BRIAN ZHOU

260381251

COMP 424

How my Implementation works and Motivation for my Approach

I will explain how my approach, a iterative deepening minimax algorithm with alpha-beta pruning and move ordering, works based on how I solved the different problems involved in implementing an AI agent for Hus.

Time Limit

Since each player is limited to 2 seconds of thinking time per player, I decided to not let my implementation fall back on the servers randomly chosen move fail safe mechanism since if the server deems my agent to be taking too much time, I would suffer an automatic loss.

In addition, a random move that the server picks is not a very optimal one. This has been quite true from observing an agent who didn't search as deep and respected the time limit every time playing against an agent who searched deeper but sometimes would timeout and fall back on random move selection. Only a couple of random moves can cause the deeper searching agent to lose the game almost every time.

Instead, I used an iterative deepening approach that searches from depth 1 to a very large number. This iterative procedure is ran inside an executor service[1] which would stop after 1980 ms. At each iteration, I update a result variable with the newly computed move index. The procedure does not return anything and runs forever until the executor service stops it. Once that happens, I simply retrieve latest result. By doing this, I can respect the time limit and take fully advantage of the properties of iterative deepening to search as deep or as shallow as time permits. Since the first depth involves only 1 level, it takes very little time to compute it, thus at minimum, I have a guarantee that my implementation would return a result of search depth 1. Using this method, now I am able to search deeper when the branching factor of the current board state is low and shallower when the branching factor is high.

Another way to save time is to make my computations as lean as possible. This means that I try to pull together computations that can be shared so they don't need to be performed multiple times. For example, retrieving legal moves is an arraylist operation involving many elements. This is costly so I would do this at the beginning of a method so that it can be used multiple times through this method. Another way to save on time is to minimize sorting times. Whenever I needed to sort something, I make sure to use Java's TimSort implementation as it guarantees O(nlogn) time complexity in the worst case.

---

1 http://stackoverflow.com/questions/4252187/how-to-stop-execution-after-a-certain-time-in-java

Finding the Optimal Move

To produce an optimal move for my agent, I first started with the naïve Minimax algorithm without a depth limit. This proved to be very problematic right away since my environment was not able function properly with the amount of resources it was consuming. The server would not choose a random move for me, nor would the server hand me an automatic loss. The whole process would hang forever. I then added a depth limit to my algorithm which solved this problem. However I soon found out that my program would only go up to depth 4 before the server started to give me automatic losses due to timeout. At this point my algorithm was already able to beat the RandomPlayer each and every time. However I wanted to make it better given the competition out there. So I decided to improve my search depth. The obvious solution taught to us in class is alpha-beta pruning. So I went ahead and augmented my minimax with alpha-beta pruning. Unfortunately it didn't make things better my for agent. Even with pruning, the branching factor was still too big for my agent to not suffer timeout losses. This is when I decided to apply a move (or more precisely, successor) ordering routine to my alpha-beta pruning in the hopes of achieving deeper search. I essentially used my evaluation function to rank each of the successors at each recursion level. I would expand the successor states with bigger heuristic values first in the hopes of increasing pruning. This allowed me to push search depth from 4 to 5 without relying on the random move selection for more than 1 or 2 times per game. Finally, I noticed that this approached sometimes caused my agent to timeout and other times it caused it to complete the calculation way under 2 seconds. I then decided to implement an iterative deepening version of what I already have. By iterating from depth 1 to a very large number, I was able to compute an optimal move based on a whole range of search depths. Combined with my time limit solution, I was able to take full advantage of the 2 seconds given and prevent relying on the random move fallback.

Evaluation Function

In order for my alpha-beta search algorithm to pick out the most optimal moves, I had to use an evaluation function that depended on domain knowledge. Since I my focus was on searching deep and fast rather than shallow and thorough, I wanted an evaluation function that is simple and easy to calculate. In fact, my evaluation function is solely based on the current board state. I simply take the difference between the seed count of all my agents pits that have more than 1 seed and the seed count of all the opponents pits that have more than 1 seed. This seemed logical since the more seeds my agent has means that the more seeds were captured from the opponent, which in turn means that the amount of moves my opponent can make is reduced. This is good since we win if the opponent cannot make any moves. I also used this very same evaluation function for terminal board states because terminal board states are states in which the losing player has no pits with more than 1 seed and the winning player has captured many of the losing players seed. This evaluation function works well enough to beat RandomPlayer every time.

