Fun with Dots and Boxes: An Approach Using Q-Learning and Artificial Neural Networks

Andrew Miller, Nam Phung, Dale Dowling
CS 5751
University of Minnesota, Duluth
1049 University Dr
Duluth, Minnesota 55812

Abstract

The purpose of this report is to explore machine-learning approaches to the simple but surprisingly complex paper-and-pencil game Dots and Boxes. We implemented a variation of previously explored stat-space reduction methods in order to make the game more manageable for a learning agent, and also implemented various reinforcement learning techniques such as TD-Learning, Q learning, and Monte Carlo Tree search, with varying success. Future work with Artificial Neural Networks, further improvements of environment efficiency, and forward-propagation strategies such as minimax search on smaller board sizes are possible avenues for future work/research.

Introduction

Dots and boxes is a fairly well-known paper and pencil game played between two players [Zhuang et al, 2015]. Players take turns drawing lines between adjacent vertical or horizontal dots, which are laid out in a grid of a certain (predetermined) size. The objective of the game is to create "boxes" by drawing the fourth line of a unit square. The player who does this is said to claim that box, and they then must draw another line. The game ends when no more lines can be drawn, and the winner is the player that has claimed more boxes. In figure 1 we see an example situation on a 2×2 board.

When referencing the size of a game of Dots and Boxes, a 4×5 board would indicate that the board size is of 4 boxes (horizontally) by 5 boxes (vertically); the number of dots on a game board of this size is $(n + 1)(m + 1)$, where n and m are the dimensions of the board. The number of edges on a $n \times m$ board is given by the following equation:

$$E(n, m) = nm + (n + 1)(m + 1) - 1.$$

Dots and boxes is a zero sum game, as the consequences of the actions of a participant are equally balanced with the actions of their opponent (with optimal play). It is also an impartial game, as either player could take the same action, assuming an equivalent game state. In game theory, a popular approach for playing a zero sum game is to use a minimax algorithm, minimizing the possible amount of points given up to the opponent while maximizing points gained.

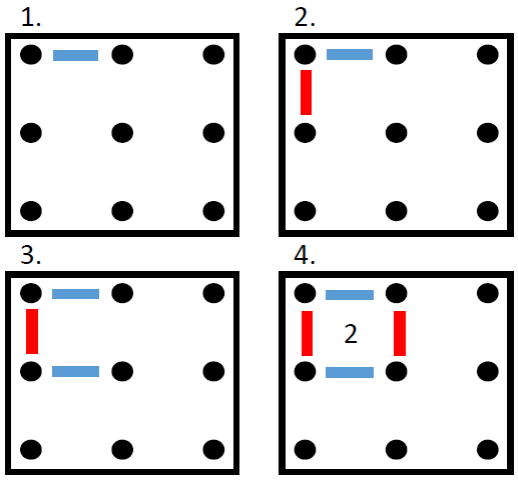

Figure 1: Player 2 draws the last line of a box, and thus claims the box. Player 2 will now be required to draw another line on the same turn.

Related Work

David Wilson, a former professor and programmer from UW-Madison, has done a lot of work with Dots and Boxes, including developing methods for analyzing a game board as the game is played. Although his work has now been topped, he created software that was able to analyze game boards using a ranking system for each possible move. This approach was exhaustive and required a lot of processing power, but it was able to solve small game boards in a reasonable amount of time. Wilson developed a specific methodology for analyzing Dots and Boxes boards which may be useful in our machine learning algorithm [Wilson, 2014]. His methodology is helpful for determining the value of specific moves but it is unfortunately very inefficient to

calculate. Since then, a faster, more precise methodology has been developed, which will be discussed later. Dots and boxes has officially been "solved" (meaning the outcomes of optimal play have been determined) on all boards leading up to and including a 4×5 board [Barker and Korf, 2012]. Under the assumption that both players play optimally, the endgame will be a draw. The main reason larger boards have not been solved owes to the extremely large state space of larger boards. Barker and Korf have outlined ways to reduce this search space but their approach still takes a considerably large amount of processing power, which is why 4×5 is the largest officially solved board [Barker and Korf, 2012].

