# Intro to AI End of Term Assignment

Andrew Hynes, Samuel Sleight, Helen Cheung, Amar Saggu

2013-12-13

## Contents

**1 Overview of the Program** **1**

**2 ROCK Algorithm - By Helen and Amar** **3**
2.1 Basis . . . 3
2.2 Learning Theory . . . 3
2.3 Efficiency . . . 4
2.4 Analysis (via Testing) . . . 5
2.5 Expectations . . . 5
2.6 Analysis . . . 6

**3 MASH Algorithm - By Sam and Andrew** **6**
3.1 Basis of the Algorithm . . . 6
3.2 Design . . . 7
3.2.1 Probabilistic Learning . . . 7
3.2.2 Adversarial Search . . . 8
3.2.3 Learning with Search . . . 9
3.3 Analysis of Behaviour . . . 9
3.3.1 Expectations . . . 9
3.3.2 Performance . . . 10

## 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 "move" method of the KalahGame class, which will in turn modify the state of the board (using the "sow" private method), 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 in essence, however, 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 - an idea based on having learning filling in the gaps where search cannot reach due to very high complexity. Shared ideas such as this one were generally shared between us, and we talked about this a lot whilst we didn't have lectures, and had a Facebook group for when we wanted to share ideas or communicate things online.

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 - 35522 seeds acquired

AI 2 - Sam & Andrew's - 12478 seeds acquired

# 2 ROCK Algorithm - By Helen and Amar

## 2.1 Basis

The basis of this agent 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 (or "full-information" games) because you can see all of the possible options that you or your opponent can make. 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. To improve the basic minimax algorithm, a method of learning is applied and combined with minimax to make the AI smarter by learning winning/losing strategies in looped game sequences. The learning method is detailed in the next section. The minimax algorithm is fairly simple to implement and quite effective, but it isn't the best algorithm out there. I referred to the lecture notes in AdversarialSearch.pdf on Canvas for the base algorithm. We looked at and modified the implementation slightly from `http://chessprogramming.wikispaces.com/Minimax`.

## 2.2 Learning Theory

Not sure if there is a known name for this algorithm, or if there is anything similar to it, but since we managed to come up with it ourselves, here is a detailed explanation on how it works. The agent condenses all of its and its opponent's past moves into a tree. The root of the tree represents the initial state of the Kalah board. From there, its children contain actions in the form of move sequences. The tree doesn't store the states of the board, just the actions from one state to the next. Keeping the state of the board as well isn't really necessary, since it can be generated via the KalahGame.getState methods. Each node also stores weights that are associated with each move. These weights score how successful that particular node has been in the past. If a node already exists in the tree, then only its weight needs to be modified. As an example of how this works in practice, say that at a conclusion of a certain game, Player 1 wins. Then all the moves that Player 1 took are added to the tree, and their weights are increased by 1. Since Player 2 lost, all of Player 2's moves are also added to the tree, but this time their weights are decreased by 1 instead. NOTE: When move sequences are added to the tree, sometimes the moves have to be converted from Player 2 to Player 1, as the tree always assumes that Player 1 starts. We ignore the case when

the agent and its opponent draw. From a draw, we cannot determine if the move sequence that either player took was optimal. Therefore no changes are made to the tree. After a couple of games, this tree starts to give the agent a good indication of what move it should take, based on past experiences. Nodes with high scores indicate a good path for the agent to take, while nodes with negative scores should be avoided. When the agent is running, it traverses down the tree according to the moves the opponent has made. Especially at the start, these moves are not actually in the tree. When we have no knowledge of how effective a move has been in the past, we just use minimax to find the next move. NOTE: There is also a flag to disable minimax in our algorithm, and use random moves instead. If however the agent does find a node for its current move, it knows that the path has been previously traversed. It can then check the weight of that node's children to determine its next move. If there is a child that has a non-negative weight, the AI will follow it down by performing the action specified by that node. In the case that there are multiple children, it picks the node with the highest weight.

### 2.3 Efficiency

When implementing minimax, alpha-beta pruning is utilised to help make the algorithm slightly more efficient. Using the method of comparing values of each state, on the agent's turn, if one state's value is lower than a certain value then that branch of the tree is terminated as it is unlikely to provide a good outcome. (And likewise for the opponent's moves.) We considered this but due to time constraints, we could not implement it in the end. 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 agent 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. In theory, the minimax algorithm will give you a perfect sequence of move for a game. In practice however, it isn't that simple. For games such as draughts and Kalah, generating every single possible game isn't practical. Instead, we had to limit the depth to just 4 turns. The algorithm actually ran in reasonable time even with a depth of 7, but for some reason a depth of 4 always seemed to win against every depth below 8. We aren't actually why this actually is, but we know that the other team also encountered this. This suggests that perhaps it is something to do with our heuristic (since we all used the same heuristic), rather than our implementation.

### 2.4 Analysis (via Testing)

We ran various tests to see how our agent would perform. To test, we set our agent to be player 1 and some other AI to be player 2, ran 1000 games between the two and returned the number of wins, losses and draws, as well as the final state of the game board in every game. With a random AI, the win/loss ratio was mostly evenly split at about a ~51% win rate 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 agent to learn from. Next we tested it against the normal minimax algorithm. Again, the win/loss ratio was fairly even at ~51%, but on rare occasions the agent would adapt so well that it won nearly every game and had a ~98% win rate. After combining minimax with our learning method, we ran it against just minimax. The result was an even 50% win ratio. This was expected since playing against minimax essentially reduces our combined AI into one as well, and the winner of the match was determined by who played first. Finally we tried running combined minimax/learning against just the learning algorithm. Again the win/loss ratio was ~50% but the learning on its own would win a few games more than the combined AI. This somewhat makes sense that the outcome is even since at the heart of it, the combined AI is minimax and from that, both algorithms would learn from it, thus staying fairly equal to each other.

### 2.5 Expectations

We expected our agent to win more games than the opponent 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, for example at the start of the first game, the agent will choose its moves randomly, but for the next game onwards, it starts to utilise learning in order to gain a lead. Since we know that the other agent will also utilise the minimax algorithm, we should expect to have a fairly even win/loss ratio. This is mainly because of the fact that minimax algorithms tend to produce a small number of different paths, compared to say a random algorithm. This implies that our learning algorithm will quickly pick up on the optimal paths it takes, and use those paths against it. Our tests have shown that when our learning algorithm on its own plays against a minimax algorithm, the learning algorithm can actually outperform the minimax algorithm. We think that was because it chose random moves when it hadn't previously explored the current path. Without this randomness though, we think that the outcome of making the two agent's play each other

repeatedly should be a 50% win/loss ratio, because of the use of minimax in both agent 's.

### 2.6 Analysis

Our agent performed far better than originally expected, with an average win rate of ~99%. Whilst our agent was minimax with our own devised learning algorithm, the other team's agent was also minimax but with probabilistic learning instead. Since we expected our agent to be just slightly better than average against minimax, this startling disparity indicates that the implementation of the other team's agent (whether it was the base algorithm or the learning itself) was not correct at some points. Since learning alone wouldn't give such a one-sided result, I think that the minimax implementation on the other team's side failed to match up against our minimax implementation. If they had been equal, then the difference between our scores may have been much smaller.

## 3 MASH Algorithm - By Sam and Andrew

This is the AI constructed by Sam and Andrew, and 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. The M&N algorithm has been found to perform significantly better than a mini-max algorithm on its own.

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). The idea of adding something on top of a mini-max algorithm is exactly what we wanted.

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

As mentioned in the starting section, our vision was an adversarial search algorithm that could be improved by learning, which would fill in the parts it would be infeasible for the AI to search. We implemented this first with 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. The results were 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. 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.

### 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 using the mini-max algorithm that was modified by the introduction of probability - where the probability was modified by the 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, and propagate the values up the tree. We ran these states by our previously generated probabilistic learning, and enhance our 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/adapted the pseudocode we found on `http://chessprogramming.wikispaces.com/Minimax` to our algorithm's needs. 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 still take an extraordinarily long time 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 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, theirs was just 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.