Ram Limit

Ram is another concern that could destroy my agents chances before its quest even began. In order to stay under the 500MB of memory allowed, I have to be extremely space efficient with my computations. For example, I would use shallow copies to duplicate data as much as I can. This allows

the same objects to be pointed by multiple references so the actual object does not need to be duplicated themselves wasting memory. This solution is applicable for those times where I simply read and not write those objects. Another way of saving space is to pass arguments by reference instead of by value. This is because passing arguments by reference only duplicates the pointer and not the object itself when its the other way around for passing arguments by value.

Theoretical Basis for My Approach

Naive Minimax

Naive minimax is the base algorithm for my approach. I implemented a minimax algorithm pseudocode found on the minimax wikipedia page[2]. It is almost the exact pseudocode as the one presented in class. Essentially, naïve minimax is a tree search algorithm that has the current game state as root node and all its possible successor states as its children. These successor states are derived by applying a legal move on the parent state node. A node is a max node if at that state, its the agent's turn to play. A node is a min node if at that state, its the opponents turn to play. The idea is to give a good score to a terminal state where the agent wins and a bad score to a terminal state if a agent lose. A max non terminal node would have a score corresponding to the maximum score of its children's score and a min non terminal node would have a score corresponding to the minimum score of its children. This is logical if we assume that our agent should always pick the move that results in most score for the agent and the opponent would pick the move that results in the least score for the agent. This score backing operation occurs all the way until we find a score for all the direct children of our root node. We then pick the child with the largest score. This means that the move that produced that child state is the move that would potentially give our agent the most score and hence the most chance to win the game.

Depth limited Minimax

Naive minimax is a nice approach. However the game of Hus has a very large branching factor that stays relatively constant from start to finish. So running the naïve Minimax is not an option. Depth limited Minimax[3] involves doing the same procedure as naïve minimax, however, once the specified depth is reached, we return the evaluation function of that state rather than continue with recursion until a terminal node. This allows us to search a couple of moves down but not all the way down.

Alpha-Beta Pruning

Depth limited minimax allowed me to beat RandomPlayer, but I want it to beat other AI agents as well. Minimax with alpha-beta pruning produces the same optimal result as dept limited minimax but without us needing to expand as many nodes. This is achieved by cutting off a branch when we know that no matter what the rest of the children on that branch would score, it would not change the score of the parent. For example, if a min node has a child with score 2, and two other unexplored children, and the min node has a neighbor node with score larger than 2, then we can cut this whole min node off

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

because even if the other two children produces scores lower than 2, the parent max node can pick the min nodes neighbor with score larger than 2 and avoid this min node altogether. This procedure cuts down on the branching factor of our search tree allowing us to search deeper. My implementation was based on a pseudocode on the alpha-beta pruning wikipedia page[4].

Move ordering

The amount of branches that alpha-beta pruning can cut off depends on the order of the children visited. If we find a child of a max node with a very large score earlier on, then we can cut off all branches with score lower than it that we visit afterwards. Analogously, if we find a child of a min node with a very score early on, then we can cut off all the branches with score higher than it that we visit afterwards. We can achieve this in our depth limited minimax with alpha-beta pruning algorithm by sorting the successor states of a node in ascending order if the root node is a min node, and by descending order if the root node is a max node, at each recursion level. This sorting done by computing the evaluation function of each child node directly to give a rough estimate of their ranking in the hopes that the successor that is ranked first according to our direct evaluation function would in fact produce the highest/lowest score once we run the search all the way to the bottom. My implementation of Move ordering is based on the first answer to a stackoverflow question[5].

Iterative Deepening

Iterative deepening is a hybrid of depth first and breadth first search. The algorithm first performs a depth first search at a given depth limit, then subsequently performs additional depth first searches at incremental depth limit. This method combines the benefit both BFS and DFS. However, its application is particularly useful in the case where we have a time limit on the computation and we want to take full advantage of it as explained in the first section of this writeup. I implemented iterative deepening based on a Will Thimbleby's algorithm wiki[6].