While there isn't any formal proof, the consensus in the Dots and Boxes community seems to be that the 5×5 board was solved by William Fraser in 2014 [Wilson, 2014]. According to the calculations done by his program (which exhaustively parsed through about 30,000,000,000,000 states over the course of 8 years), there are 8 different confirmed starting moves that player 1 can make to have a guaranteed win under optimal play.

In 2015, a group of researchers from the University of Beijing and the University of Bremen, Germany developed an A.I. called QDab to play Dots and Boxes on a 5×5 board, with the goal of increasing both the efficiency and efficacy of their design as compared to other agents available at the time. Their main strategy utilized Monte-Carlo tree search and an artificial neural network for playing the game. The team also trained their artificial neural network (ANN) by generating partially complete games and training it against a minimax algorithm's findings. Through back propagation, the ANN used its output to optimize its cost function [Zhuang et al, 2015].

One of their primary contributions was demonstrating the possible efficiency gains of representing the game board as a series of chains in a game of Strings and Coins, which reduces the state space of the game. The game board was re-represented by creating a table of 12 basic chain structures, which are used to comprehensively categorize each board state; Utilizing these structures in its algorithms, QDab decides which move is the most beneficial to make (*see* ***Proposed Work*** *for more information*).

QDab was able to play the game effectively, beating most other competing A.I. in Dots and Boxes, but has a few downfalls: First, it runs very slowly (on a modern laptop), requiring up to a minute of processing time for each move. Also, it seems to be space inefficient, requiring a large amount of memory for moves. In spite of this, the research team contributed several useful strategies for implementing a Dots and Boxes bot.

Other works involved with implementing an AI for Dots and Boxes include works by teams from the University of Sydney, who made use of Evolutionary Reinforcement Learning [Knittel, 2007], and work by Lex Weaver and Terry Bossomaier, who also made use of an ANN [Weaver and Bossomaier, 1992]. The techniques integrated into the development of QDab, along with the works of David Wilson, have provided a solid path for further research and development.

The goal of our research was to expand upon what has already been done by implementing the environment described in *Improving Monte Carlo Tree Search* and trying new learning algorithms. Unfortunately, the paper did not always give enough of a detailed description of how the agent was trained in the Monte Carlo Tree Search which greatly hindered progress. The board representation, however, had a very detailed description, so that was our starting point.

The researchers represented the board by keeping a list of chains which tracks every edge in the board exactly one time (*more information about chains in the next section*). Representing the board with chains simplifies the definition of a state also also improves efficiency. Using chains seems to be the best known candidate for board representation, but the method of learning is still open for debate. The authors of *Improving Monte Carlo* did good work; Their agent outperforms the majority of the other Dots and Boxes A.I. and is genuinely good at playing (although slow). Since the 5 x 5 board has already been well covered, we decided to move up in length and width to a 6 x 6 board with hopes both to construct an agent capable of performing well in a larger state-space and to have a good runtime efficiency.

Proposed Work

The objective of this project is to design and implement an AI to play Dots and Boxes on a 6×6 board using the TD-Learning, Q-Learning, and later an artificial neural network algorithm with back propagation utilizing data given by Monte-Carlo tree search. There have been several attempts to create an AI for Dots and Boxes, but they have been implemented to solve games that have fewer than 36 boxes.

Although this project aims to solve a board size of only 6×6, this still requires a considerable amount of work, as the number of states on a board of this size is (naively) 84!. Because the naive implementation of the board will give a state space of such a large size, we utilize more efficient implementations for this board by applying Zhuang et al's Strings and Coins model in order to reduce the search space.

Strings and Coins appears as a sort of "visual inversion" of Dots and Boxes: It consists of a grid of coins, each with four strings sticking out to the left, right, top, and bottom. Once all four of the strings are "cut", the player making the final cut claims the coin. So, in the case of Dots and Boxes, each un-drawn edge can be thought of as an uncut string. Figure 2 (below) shows an example of an equivalent situation in Dots and Boxes and Strings and Coins. Representing the board in Strings and Coins opens

up the possibility of representing the board as a list of chains.

