moves are encountered.

4 Other Approaches

I tried using both heavy and light rollout policies with Monte Carlo Tree Search using a upper confidence bounds in the descent phase. Although the UCT method does allow the agent to win 100% of the time using both light and heavy rollout policies, on average it takes twice as long to win against the random player. Although one of the main benefits of using Monte Carlo is that it's easy to run in parallel and the best move can be obtained at any moment with ease, I felt that my multithreaded solution using minimax and alpha-beta pruning adequately handled the problem with interruption, and given that we don't have multiple cores available to us, there was little upside to using this technique.

5 Further Improvements

The agent does not fully utilize the data capacity. If I had more time, I would've wanted to create a compact bitwise representation of board states. (i.e. 6 bits per pit (no way we have 64 stones in one pit) and 1 pit to determine the turn player for a total of 385 bits (392 bits to be fully byte aligned)) and a bitwise representation of HusMoves (6 bits for pit number and 1 bit for turn player for a total of one byte with an extra bit). Then I wanted to take the last 6-7 moves from hundreds of games that my agent would play against the random player and exhaustively minimax search through all of possible moves to determine the best possible move for each state, and map states to moves for an end game lookup table.

I also wanted to try, instead of choosing the best possible move using the heavy rollout policy, instead create a skewed probability distribution by using the ordering produced by the evaluation function in the heavy rollout policy, and then sample moves from that in order to get a less deterministic rollout policy that still tended towards the results of the evaluation function.

Finally I would have wanted to implement transposition tables for starting positions and record the scores of every state reached in a proper game in order to speed up future searches. It's unlikely that this would be efficient though, given that I would need to record many positions. If I just used an end game lookup table, then I would only be able to (assuming source code takes up 250kb) using 9.75mb for storage. This would allow me to save a maximum of $\frac{49+1}{9,750,000} = 195,000$ board state to move mappings, not including the data structure overhead. In order to most efficiently use the storage available, it may be more ideal to only save the states that occur the most often. Afterwards, perhaps the map could be compressed to fully utilize the first move time and also slightly expand the capacity of the map. Compression would likely be able to save a lot of space as oftentimes there are pits with the same number of stones. Even though the pit struct would not be byte aligned, it may result in recurring 8-bit long patterns which could be compressed. If this is not the case, then an 8-bit pit representation could be tested to see if compression would work better.