Dots-and-Boxes AI Algorithm

Rowland O'Flaherty and Jay Wu

I. Problem

Games are appealing to AI researchers because they give a set of precise rules, have a discrete set of states and actions, and are easy to implement on a computer. Moreover, games still are interesting problems to solve because of their large set of states, which means their solutions can be applied to a wide variety of application to address many other problems that are seen in society. For example, the work on creating the best computer chess player lead to many new tree search techniques in AI research.

For this project we will be looking at the dots-and-boxes game. The setup of this game is a grid of $n \times n$ dots, where $n \geq 2$. Each player takes turns connecting two adjacent dots using a either a vertical or horizontal line with the goal of connecting four dots to enclose a box. If a player encloses a box they get a point and get to take another turn. The player with the most points after all the dots have been connected wins the game. The dots-and-boxes game is a deterministic, fully observable, perfect information, zero-sum competitive game.

In our work we implemented different tree search algorithms to compare each algorithms effectiveness at playing the game of dots-and-boxes. The algorithms that we compare are the minimax search, minimax with alpha-beta pruning, and Monte Carlo search. As a result of creating these algorithms we have created an computer opponent for the dots-and-boxes game that can play at an advanced level in real time.

II. Related Work

Extensive research has been done in AI on how to make computer opponents for the dots-and-boxes game. Elwyn Berlekamp of UC Berkely, wrote a book on the game in 1994 [1]. Berlekamp was interested in the dots-and-boxes game from a combinatorial game theory perspective and the mathematics associated with it. One method to solve this game and other games is with an AI tree search algorithm. Campbell and Marsland's paper gives a comparison of minimax tree search algorithms [2]. Reinefeld and Ridinger proposed an efficient state space search using recursion [3]. A deficiency in Reinefeld and Ridinger's proposed algorithm is that it can only be used in uniform trees unless some sort of hybrid variant is used with a traditional search algorithm such as Alpha-Beta. More recent work by Chaslot, Windands, Van den Herik, and Uiterwijk used Monte-Carlo algorithms for tree searching [4]. However, for their algorithm to work well it needs some previous knowledge of the tree structure.

III. Approach

Before the creation of any AI algorithms could take place an application of the dots-and-boxes game was needed. Thus, a graphical user interface of the dots-and-boxes game was first created using Matlab. In this application the game boards can range from 3x3 grid of dots to 10x10 grid of dots. In addition, the application allows for selection of human vs. human play, human vs. computer play, and computer vs. computer play. The computer vs. computer mode is used to test the performance of the various algorithms against one another. In addition, the human can select which computer algorithm to use as his computer opponent.

Similar to many AI algorithms for games a tree search algorithm is used to determine the best move to take for the computer opponent [5], [6]. Given that the number of different states ($|S| = 2^{2n(n-1)}$) and number of different branches ($|B| = 2n(n-1)!$) for the dots-and-boxes game the tree search can be quite large, even with a relatively small n. For instance, a grid of 5x5 dots has more than one trillion different states. An efficient tree search algorithm is necessary to allow the computer opponent to be a worthy adversary.

For this project five different AI algorithms (Random, Greedy, Minimax, AlphaBeta, and Monte Carlo) were implemented and each was limited to one second of computation time to select its move. Each of these algorithms are explained in the following subsection III-A-III-E. The effectiveness of any tree search algorithm has a large part to do with the utility function (also known as the heuristic function) used to assign a value of each node in the tree. A custom utility function was created for the dots-and-boxes game for this project. This is explained in detail in subsection III-F.

A. Random

The Random algorithm is very simple; the computer chooses a move at random with a uniform probability. This algorithm is not meant to be a genuine computer opponent, but more used as sanity check that the application is working correctly. Also, it is expected that every other computer AI algorithm should be able to beat this algorithm; and they all do.

B. Greedy

The Greedy algorithm picks a move by analyzing the value of every possible move it has to choose from and selects the one with the highest value, thus the name of the algorithm. This algorithm selects only based on short term reward not long term payoff.

C. Minimax

The Minimax algorithm uses the standard minimax tree search algorithm as outlined in the textbook [7] section 5.2.1. This uses a simple recursive computation of the minimax

values of each child node for a given node. Once the recursion reaches the leaf nodes the minimax values are backed up to the initial calling node. For most of the moves in the dots-and-boxes game it is infeasible to recursively traverse the entire tree all the way to the leaf nodes and thus the recursion is cutoff after a specified search time of one second. Since every branch can not be searched in the tree the branches are chosen at random with a uniform probability. The algorithm selects the move with the maximum minimax value.

