Comp 424: Artificial Intelligence
Final Project

Geoffrey Saxton Long
Student #260403840
Geoffrey.Long@mail.mcgill.ca

April 3, 2016

1 Introduction

Hus is a game that lends itself well to AI development. It is from a family of rather well studied Mancala games, although Hus itself is rather unstudied. There are several variations of Mancala, but they all share the basic principles. Specifically that the board has holes which are ordered into rows; the game is played with indistinguishable tokens (henceforth referred to as seeds); each player owns a fixed and equal number of holes on the board; a move involves taking the seeds out of a selected hole and placing them one-by-one in subsequent holes (this is often referred to as sowing); the sowing may or may not terminate with capture conditions which allow a given player to seize another's seeds; and the goal of the game is to capture the most (or all) of the seeds [**?**]. In Hus, there are 4 rows of 8 holes a piece, and each player owns the 2 rows closest to them. Each player begins their turn by choosing a hole with more than one seed. If they cannot do this, they have lost the game. All of the seeds are taken from the hole and sowed one-by-one in a counter clockwise manner starting with the hole directly after the initially chosen hole. If the sowing ends on a previously occupied hole, then the seeds are taken. If the previously occupied hole is in an inner row, then the opponent's seeds on the same column can also be seized. If there are seeds to be sown, then the player continues as before.

There are several reasons why Hus is a good candidate for an AI agent. Since the problem has a discrete state space, the search will be over a finite set of different game states. The changes between states in this state space is deterministic. This means that for each player action, we know exactly what the outcome will be. We might not know the full effect of the move due to the execution time constraints, but we know the immediate effects and how this alters the game flow over a finite span. This lowers the complexity and ensures that we will know the exact state of the board given a set of moves. This allows us to plan more effectively and with more certainty. The static and observable nature of the environment affords us similar benefits. Since the information on the game is perfect, planning can be performed with greater certainty.

The game rules themselves also help to make planning easier. There is a finite set of moves that can be applied at each turn, each with the same basic operator. Each player has the same simple moves and has the same simple goal, so it is easy to see the relationship between the players. The immediate cost of a player's action can be easily observed by counting the number of stones gained or lost. The simple rules, few operators, and observable cost all make it easier to derive a well rounded set of heuristics for estimating the quality of a move. The heuristics (discussed in Section **??**) are really what drives the AI agent to the goal quickly.

As can be seen, Hus is a good candidate for AI development. The remainder of this paper discusses an approach taken towards a suitable AI agent.

2 Background

A theoretical basis for the approach was derived by parsing articles on AI approaches in the Mancala family of games. Although no papers were found on Hus specifically, Mancala and several of its derivatives did have AI research. Since the Mancala games all have similar mechanics (described in Section **??**), many of the approaches are extensible to Hus. Specifically, an algorithm that works well for Mancala can be expected to work for Hus as well. This extensibility is mostly due to the simple and shared game operators. In any Mancala variant, a player will take seeds and distribute them evenly amongst the subsequent holes. The effect of a given action is rather similar across the entire family of games. Also the branching factor, representation, goals, and states are also quite similar among the family. Since the state, operators, goal, and cost are similar along with the environment, we can expect similar algorithms to perform similarly. That being said, variants that have mechanics that more similar to Hus's offer better indications.

Amongst all the strategies employed, MiniMax variants were the most common. Specifically MiniMax with alpha-beta pruning which relies on an evaluation function consisting of a linear combination of weighted features. Minimax variants were used in nearly all of the papers read on Mancala games.

Abayomi et al. studied a series of supervised machine learning methods in "An Overview of Supervised Machine Learning Techniques in Evolving Mancala Game Player". They focused on developing AI for the variant Awale. The majority of the methods were employed using minimax combined with some sort of machine learning. The machine learning was used to classify moves into two classes, one which helped the player and another that helped the opponent. The move was chosen which was the farthest away from the separating hyperplane. The classification was learned several ways including case-based reasoning using a perceptron; features using linear discriminant analysis; and linear discriminant functions learned with a perceptron. They also employed evolutionary computation to learn the evaluation function using both co-evolution and genetic algorithms to mixed success [**?**]. In a similar paper Randle et al. studied unsupervised methods in "An Overview of Unsupervised Machine Learning Techniques to Evolve Mancala Game Player". They initially attempted retrograde analysis, which takes a bottom up approach to scoring, but this was deemed too expensive. Then, similar to Source [**?**], they used learning to create a binary classifier separating good and bad strategies. Again, the move was chosen to maximize the distance to the hyperplane. The machine learning used was the Aggregate Mahalanobis Distance Function (ADMF), and it was used with minimax with a depth of 6. This method appeared to outperform the supervised methods seen earlier [**?**]. In "Searching & Game Playing: An Artificial Intelligence Approach to Mancala", Gifford et al. were able to achieve a 100% success rate by varying weights on a set of parameters used for an alpha beta minimax evaluation function. They found that small changes in certain weighted heuristics can make a big impact on the outcome of the search [**?**]. Other authors chose to use iterative deepening with memory enhanced test driver (MTD(f)), which is a variant of

alpha-beta minimax which searches using only zero-window windows. MTD(f) is often considered one of the more speed optimized versions of minimax. It was used for BAO by Donkers et al. in "Programming Bao" [**?**]. It was also used by Irving et al. to solve Kalah(6,5) in the paper "Solving Kalah" [**?**]. As is shown, minimax and its variants work well for Mancala style games.