Advantages and Disadvantages of My Approach

Advantages

The advantages of iterative deepening minimax with alpha-beta pruning and move ordering is that the optimal solution is produced and it is done in a way that minimizes the amount of nodes expanded. This allows us to search a couple of moves ahead of naïve minimax or even alpha-beta pruning without move ordering since we are cutting down on the branching factor.

Using Iterative Deepening allowed me to take full advantage of the 2 seconds computation time as well as prevented me from relying on the random move selection fallback which produced less than optimal

4 https://en.wikipedia.org/wiki/Alpha-beta_pruning
5 http://stackoverflow.com/questions/9964496/alpha-beta-move-ordering
6 http://will.thimbleby.net/algorithms/doku.php?id=iterative_deepening_depth-first_search

moves. Using Iterative Deepening is also very advantageous against an agent with a fixed search depth. I found through simulation that if 2 search results differing by only 1 depth, it can make a big difference in terms of how good these results are. So if I can continuously search deeper than my opponent by even just 1 depth at every turn, there is a big chance that I would come out on top. This of course is only valid if both agents are identical in every other aspect.

Running the whole procedure in a executor service allowed me to control my computation time by preventing me from getting disqualified due to timeouts.

Using TimSort for move ordering is advantageous because without it we could not perform move ordering and any lesser sort algorithm cannot guarantee a better time complexity that we require for search deepening.

Disadvantages

minimax and alpha-beta pruning is mostly effective for games with low branching factor. This is the reason why using it on Hus (branching factor of approximately 32 in our specific case) would limit the depth it can go. Perhaps it would not perform as well as the Monte-Carlo search tree method which allows for much deeper searches.

My evaluation function is quite simple. I have only used 1 single element of the domain knowledge which allows me to search faster. However, My agent might benefit perhaps from additional domain knowledge that I am not aware of. This could significantly impact my agents performance against AI agents who does have superior domain knowledge embedded in their evaluation function.

TimSort used for move ordering is very good in terms of time complexity, but its lacking in terms of space complexity when compared to QuickSort. This is because QuickSort is a in-place sorting algorithm that only takes up O(nlogn) space in the worst case while TimSort is O(n) space in the worst case. This might impact my RAM limit but so far I have not encountered any problems using TimSort given my implementation only has at most 32 elements to sort each time.

My approach leaves no room for randomness due to my own personal aversion to them acquired through simulation. However, my agent may benefit from the occasional random move if the opponent can be thrown off by it. Unfortunately I lack the domain knowledge to know exactly how to introduce such an element into my AI.

Comparison with Other Approaches

I have tested my Iterative Deepening minimax with alpha-beta and move ordering agent against my depth-limited minimax. The iterative version averages a search depth of 6 to 7 while the depth limited agent can only search at exactly 4. The iterative deepening agent is able to pretty much win every time.

I have also tested my depth limited minimax with alpha-beta and move ordering against my iterative version. My depth limited agent has a search depth of 5 which is pretty close to my iterative agent. However my depth limited agent would some times fall back to a backup move computation that

involves a depth 1 search. One or two of those depth 1 search moves pretty much guarantees a loss against my iterative agent who pretty much always turn up move selections from search depth 4 and above.

Lastly, I have implemented a iterative deepening version of naïve minimax to play against my iterative deepening minimax with alpha beta pruning and moves ordering. The games go on for longer times but my iterative deepening minimax with alpha beta and moves ordering won 5 games out of 7.

Improvements to my Approach

The biggest improvement I could bring to my approach would be to improve my evaluation function with additional domain knowledge. I suspect it would allow my agent to make better move choice decisions.

Another improvement I can offer would be the way I order moves. My approach was to rank them directly based on evaluation function. Another way to do would be doing a very shallow search on each move instead. This would give the ranking more accuracy perhaps.

If I had more time, I would try out Monte-Carlo Search Tree so that I could search deeper to see if it improves the performance of my Agent. This improvement would be particularly interesting because the difference between searching all the way to the bottom of the tree compared to only 6 to 7 moves ahead maybe produce a big difference in terms of performance.