Artificial Intelligence Techniques in HUS

Ossama Ahmed

April 9, 2016

Abstract

Reinforcement learning techniques have been employed to solve many games that involve a high branching factor. This paper studies the implementation of the TD-learning algorithm with the min-max search algorithm to develop an AI agent to play HUS board game which is from the family of "Mancala" games.

1 Motivation

There are a lot of techniques that can be used when it comes to developing an Artificial Intelligence agent that plays a deterministic observable board game like "HUS". The most common techniques used are game tree search, supervised learning and unsupervised learning. Game tree search is a family of algorithms where a game tree is searched. The nodes of the tree represent the game/board state and each node's children represents the resulting game/board state when a certain move/action is performed. Supervised learning techniques are techniques that have a predetermined classification which can be conceived of as a finite set, previously reached by humans. Such techniques can also be defined as the process of searching for algorithms that reason from externally supplied instances to produce general hypotheses which then can make predictions about future events. On the other hand, unsupervised learning techniques are not provided with classifications. The basic concept of unsupervised learning is to develop classification labels automatically or it can be defined as the attempt to uncover hidden regularities or similarities or detect anomalies in the data[1].

Moreover, since this AI agent was developed by a non-expert in "HUS", it was hard to use supervised learning techniques. In effect, the algorithm has to be provided with the input, the output and the expected output. In the case of training the agent by a non-expert human, the expected output would involve an error; thus the agent would base its own decisions on erroneous information.

Finally, I decided to use a simple MIN-MAX algorithm that uses state searching to find the best possible move using a linear evaluation function. Afterwards, reinforcement learning techniques were used to learn the weights included in the evaluation function. In supervised learning an agent is taught how to respond to a given situation while, in reinforcement learning, the agent is not, but rather it has a "free choice" in how to behave. However once it has taken its actions it is then told if its actions were good or bad (this is called the reward – normally

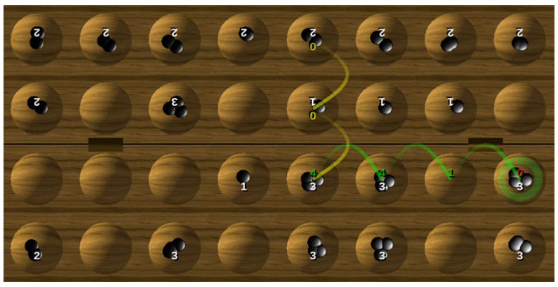
Figure 1: This is the same HUS board game discussed but with less pits.

a positive reward indicates good behavior and a negative reward bad behavior) and, from this, it has to learn how to behave in the future[3].

2 Technical Approach

I implemented a MIN-MAX algorithm with alpha-beta pruning at the beginning because it is reported that MIN-MAX algorithm is the most effective algorithm for creating an artificial agent for board games. It works by creating a secondary copies of the present game during play and exhaustively tested moves on these copies to get the outcome, and when the desired outcome is found: it retrieve the move that is responsible for this outcome and send it to the present game[2]. For implementing the MIN-MAX algorithm, a choice had to be made for the evaluation function to use and the depth of the tree.

2.1 Evaluation Function

After reading previous research papers done on similar board games and testing different heuristics/features. I have decided on using 10 different heuristics based on several trials.

1. H1 : Agent's number of seeds.
2. H2: Opponent's number of seeds.
3. H3: Agent's number of pits that contains 0 seeds.
4. H4: Opponent's number of pits that contains 0 seeds.
5. H5: Agent's number of pits that contains 1 seed.
6. H6: Opponent's number of pits that contains 1 seed.
7. H7: Agent's number of pits in the front row of it's home zone that contains more than one seed.
8. H8: Agent's number of seeds in the front row of it's home zone that are movable (in a pit that contains more than one seed).
9. H9: Opponent's number of pits in the front row of it's home zone that contains more than one seed.

10. H10: Opponent's number of seeds in the front row of it's home zone that are movable (in a pit that contains more than one seed).

The utility function is used to evaluate the board state if the maximum depth was reached, or with the alpha-beta pruning implementation, the move didn't seem promising enough to explore the node (state) deeper. I decided to go with a simple utility function where the board state is evaluated and given a specific score based on the heuristics and weights corresponding to each heuristic, then the arctan function is applied on the score*0.5 to cap the states utility values between 1 and -1 and to have a wider range of results. The value 0.5 was chosen using trial and error testing.

Utility(S)= $\tan^{-1}(0.5 * \sum_{i=1}^{i=10}(W_i * H_i))$

S denotes the board state

W denotes the weight applied to the corresponding H (heuristic).

The reason I caped the utility function values between 1 and -1 is that it is hard to determine one score that represents a winning board or a losing board when the game is over. If game over is reached during the game tree search, the state/node is given a score of 1 if the agent won, -1 if the agent lost and 0 if there is a draw.

All the weights that corresponds to H1, H3, H5,H7 and H8 were initialized to 1, and all the weights that corresponds to H2, H4, H6, H9 and H10 were initialized to -1 since an increase in these features indicates a losing game for the agent. The nearly optimal weights for these heuristics were deduced from using Reinforcement learning which I will discuss later.

2.2 The Game Tree Depth

In a perfect scenario the agent would search the whole tree till the game ends in each leaf in the tree, but since the agent is constrained by the memory available to it and the time spent on each move, which was 2 seconds for this implementation, it can't develop and search the entire game tree. Without implementing the alpha beta pruning, the maximum depth achieved was 5 levels.

After implementing alpha beta pruning, the agent was able to go 1 more level but it would time out on some moves since the branching factor differs on each node in the tree, not on in the root only. After testing and playing the agent against itself for more than 500 games, we found that on average, it's better for the agent to search deeper and return it's best current move when the 2 seconds pass.

Furthermore, it was found after testing that the agent could make a decision on the depth that it should use based on the initial branching factor of the root. The following depths corresponding to the branching factors were proved to perform better in general.

Branching Factor	Depth
b >18	6
19 >b >8	7
9 >b >5	8
6 >b >0	6

The reason I chose to explore 6 levels only when there is less than 6 branches is that branching factor can grow later in the tree. If the root has 6 legal moves only, then this move is considered a "critical" move. Therefore, the agent needs to explore the full tree and it has to avoid to time out and return the best current move.

2.3 Moves Order In The Game Tree

The effectiveness of alpha-beta pruning is highly dependent on the order in which the states are examined[4]. Therefore, ordering and sorting the states according to the most optimistic states should result in a better performance. I found that ordering the moves according to the evaluation function is effective and leads to a better performance. However, sorting the moves at each node was found to be an expensive operation that results in a slower exploration rate. Taking this into consideration, I decided to only order the moves at the root and then later on choose moves randomly.

2.4 Learning the Weights using Temporal Difference

Learning in this system is a combination of the back-propagation method of training neural networks and TD(λ) algorithm from reinforcement learning. We use the second to approximate our game theoretic function, and the first to tell the neural network how to improve. If we had a game-theoretic function then we could use it to train our neural network using back-propagation, however as we don't we need to have another function to tell it what to approximate. We get this function by using the TD(λ) algorithm which is a methods of solving the temporal credit assignment problem (that is when you are rewarded only after a series of decisions how do you decide which decisions were good and which were bad). Combining these two methods gives us the following formula for calculating the change in weights,[3]

$$\Delta W_t = \alpha(Y_{t+1} - Y_t) * \bigtriangledown_W Y_k$$

t is time (in our case move number).
Y_t is the evaluation of the board at time t.
α is the learning rate.
$\bigtriangledown_W Y_k$ is the partial derivative of the weights with respect to the output.

3 Conclusion

3.1 Advantages

The evaluation function is fast to calculate, and the agent learned nearly-optimal weights using reinforcement learning; the agent played around 8000 games against itself with a 30% chance of returning a random move. Also, changing the depth according to the branching factor had a large impact on the performance because the agent was able to make stronger/ smarter moves later on during the game when the branching factor decreases.

3.2 Disadvantages

One of the disadvantages of the agent that it doesn't account for any surprising moves made by the opponent, it always assumes that the opponent is smart and will make the best move available, which might enable the opponent to trick the agent into losing. In addition to this, it doesn't cache any moves that proved to be killer moves or winning moves based on the past games played, so the tree is constructed from scratch every turn even though some of the states were encountered in nearly every game and the optimal move for these states were discovered before.

4 Further Improvements

In my opinion, there are three improvements that would be beneficial:

The first one is to train the agent using the Q-learning algorithm, that way the agent would be able to make faster decisions based on the games played in the past.

The second one is to try to come up with better heuristics and test them with more games and implement more than one neural network layer to make use of the sigmoid function and the advantages of deep learning in general.

The third one is to explore the moves in a different order according to the learned moves from the previous games played and its values. I believe this can affect the agent's performance tremendously since, as I mentioned, the alpha beta pruning implementation might prune optimistic moves just because of the order. Fixing this would ensure better and more accurate pruning.

References

[1] Randle Oluwarotimi Abayomi, Olugbara O.Oludayo, and Lall Manosh. [*An Overview of Supervised Machine Learning Techniques in Evolving Mancala Game Player*].

[2] Noraziah ChePa, Asmidah Alwi, Aniza Mohamed Din, and Muhammad Safwan1: THE APPLICATION OF NEURAL NETWORKS AND MIN-MAX ALGORITHM IN DIGITAL CONGKAK, `http://www.icoci.cms.net.my/proceedings/2013/PDF/PID127.pdf`

[3] Imran Ghory: Reinforcement learning in board games, `https://www.cs.bris.ac.uk/Publications/Papers/2000100.pdf`

[4] Stuart Russell and Peter Norvig: Artificial Intelligence, A Modern Approach.