In addition to the overall strategy, the papers also showed possible evaluation functions and heuristics. Although a majority of the features are not directly applicable to Hus, many could be adapted to optimize a Hus evaluation function. A list of relevant and possibly adaptable features is shown below. Note that some of the features have been altered slightly to align more closely with Hus's objectives:

- The number of pits that we / opponent can use to capture seeds [**?**]
- The number of pits with more than X seeds on the our / opponents side [**?**]
- The current score of our player / opponent [**?**]
- The number of empty pits on our / opponents side [**?**]
- number of holes in the inner row that are filled [**?**]
- total number of counters in the back row [**?**]
- How far ahead of my opponent I am ("good heuristic") [**?**]
- How close I am / opponent is to winning (¿ half) [**?**]

As with any heuristic, these features provide insight to the game status. Many of the features only weakly classify the moves as beneficial/detrimental however, when combined into a linear combination, they can form a rather strong classification of move quality. Note that the list above is is just a subset of the heuristics found. Many of the other heuristics can be used to draft heuristics for Hus. A further description of heuristics can be found in **??**.

3 Algorithm

As was stated in Section **??**, one of the more common approaches was MiniMax with Alpha-Beta pruning. This algorithm was a logical choice for my AI agent because it is proven for this problem, it can be tailored using an evaluation function, and it is rather easy to implement. The most difficult part of the problem would be in crafting the evaluation function.

3.1 Ensuring Timing

A big component of this project is adherence to the strict timing guidelines. As was stated in the project specification document, the agents have 30 seconds for the first move, and 2 seconds for each subsequent move. Although one could theoretically perform timing analysis on their program to ascertain an average depth of their game tree or number of iterations on their algorithm, this method is ineffective. The issue is that different environments and even different games could cause drastic changes in the timing. If, for instance, the branching factor is much larger for a given instance, the agent could run out of time before the tree is fully searched. On the other hand, if the branching factor is small, then the agent could end up returning a value prematurely, when they could have searched farther down the game tree. An effective timer would perform the following (in order of priority): stop before the time constraint; stop as close to the time constraint as possible; return the best possible value; and don't waste CPU cycles polling the timer.

Although there are several different ways of achieving effective timing constraints, most of the ones theorized involved spawning threads. The pattern would be for the main thread to spawn a worker thread to perform all the execution. When the time is approaching the end, the main thread sends a signal to cancel the worker thread, and a value is returned. Of all the patterns found, Java's executor service was found to be the best. This is due to its thread safety, its ability to shut down immediately when the command is given, and its efficiency in not wasting CPU cycles. The most time-safe method would be for the system to poll a value from the thread's class variable after the thread has stopped executing. This class variable would store the current best move as found by the minimax algorithm. The thread would have to keep updating this class variable to ensure that this value is actually corresponds to the best move. The most natural approach here would be to use iterative deepening. At the end of each iteration the value would be updated.

An alternative method that was not employed would be to cancel the thread early, saving the game tree. Then, the system would use the remaining time to back-propagate the minimax values of the tree in order to find the most optimal move. If the algorithm proceeds to the deepest full level, this method will save execution time since the algorithm would not have to employ iterative deepening, so the back-propagation would only have to occur once.

Due to the branching factor, the final level reached would take the longest to compute. If we use either method described before, much of this deepest level, and therefore much of the computation time, would be wasted. This is why an alternative approach was considered where even the leaves of the unbalanced tree could be back-propagated. The risk of using an unbalanced tree would be the skewing of parent values. If a parent only has one child, then the parent will take this value. However if the parent is full, or close to full, then the probability of the back-propagated value being correct, or close to correct, increases. It also increases for a given node if the branching factor is high enough, and the node is one or two generations up the tree before encountering an imbalance. Since

all levels except the last are expected to be filled, we only require that the leaves have a full set of siblings in order for their values to back-propagate. Although, the imbalance may skew the results slightly, it is not expected to have a major impact, especially since the values of nodes are estimates themselves. In either case, there is still the risk of execution time variation in the variable back-propagation. This variation is expected to be rather minimal though, especially if we include unbalanced trees.

3.2 MiniMax

As was stated in Section **??**, a vast majority of the algorithms chose minimax or an optimized version of it. The adversarial nature of the algorithm lends itself well to minimax. Given the wealth of research behind this algorithm, minimax was the obvious choice for the base of the AI player. In order to implement minimax, a Node class was created to store the value of the node, the move the node corresponds to, and the parent and children of the node. The minimax itself works like any other minimax algorithm. The game tree is walked in a breadth first manner, the values of the leaves are estimated using an evaluation function (discussed in Section **??**), and the minimum or maximum values are back propagated according to whether the parent node is a max or a min node. In order to increase the depth of the search, several optimizations were made.

3.2.1 Optimizations

Alpha-Beta pruning :

3.3 Evaluation Function

3.4 Alternate Approaches and Issues Encountered

4 Testing

5 Results

6 Conclusions