D. AlphaBeta

The AlphaBeta algorithm uses the same minimax algorithm as described in section III-C but it increasing the efficiency of the search algorithm by pruning certain branches of the tree. This is described in the textbook [7] section 5.3. This algorithm achieves the same results as minimax without looking at every node by keeping track of the minimum and maximum values that any given subtree can return (known as the *alpha* and *beta* parameters). Knowing these parameters for a given subtree is used to eliminate further search of the subtree if that tree's alpha and beta parameters demonstrate that it can do no better than another subtree that has already been searched. The ordering of the search matters for alpha-beta pruning and affects how efficient the algorithm is at removing subtrees to be searched. For our implementation we did not attempt to optimize the ordering and it was chosen at random with an uniform distribution. The algorithm selects the move with the maximum minimax value.

E. Monte Carlo

The last algorithm that we implemented for this project is know as Monte Carlo tree search. We implemented the same algorithm as described by Chaslot and Winands [4]. This does not attempt to search the entire tree but instead picks branches randomly at each level to search down. Once the algorithm has either reached a maximum search depth or a leaf node the values are backed up through the tree to initial node. Each of the backed up values are averaged with other backed values for that branch. The algorithm selects the move with the maximum averaged value. For implementation we selected maximum search depth of 7 moves. This allowed for around 800 branches to be evaluated in the one second search time limit.

The branch probability distribution does not have to be uniform for this algorithm. For our implantation we examined both a uniform distribution and a distribution discussed by Chaslot and Winarnds known as Crazy's Stone algorithm. This algorithm uses the mean values as well as the variances of each node's backed value to determine the probability of selecting a particular branch for the Monte Carlo search. If the mean values are labeled as u_i and the variances are σ_i^2 and the nodes are ordered $\mu_0 > \mu_1 > \cdots > \mu_N$, then each move is selected with a probability proportional to

$$p_i = exp\left(-2.4\frac{(\mu_0 - \mu_i)}{\sqrt{2(\sigma_0^2 + \sigma_i^2)}}\right) + \epsilon_i. \quad (1)$$

Here ϵ_i is regularization constant. In our implementation we use a slightly different value then used by Chaslot and Winarnds for the regularization constant, which was found to give better exploration verses exploitation tradeoff. We defined ϵ_i as

$$\epsilon_i = 15\frac{0.1 + e^{-i/N}}{N}. \quad (2)$$

F. Utility Function

For each of the algorithms explained above a utility functions is needed to evaluate the current state of the board. If S_{4A} is the number points player A has received, S_{4B} is the number points player B has received, S_3 is the number of squares with 3 lines, and S_2 is the number of squares with 2 lines, then the utility function for a given board state for player A is

$$utility(s) = 2S_{4A} - 2S_{4B} + 0.75S_3 - 0.5S_2. \quad (3)$$

The reasoning behind this utility function is that it obviously good to gain points and not let your opponent gain points, which is demonstrated in the first two terms in (3). Boxes with three lines are good because a point can be gained with the next move, third term in (3). Moreover, boxes with two lines are bad because if you place a line on those boxes then your opponent can score on their next move, fourth term in (3). This utility functions causes the board to have many squares with two sides because it is not beneficial for a player to add the third line to a square. The weighting values in (3) were found empirically.

IV. Evaluation

The evaluation of the algorithms described in section III was done by running each algorithm against the other algorithm for 100 games on four different board sizes: 4x4, 5x5, 6x6, and 7x7. To make sure there was no bias from which player made the first move, going first was split 50-50 between the two players. The results are expressed in the following tables by showing the number of games won by each algorithm for each board size. Tie games account for any columns that do not add up to 100. Some results are not shown here because their results can clearly be inferred from the data that is shown below (e.g. AlphaBeta vs Greedy is not shown because AlphaBeta usually beats Minimax which almost always beats Greedy).

Table 1: Random vs Greedy

Algorithm	4x4	5x5	6x6	7x7
Random	0	0	0	0
Greedy	100	100	100	100

Table 2: Greedy vs Minimax

Algorithm	4x4	5x5	6x6	7x7
Greedy	8	0	0	0
Minimax	92	99	98	99

Table 3: Minimax vs AlphaBeta

Algorithm	4x4	5x5	6x6	7x7
Minimax	45	35	4	2
AlphaBeta	54	63	94	97

Table 4: AlphaBeta vs Monte Carlo (Uniform)

Algorithm	4x4	5x5	6x6	7x7
AlphaBeta	50	33	14	1
Monte Carlo (Uniform)	48	65	80	98

Table 5: AlphaBeta vs Monte Carlo (Crazy's Stone)

Algorithm	4x4	5x5	6x6	7x7
AlphaBeta	49	45	34	3
Monte Carlo (Crazy's Stone)	48	54	64	95

Table 6: Monte Carlo (Uniform) vs (Crazy's Stone)

Algorithm	4x4	5x5	6x6	7x7
Monte Carlo (Uniform)	49	50	48	49
Monte Carlo (Crazy's Stone)	49	48	52	50

One last match up was performed and that was AlphaBeta verse a human (not necessarily an expert) on the 5x5 grid. The AlphaBeta algorithm won 90% of the time.

It also interesting to note the percentage of wins for the player that goes first in the game for all games where the win-loss percentage is approximate 50-50 (i.e. AlphaBeta vs either Monte Carlo variant for 4x4 games and Monte Carlo vs Monte Carlo for all game sizes). It was found that in these games the player that went first won 94% of the time.

V. Discussion

It is not surprising that the Greedy algorithm always beats the Random algorithm (Table 1). This was a good sanity check that our utility function discussed in section III-F was correct. It is also not surprising that the Minimax almost always beats the Greedy algorithm (Table 2). Basically, the Greedy algorithm is exactly the same as the Minimax algorithm if the Minimax is limited to look ahead by only one move. Therefore, since the Minimax looks ahead many more moves than one (usually around 5) it has the clear advantage over the Greedy algorithm.

The results for Minimax verse AlphaBeta (Table 3) demonstrate the power of pruning. In the smaller games the match up is almost equal because Minimax will search most of the tree for the best move. However, in the larger games pruning is much more important because the limited time to search. Minimax will cover a small percentage of the tree in one second making its move almost equivalent to a random guess. With pruning, AlphaBeta can cover a much large portion of the tree giving it a quality move.

The same thing is seen for the matchup between AlphaBeta and both variants of the Monte Carlo algorithm (Table 4 & 5) as is seen for Minimax vs AlphaBeta. The results are almost split for the smaller games but are clearly dominated by Monte Carlo in the larger games. This is believed to because the Monte Carlo algorithm searches to a greater depth (7 moves ahead) than AlphaBeta. The search depth of AlphaBeta might be 7 moves for the smaller games and only 3 moves for the larger games.

We found there to be no significant improvement of using the Crazy's Stone branch selection algorithm over a uniform random selection (Table 6). This is most likely because the means of the values can be accurately estimated with the number of selections that occur with the uniform random selection. With a one second search limit about 800 branches are covered. Even with the largest game of 7x7 this covers each possible first move about 10 times, which seems to be enough to estimate the mean value . Thus, it is not necessary to prioritize branches that are expected to have higher values as is done with the Crazy's Stone algorithm.

Many insights in tree search algorithms have been acquired with this project. The first and foremost is that the effectiveness of the algorithm is directly related to the ability of the utility function to correctly evaluate the board. Not shown in the paper but several other utility functions were examined which made the algorithms almost useless–sometimes not even being able to beat the Random algorithm. If there was more time for this project it would have been interesting to see if the performance of the AlphaBeta could have been improved with a more intelligent ordering scheme, instead of just picking branches at random.

In conclusion, we have demonstrate that it is possible to use a tree search algorithm as a computer AI opponent for the dots-and-boxes game that can competitively play against a human. Moreover, in the process we compared algorithms that were studied in class to algorithms that were found in the literature. The results of the comparison of these algorithms (Random, Greedy, Minimax, AlphaBeta, and Monte Carlo) are presented and analyzed in this paper.

References

[1] Berlekamp, Elwyn. *The Dots-and-Boxes Game: Sophisticated Child's Play*. A.K. Peters, 2nd Edition, 1994.

[2] M. Campbell and T.A. Marsland. *A Comparison of Minimax Tree Search Algorithms* Artificial Intelligence, 20 (1983) 347-367.

[3] A. Reinefeld and P. Ridinger. *Time-efficient state space search*. Artificial Intelligence, 71 (1994) 397-408.

[4] G. Chaslot, M. Windands, H. Van den Herik, and J Uiterwijk. *Progressive Strategies For Monte-Carlo Tree Search*. WSPC (2008).

[5] Y. Tsuruoka, D. Yokoyama, and T. Chikayama. *Game-Tree Search Algorithm Based on Realization Probability*. ICGA Journal (September 2002).

[6] van den Herik, H.Jaap, Uiterwijk, Jos W.H.M., van Rijswijck, Jack. *Games solved: Now and in the future* Artificial Intelligence, 134 (2002) 277-311.

[7] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach* Prentice Hall (2010) 0-13-604259-7.