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 Something Algorithm - By Helen and Amar **2**

3 MASH Algorithm - By Sam and Andrew **2**
3.1 Basis of the Algorithm 2
3.2 Design 3
3.2.1 Probabilistic Learning 3
3.2.2 Adversarial Search 4
3.3 Analysis of Behaviour 4
3.3.1 Expectations 4
3.3.2 Performance 5

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 algoririthm 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 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 - X/1000 wins

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

2 Something Algorithm - By Helen and Amar

Put stuff here!

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 improement 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 probibilities 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 a state such that the sum of the probabilities is 1. Any change to any probability goes through the updateProbability method, which calculates the other probabilities in a way that they remain at 1, even if one of them is increased by an amount.

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, $\text{PROB}_{\text{DELTA}}$. 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 5 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.

3.3 Analysis of Behaviour

3.3.1 Expectations

We expected our algorithm to perform quite well 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.

3.3.2 Performance

Our algorithm performed