COMP 424 – Artificial Intelligence
Project Report

Guillaume Labranche (260585371)

due on April 8, 2016

1 Description of my agent

I made my agent with the minimax algorithm. I used alpha-beta pruning and move ordering and iterative deepening to maximize the search depth. I chose to go this way after discussing the properties of Hus with classmates:

- Hus is a two-player game, which applies well for minimax since it is easy to reason about.
- Hus is a zero-sum game. This means that for any points we earn, the opponent loses the exact same amount.
- Hus is a full-information game, meaning we have access to the entire game state and therefore do not need to reason under uncertainty.
- Hus is fully deterministic. This means that future game states are entirely based on the actions of each player and there is no randomness involved. Probabilistic reasoning is therefore not obligatory.
- Although the short-term outcomes of a move are easy to compute, the long-term effects are hard to predict without actually playing out the moves. Unlike in chess where we can statically see that a well-protected piece is a good feature to have even for the long term, in Hus the landscape of the board can change so drastically in such few moves that most static reasoning made is irrelevant past a few moves. There is a low correlation between a short-term gain and a victory. This is why humans are so bad at this game, even against a minimax limited to depth 3 and especially if a human has to follow the 2-sec turn time limit.

All of this lead me to conclude that an approach such as training a neural network that would take the board state as input and would return which move to play would not work well. Another possible approach is Monte-Carlo tree search. I decided against it because games played at random can last very long. It is almost a coincidence that the game ends. Unless Hus is played optimally to minimize losses, one mistake (such as a timeout causing a random move) can drastically alter the strength of the players' position.

Minimax seemed to be very well suited for Hus and I would not be surprised if the top of the leaderboard after the contest is mostly minimax agents.

2 Features of the agent

2.1 Evaluation function

My evaluation function is rather simple. The value v of a board is:

$$v = 1 \cdot \sum_{i=0}^{31} p_{s,i} + 1 \cdot |m \in p_s : m \geq 2| + 1 \cdot (32 - |m \in p_o : m \geq 2|)$$

where p_x is the score for player with id x, $p_{x,i}$ is the number of seeds in pit $\#i$ of player x, and s and o are the player ids for my AI and its opponent respectively.

Special cases:

- $v = \infty$ in the case that the board state is a win for my AI.
- $v = -\infty$ in the case that the opponent wins or that we created an infinite move (which could make us lose if done too many times).

The feature counting the number of possible moves by the opponent is very useful. Given a random valid board state and a possible move, the true value of that move is very hard to predict. It can only be calculated using minimax or MCTS. This lead me to the conclusion that the value of the best move possible is directly correlated to the number of moves we are allowed to make. Then minimax is performed in order to find which move exactly is the best one. Similarly, having a low number of possible moves makes a player very limited in its choices and therefore lowers the chance that one of them is very good.

The reason for having a simple (i.e. fast run time) evaluation function is that there is nothing to be gained by putting more gameplay analysis when the same information could be gained by going one level deeper. Making a slower evaluation function would effectively reduce how many states we can evaluate in our fixed amount of time, and therefore how deep we can perform our search.

2.2 Minimax

I implemented minimax in the standard way, which returns the best score at the depth at which it is exploring. I later modified it in order to return the move associated with that best score. The minimax function is also passed a depth, in order to limit how deep it will search and therefore return a move in time.

2.3 Alpha-beta pruning

One way to enable the search to go deeper is with α-β pruning. For example, suppose we are currently trying to optimize our own move by maximizing the best score of each possible move. Let α be the current best score. Note that through the search, α's value only goes up. For a given move, we then go simulate our opponent's moves and try to find the move with minimal score. If we find a move who's score is lower than α, then we will inevitably return a move with that score or lower since we are minimizing. And since one layer above we are maximizing, then that move will not be the best since the best has a score of α and our move has a score less than or equal to α. Therefore we do not need to search further for this move and can simply return what we've found so far knowing that it will have no impact on the result.