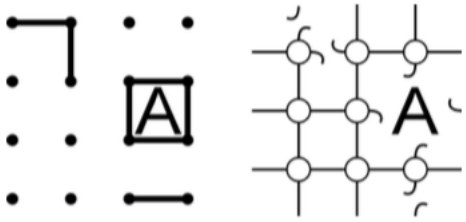

Figure 2: Two equivalent game states, one in Dots-and-Boxes notation (left), the other in Strings and Coins notation(right). Taken from Barker and Korf [July 2012]

In Dots and Boxes, a chain is a set of edges connected in such a way that cutting a given edge will always result in the same transformation to the given chain. These transformations include a union with another chain, a split, and a change. It turns out that there are 12 distinct types of chains and every edge on the board is always part of exactly one chain, displayed in the figure below.

It is notable that this board representation assumes that coins which are tied to the outside of the board are connected to "squares" existing in a hidden layer bordering the board.

TABLE I: *Classification of chains and edges.*

Chain type	Chain	Length	Candidate edge category
1	○ + □	0	0
2	○ + ○ − ○ − □	≥ 2	0
3	○ + ○ + □	1	2
4	○ + ○	0	0
5	○ + ○ − ○	1	0
6	○ + ○ − ○ − ○ − ○	≥ 3	0
7	○ + ○ + ○ − ○	2	2
8	□ + □	0	1
9	□ + ○ − □	1	1
10	□ + ○ + ○ − □	2	2 (middle), 3 (left)
11	□ + ○ − ○ − ○ − □	≥ 3	3
11	□ + ○ ¦ ¦ ○ − ○	≥ 3	3
12	○ + ○ ¦ ¦ ○ − ○	≥ 4	3

Figure 3: The twelve chain types as described in Zhang et al. [November 2015]

Each chain type has either one or two candidate edges which can be regarded as the only acceptable moves for the agent.

The candidate edges are also split up into four different categories which are determined by the position in the chain and the chain's type. Since every edge is part of a chain and cutting a candidate edge will always lead to a transformation which will result in a different set of chains on the board, it is possible to represent the board as a list of chains instead of keeping track of each individual edge. Using this approach greatly simplifies the state space. It is difficult to say exactly how much smaller the state space is because of the different properties of each chain type but our approximation for the reduced state space size is:

$$|V| \approx {}^{95}C_{84} = 7.8083582 \times 10^{13}$$

We have used a variety of reinforcement learning algorithms such as TD(0) Learning and Q-Learning with Off-policy TD Control. We intend to use reinforcement learning in combination with an artificial neural network to implement our 6×6 Dots and Boxes game. The state of the game will be the current amount of chains of given types in the current board game (the size of each element in set C below), as well as the number of loops of given lengths and the valency (number of remaining strings) of each node in set V, as described in (Zhuang et. al. 2015)(pg. 318).

$$C = \{c_1, c_2, ..., c_{12}\}$$

$$V = \{v_1, v_2, ..., v_{35} \mid v_i \in [0, 4]\}$$

$$S = \{(|c_1|, |c_2|, ..., |c_{12}|, v_0, v_1, ..., v_{35}) \mid c_i \in C, v_j \in V\}$$

The agent's action is either to cut one of any edge between 2 coins or to only cut candidate edges. In a 6×6 board game, there are a total of 84 edges which can be cut. Thus the set of edges A would be:

$$A = \{0, 1, 2, 3, ..., 83\}$$

As for the reward, there were several approaches that we have tried. Some of these include: rewarding the agent 1 point if it wins a game and -1 if it loses a game, rewarding 1 point for winning a box, or awarding points based on the category of the edge that is cut.

Given more time, we would use the Improving Monte Carlo Tree Search algorithm discussed in (Zhuang et. al. 2015) to implement Artificial Neural Networks. We could verify, as well as generalize, their results to a larger-sized board this way. The training data for the ANN would be generated using the game between 2 reinforcement learning agents implemented in phase 1. The state of the game and the player's move would be fed into the neural network model. There would be one logic unit in the output layer of the ANN, with a value ranging between 1 and -1, where 1 represents the winning and -1 represents the loss of the current player using the $tanh(x)$ function below:

$$tanh(x) = \frac{4x}{1 + e^{-x}} - 1$$

Experimental Evaluation

Tools and Methods

We used multiple machines to train and test the performance of our agent. Here are the machines and the specifications of them:

- MacBook Pro "Core i7" 2.4 GHz with 16GB RAM
- HP Envy 14 "Core i5" 2.4 GHz with 12GB RAM
- Asus Zenbook "Core i7" 2.6 GHz with 16GB RAM

Phase I: Implement Environment

After implementing the environment, we have reduced the state space down to be less than 84^{12} states, which is much smaller than the size of the naive implementations such as 2^{84} or $84!$ states. Note that this new size of the state space is only an approximation, the true size is believed to be much smaller than this.

Unfortunately, our implementation proved to be far more complex than we anticipated. The implementation reduces the state space considerably but at the cost of requiring a deep understanding of the approach in order to get it working. A large portion of our time was spent creating pseudo code and walking through unique cases we discovered that might cause problems. After we were confident with our plans, the code was implemented. As far as using reinforcement learning, we implemented several naive approaches and a simple Monte Carlo Tree Search Agent.

Ideally, every state ought to be explored. With a state space of size (up to) 84^{12}, however, and with each state calculated to be around 30KB, this would require data amounting to 3.7×10^{18}GB in order to store. The true size of the state space is smaller than this, and in the optimization process we have tried to minimize it, but based on this calculation we nevertheless would very likely not have enough memory to store all the states in the memory. Because of this, our goal was to implement a model (hopefully an ANN) which can predict values which would occur in the table, and begin to use Monte Carlo Tree search when the size of the remaining states on the board is minimal.

When considering the size of the state space, it is easy to see why the Monte Carlo Tree is useful if we are going to use reinforcement learning: The tree only keeps track of states that have been encountered before. In other words, if a state has never been encountered, it does not exist in the tree. This way, any obscure states which very rarely occur naturally will not be take up space in the computer's memory as dead weight. Of course, the more trials we do, the larger the tree will get and the more memory it will take up. This is where an Artificial Neural Network would (hopefully) help.
Until the board is reduced to a size which has a smaller state space, it may work to make moves using an ANN. The ANN will observe the state (as described earlier) and make a decision. This is very similar to the approach taken by Zhuang et al. At this point our entire project has been modeled after their description and approach. Once our agent and environment gets to a functional state, we will begin exploring ways to improve upon their performance.

Phase II: Implement Reinforcement Learning Agent

Currently, we have trained several agents based on different reward policies. The four most notable of them are listed below together with their performance against a mindless player and against each other after playing 1000 games. All agents were trained with a learning rate of 0.0001 and a gamma rate of 0.9.

Agent #	**Reward(s)**	**Learning Approach**
Agent 1	+1 cutting a an ideal edge, +0.5 for cutting a less ideal edge, -1 for cutting a bad edge (However, this policy is unstable because the list of ideal edges are not frequently updated to improve efficiency)	Tabular TD(0) Learning for 3,000,000 episodes
Agent 2	+1 for making a box	Q-Learning with Off-policy TD control for 2,000,000 episodes
Agent 3	+1 cutting a an ideal edge, +0.5 for cutting a less ideal edge, -1 for cutting a bad edge; only allowed to cut candidate edges and the rest are hidden from agent	Tabular TD(0) Learning for 1,000,000 episodes
Agent 4	+1 for winning a game, -1 for a loss or a draw. Only allowed to choose candidate edges	Monte Carlo Learning for 1,000,000 episodes

The members of our group also played 5 games each against agent 2, with the results in the table below.

Player	**Human wins**	**Draws**	**boxes claimed by agent (AVG)**
Dale	5	0	7.8
Nam	4	1	15
Andrew	4	1	11

Performance Measures

Performance was measured by having the agent play a number of games against an opponent, either the random agent that it was trained against or, in the case of agent 2, both the random agent and agent 1.

After performing several tests, it seems that Agent 2 has the best performance out these agents against random

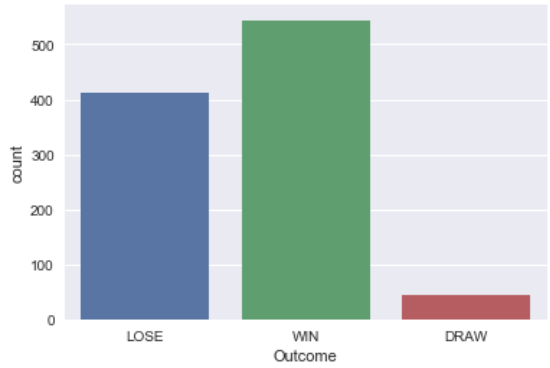

Figure 4: Performance of agent 1 against a mindless agent after 1000 games

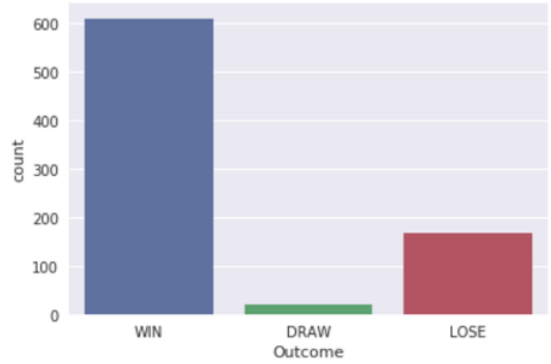

Figure 5: Performance of agent 3 against a mindless player after 1000 games

(mindless) moves. However, when we tested against a real human player, the agent is no better than a naive computer player. Although it recognized where to draw a line to form a box, it often gave away boxes to the opponent by making the third line to draw a box, which gave the other player the move to draw the last line to gain a score. At the moment, we are working on other reward policies such as giving reward when the agent wins a game. Also, another approach that the team is working on is to move from the Tabular TD(0) Prediction to Q-Learning with off-policy TD control to see if the performance would improve.

The Monte Carlo agent, Agent 4, needs some work. The performance measures for it have been inconsistent up to this point; This is likely due to the large state space, the reward system, and the fact that (up to this point) we haven't had the agent update its policy based on its findings (pure exploration without exploitation in training). Since the rewards are based on win/loss, the agent needs to process an extremely large number of episodes before it will learn which paths to take. Against a mindless agent, Agent 4 performs almost as well as Agent 2. However, when it faces a naive strategy, the agent wins only about 40 percent of the time, and against a human, its success rate is even

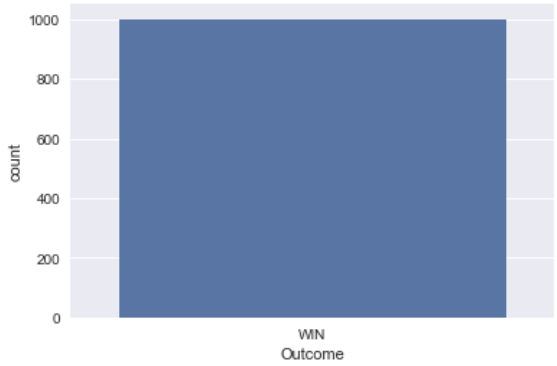

Figure 6: Performance of agent 2 against agent 1 after 1000 games

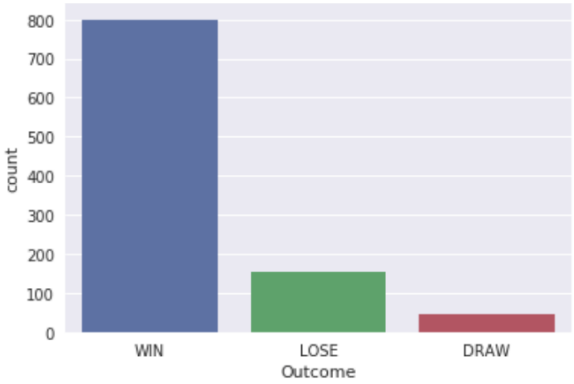

Figure 7: Performance of agent 4 against mindless agent after 1000 games

worse. By having humans play against the agent, it was discovered that the agent has learned that it is important to make long chains but it fails to claim the correct boxes when the time comes. We have done some experimentation with exploitative training but so far it has proven to work too slowly to make a significant difference; the large size of the reward table and the amount of possible moves make it difficult to exploit the best moves on a large scale. Another strategy, attempted with Agent 4, was to train it against a pure-random agent. Based on the results so far, this approach again seems to make little difference in performance.

Since we implemented the Monte Carlo agent late in the course of the project, we had little time to allow it to train for long periods, so it is possible it could improve with a larger episode size. In the mean time, we worked on the optimization of the exploitative strategy in hopes of significantly shortening the processing time. Future optimizations may include adding an undo function to the board to allow an agent to make a move and undo it instead of making a copy of the board and reducing the amount of times that lists need to be searched through or compared. Successfully implementing Monte Carlo Search has set the stage to add an ANN to our implementation, which is the last, most significant portion of this project that remains unimplemented.

To help evaluate the performance of agents playing against human players, we implemented a simple user interface for the player to interact with. To do this, we found an open source Dots and Boxes game which was programmed in Python and modified it to work with our agents.

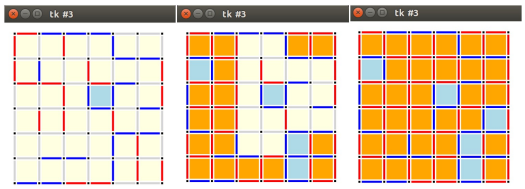

Figure 8: An example of a game played on the GUI. The human player is represented by the orange boxes and the agent is represented by the blue ones.

Conclusions and Future Work

Based on the experimental evaluation of the agents above, we can conclude that Agent 2 is by far the best among the agents we were able to implement. However, when testing against real human players, it performs not much better than a naive agent. Although it usually captured potential areas with 3 edges, it often gave up boxes to the opponents by making the third line to make a complete box. The performance of all the agents we created is disappointing, but that does not mean that our research has been a complete failure. The environment that we set up is robust and should be adaptable to any type of learning.

Unfortunately, we were unable to recreate a Monte Carlo Tree Search that could perform as well as other agents that were researched. There are a variety variables at play that can be controlled with MCTS and it's possible that we simply did not choose the correct parameters. Another possible problem that we suspect is that we trained the agent against a relatively mindless player, making it learn extremely slowly in such a large state space. Our attempts to train the agent against itself took far too long to search through the possible moves and make a choice for the agent to learn. We lacked the hardware and time for training this sort of agent effectively. Improving the Monte Carlo agent would be an excellent starting place for future work.

Other future work could include further optimization of the environment. The faster that an agent is able to analyze the environment and make a choice, the faster it will be able to play through games. A faster environment, therefore, would mean faster learning which would be helpful for training any agent. Probably the most important future work, however, would be to implement an Artificial Neural Network agent. A Neural Network implemented correctly could potentially eliminate the problems discussed earlier with the Monte Carlo Search tree, as far less information would need to be stored and searched when the agent makes a move. After successfully implementing an ANN, making use of minimax-search on smaller board states as in Zhang et. al. could further improve performance and/or expedite the process of training the ANN.

Overall, although we haven't achieved the goals which we originally set for this project, we have been able to test several reinforcement learning methods, as well as create a Dots and Boxes environment in Python which greatly reduces the state-space of the game, and could easily be expanded upon for future work.

References

Solving Dots-And-Boxes
Joseph K. Barker and Richard E Korf July 2012. AAAI Press.

Dots and Boxes and Strings and Coins. Erik D. Demaine September 2001. Mathematical Foundations of Computer Science.

Concept Accessibility as Basis for Evolutionary Reinforcement Learning of Dots and Boxes. Anthony Knittel, