# Intro to AI End of Term Assignment

Andrew Hynes, Samuel Sleight, Helen Cheung, Amar Saggu

2013-12-13

## Contents

## 1 Overview of the Program

The basic game structure is that the game is represented in an integer array. Positions 0 - 5 represent the top row of houses, position 6 represents the first player's store. Positions 7-12 represent the bottom row of houses, and position 13 represents the second player's store.

We had both of our AI extend a class called AIBase, which get passed into the playGame method when they play the game. The AI individually have a method called makeMove, which will implement their algorithm and will call the "sow" method of the KalahGame class, which will in turn modify the state of the board, ready for the next AI's move.

After constructing the infrastructure of the game and whatnot, including the base AI class and all of the plumbing for making sure everything connects properly, we went off into our subgroups and made out individual AI which extend the abstract AIBase class. Needless to say, the methods (such as playGame) take an object of type AIBase, which meant we could do all of the game together before we'd made our AI, utilising abstract classes and methods to denote what each AI should have, such as a win/lose method, which each learning algorithm could take and learn from.

We both did have similar ideas in regards to usage of algorithms, and we shared resources found, and ideas about our AI - and there were some cases where methods were useful for both parties, so were put outside the individual AI class, and shared the method for usage in both of our AI, to save time and keep relative concurrence as far as how the AI works is concerned. Despite creating both of AI separately, sharing ideas wasn't something we shied away from. We did share the idea, for example, of extending something like min-max with learning, to make up for the part you cannot search down, and combining the best of both paradigms of AI.

We used GitHub as version control. The repository can be found here - `https://github.com/YeyaSwizaw/kalah`. We found this was the easiest way to collaborate as a group to create coherent code that meshed well with each other, even despite it not being coded all by one person. This is a lot more efficient and sensible than simply having a dropbox folder, we found, as it's less fiddling around. This made it extremely easy to download the repository from any machine, and push changes from anywhere, without any re-downloading of single files or one zip.

The results of our 1000 games were -

AI 1 - Helen & Amar's - 989/1000 wins

AI 2 - Sam & Andrew's - 10/1000 wins

There was 1 draw.

## 2 ROCK Algorithm - By Helen and Amar

### 2.1 Basis

The basis of this algorithm is the minimax algorithm, which is a staple AI algorithm for logic games. The minimax algorithm is a popular algorithm that is used in two-player games such as chess and tic-tac-toe because you can see all of the possible options that you or your opponent can make. (Otherwise known as 'full-information' games.) This helps to represent the game as a search tree of sorts, with the levels of the tree alternating between player 1's and player 2's available moves. The aim of the algorithm is to 'maximise' your options whilst 'minimising' the opponent's, in order to (hopefully) find a winning solution.

The algorithm is fairly simple to implement, but it isn't the best algorithm out there. (I referred to the lecture notes only in AdversarialSearch.pdf, but it is well-documented in literature normally.) ]

### 2.2 Learning

(As explained to me earlier, may need tweaking.) In every game, a search tree of possible moves up to a certain depth is created, and the AI stores the path that it takes. Every possible state has a value attached to it, and this value is the heuristic (I think?) that determines how likely the AI will choose that move. At the end of the game, the AI adds a value to the path taken depending on the outcome of the game. If the path was a winning solution, a positive value is added to all of the states in the search tree of the next game, so that it is more likely to choose said states. (And vice versa for a losing solution.) If a number of states have the same highest-weighted values, then one state out of those is randomly picked to be the next move. We thought it wasn't in our best interests to create another search tree from those states (minimax-ing the minimax, woahhh), so it saves time.

### 2.3 Efficiency

Alpha-beta pruning is utilised (Amar said he would implement it, did he?) to help make the algorithm slightly more efficient. Using the heuristic of comparing values of each state (again, not sure on actual heuristic), on the AI's turn, if one state's value is lower than a certain value (or was it compared against the other states?) then that branch of the tree is terminated as it is unlikely to provide a good outcome. (And likewise for the opponent's moves.) The algorithm also takes a value for the max depth of the tree to be searched.

The higher this value is, the further ahead the AI can search, but also the longer it takes to decide on its next move. Since the time complexity of a search tree is dependent on the depth of the tree searched, a suitable value is needed to balance efficiency and effectiveness. We found that searching up to five levels returned results in a reasonable amount of time.

## 2.4 Testing

We did various tests to see how the AI would perform. With a random AI, the win/loss ratio was mostly evenly split because the chance of any of the opponent's moves being optimal is very low as it is random. This makes it very difficult for our AI to learn from. Next we tested it against the normal minimax algorithm. Again, the win/loss ratio was fairly even, but sometimes the AI would adapt so well that it won nearly every game. (Why did this even happen.) (Next test(s): Combined minimax/learning vs minimax and combined minimax/learning vs learning)

## 2.5 Expectations

We expect the AI to win more games than the opponent (KD + 1) but have the win/loss ratio be fairly even. This is because the AI tries to use the winning moves of both players against the opponent to keep a lead. When no information is present, the AI will choose its moves randomly.

## 2.6 Analysis

Anal-y-sis

# 3 MASH Algorithm - By Sam and Andrew

This is the algorithm and AI constructed by Sam and Andrew, which can be found in MASH.java.

## 3.1 Basis of the Algorithm

We based our algorithm largely on the M&N algorithm - an improvement on the mini-max algorithm. We chose this as it has been greatly successful in the past, and an AI written in Lisp utilising this algorithm has won tournaments with other AI based on other algorithms before. In short, the M&N algorithm has been found to perform significantly better than a base mini-max algorithm.

We found a PDF on the M&N algorithm here - `http://dl.acm.org/citation.cfm?id=362054` and though it was originally written in Common Lisp, we took the ideas of the M&N algorithm, namely that a min-max algorithm should pick from a few options and take into account relative uncertainty (especially considering the fact that algorithms for this task are designed to learn) - therefore we can't be certain as to whether the opposing AI will modify their moves using what they've learnt (potentially from how our AI plays) from the last game(s).

We also took some inspiration from Artificial Intelligence: A Modern Approach, for example, pages 480 - 483, and applied its comments on reasoning under uncertainty to our implementation of the M&N algorithm. We felt it would be prudent, when against any decent learning algorithm, to consider uncertainty when we are unsure, indeed, what move the opposing AI will choose, and whether they will have adapted their efforts from last time. The book proved useful a great deal for referencing in regards to how to construct a sensible AI, and gave us some places to start with algorithms and design. The textbook (and lecture's) comments on probability inspired the probabilistic learning section of our algorithm a great deal, too.

## 3.2   Design

We originally designed a naive base learning algorithm that was based on probability and weighted probability depending on wins/losses. We opted to design this first and then give the algorithm a basis from where to start. In our case, we designed the decision and learning first, via the makeMove method, then fleshed out the search, which was the base our algorithm was going to learn from. Our algorithm was designed with previous games in mind, and we created a HashMap with the "memory" of the game so far, which mapped the GameState with an array of the probabilities based on the results of the last game. Needless to say, the results will be weighted based on how that probability performed, as will be mentioned below.

### 3.2.1   Probabilistic Learning

We generated a probability array (represented in a private class ProbArray) based on the probability distribution of the possible moves that can be made. Before the search algorithm and any learning has weighted these distributions, they start at an integer that adds up to 100. We originally experimented with using doubles, which added up to 1, though errors in calculations with numbers represented in floating point form meant we had to

change to using integers instead for a more precise and sensibly calculated program. This system, however, meant we could weight certain probabilities, and choose how much to weight the AI's choices based on its learning - it'd get a much higher probability if the move has worked in the past, and a much lower probability if the move has resulted in a loss in the past. This means we can also weight heavily based on the results of our min-max search.

Based on the results of past games, and depending on the result, the probability of certain states will be increased, based on an int defined at the top of the class, PROB DELTA. We can (and did) fiddle with the number a bit to try and perfect the amount of learning our algorithm took from a certain move. It'd be foolish to make it learn too much - as the algorithm would favour things that have worked in the past even if they mightn't work in this situation, likewise with too little, as you don't want the algorithm not learning enough from the results of the previous games. We ran the AI against itself a few times, and based the effectiveness on how often player 1 won proportional to player 2 - as since Kalah is a biased game, as the AI learns, player 1 will win more often.

### 3.2.2 Adversarial Search

As mentioned above, in the Basis of the Algorithm section, the algorithm we mainly looked at was the M&N algorithm, which is an extension of mini-max. We generated a search tree - utilising pruning to keep the algorithm running in an amount of time that's manageable. We used the mini-max algorithm that, of course, modified by the introduction of probability, and the very act of learning from past games. Needless to say, the search was just a place for the algorithm to begin to learn from, and we could have picked an algorithm that wasn't an adversarial search, nor took into account the opponent's moves at all, which would be completely doable for a search algorithm in this case, since it's paired with a learning algorithm. However, this wouldn't be anywhere near as effective as starting with a strong adversarial search algorithm and utilising probability and learning to enhance this base.

Our program creates a tree based on the potential outcomes of each move, and assigns a value to each. Since a full search of every possible state is quite obviously not feasible, we search a limited amount, to a capped amount of 4 levels, whereby we use the heuristic of the amount of stones in our pit subtracted by the stones in their pit, where the highest number is the optimal state [that we can see without searching deeper]. Naturally, we can run these states by our previously generated probabilistic learning, and enhance this heuristic by our learning and the element of probability, which

can, in turn, create a further level of stochastic behaviour that the opposing AI mightn't expect - and its learning can be slightly quelled by utilising randomness. We originally attempted to implement an algorithm we thought was similar to min-max, (which turned out to be similar to negamax), but switched to an established algorithm, and converted the pseudocode we found on `http://chessprogramming.wikispaces.com/Minimax` to our algorithm. The website itself proved to be very useful when researching algorithms and choosing one to use.

### 3.2.3 Learning with Search

As the assignment was to make a learning algorithm, we naturally did attempt to make the AI learn based on incorrect moves in the past on top of a min-max search base. We found it important to make sure that though the algorithm does learn, it learns from an established point of rational behaviour. Starting completely naively is, naturally, worse than starting with an established base, and learning from that base can create an AI that utilises two strong ways of beating the task at hand.

Search can only take a program so far in a certain amount of time. Reasonable amounts of time restricts simply searching every possible move ever - something which likely wouldn't ever complete in some games, such as Go, and would take an extraordinarily long time, probably not able to compute fully, in games with smaller potential states, such as Kalah. Alpha-Beta pruning can help, but it won't help your algorithm search much further - even if it does help the speed a bit. We found learning was a perfect place to go where our search leaves us - and though an algorithm based on learning alone generally won't beat (from something other than dumb luck) a well-made search based AI, not at least for a very very large amount of games, an algorithm with search that also takes into account what it has learned can generally trump one that doesn't, but performs similarly in terms of search. Our learning wasn't perfectly implemented, but we felt like it was more than good enough for this particular task, especially considering it was paired with a min-max.

## 3.3 Analysis of Behaviour

### 3.3.1 Expectations

We expected our algorithm to perform quite well (and at least equally) throughout the 1000 games. We expected the learning we utilised to not gain a giant lead from the other AI, rather, to mainly 'keep up with' the

opposing team's efforts of learning from our AI. Rather than having a huge boost in improvement as time went on, we expected a slight boost, but that would also be counteracted by the fact the opposing AI was also learning. We expected this from pairing our learning algorithm with a tried and tested adversarial search algorithm.

We expected our AI's lead (if one existed) to stay relatively constant as time went on, and any growth or reduction in performance to be slight. Our algorithm didn't start out entirely naively and learn rapidly - it utilised search as well as learning to get a nice foothold immediately. Needless to say, we were playing against another very very strong and well built AI, so we weren't expecting to completely clean the floor with it whatsoever, like we might expect when versing pure randomness or versing a human.

### 3.3.2   Performance

Our algorithm performed a great deal worse than expected, though this was mostly due to the fact that the other team's AI was pretty much as good as ours was, as we should have probably expected. Creating an algorithm that utilises learning on top of search is no easy feat, however, and our algorithm did not perform poorly, by any means, considering the AI we went against. An AI that stomps random chance does not necessarily walk over AI that do the same to random chance. Overall, considering the time-frame we had and the fact that we were (unfortunately) restricted to Java, the performance was a lot worse than we expected, but the algorithm itself was actually good against random/naive AI - it wasn't that our AI was bad, per se, just theirs was very, very good.

Our AI was relatively overshadowed by their AI, but we found a great deal about using AI in practice from the exercise, and the performance of this wasn't quite so surprising when comparing implementations, as their implementation was a great deal better than ours - despite our ideas being relatively similar. Despite this poor performance, however, we learnt a lot about AI, and how potent an AI can be when it utilises multiple paradigms in tandem to cover the others' weaknesses.