2.4 Move ordering

Essentially α-β pruning stops searching down low-scoring paths based on the best move found so far. This means that the sooner we find (i.e. fully explore until depth limit) a high-scoring move, the more moves get pruned out of the search and therefore the faster we complete the search, which then means we have more time for searching deeper. Therefore at each level, the possible moves one can perform are sorted in ascending and descending order depending on whether we are minimizing or maximizing. We the process each move (i.e. perform minimax on them) in order.

2.5 Iterative Deepening

In order to return the best move within a 2s time limit, the naive solution is to set a conservative fixed maximum depth that will exhaustively search the tree in under 2s in the worst case. But we can often do better than such depth since the max depth depends on a lot of factors which cannot be predicted:

- The max depth depends highly on the branching factor. Since the number of possible moves is constantly changing, the depth at which we have time to search varies.
- We cannot predict how many branches are pruned by α-β pruning.

Because of those factors, we cannot dynamically decide on a maximum depth. The best option is then to keep searching until we run out of time, and return the best move so far. Minimax performs depth-first-search on the game tree. Because of that, if we increase the depth we must then start from the root. It would be possible to modify minimax to use breadth-first-search but memory usage would be very big. Minimax is also designed such that the best move found so far is really only the best move if the tree has been completely searched. For example it is better to choose the best move when the game tree has been fully explored to depth 5 than when the game tree has been partially searched to depth 6, since it may ignore really good moves by either player.

Therefore we rely on a technique called iterative deepening which iteratively increases the depth at which we perform the search. One drawback is that we end up searching the top part of the tree multiple times while searching at increasingly deeper levels. One could memoize the tree and associated game states but I did not do that.

3 Advantages and disadvantages of my approach

One advantage is that it finds the best move theoretically possible while minimizing the opponent's gains. Another advantage was that it requires very little Hus-specific knowledge except for the evaluation function. This means that I was able to implement the algorithm without playing the game a lot myself or having to look deep into the mechanics of the game and its implementation in the provided codebase. It is also very portable and could be applied to other games with little modification.

The disadvantages are that it does not go very deep even with all the optimizations applied to the minimax algorithm because the branching factor results in exponentially many nodes to search at each level. This essentially creates a "horizon" effect where our agent is completely unaware of the consequences of its moves past a depth of about 10. It therefore has no long-term strategy except the one embedded in the evaluation function,

which is to simply gain more seeds. This also leads it to sometimes being mistaken about a move which appears the best when searching to a certain maximal depth but then leads to a catastrophic situation right after.

4 Unsuccessful attempts

I initially implemented iterative deepening using a timer. More specifically in the main thread I started another thread to search for moves which the main thread slept for 1.9s. The multi-threading was causing issues with regards to printing and there were other bugs associated with sharing the results between both threads, as well as stopping the searching thread once time was out. I therefore decided to remove all multi-threading and instead rely on a call to System.currentTimeMillis() being called periodically. This turned out to be pretty efficient.

5 Possible improvements

Many improvements could be made in order to make the AI better. First and foremost would be a better evaluation function. This would not only lead to choosing better states, but it would also lead to a deeper search since α-β pruning's efficiency relies on the effectiveness of move ordering which is based on the evaluation function. In that department, we could learn the best static weights of each feature using machine learning or other means. Furthermore, we could design a dynamic evaluation function whose weights are dependent on other factors themselves such as how far we are into the game or certain features of the board state. This could lead for example to an agent which is more aggressive at the beginning of the game rather than the end, or more defensive when it has a low amount of seeds.

Another improvement would be to run minimax and precompute a lot of the information that ends up being computed at run-time during games. We could fully explore the game tree of states that are near the end of game and compute their score beforehand and load this into memory at the beginning of the game. In this contest it would be hard to achieve due to the memory constraints imposed on all submissions, namely the total submission size of 10MB.

Another improvement which seemed to legally "break" the rules would be to set up our agent on a powerful supercomputer and submit only a lightweight agent who queries our supercomputer agent over the network. This would undeniably add network overhead but considering how much faster we would arrive at a solution, it would still be better. I opted against doing this since it would be very reasonable for the graders to run the simulation without an internet connection. I suggest clarifying this point in a future iteration of the course. I also wonder if any student dared to attempt this.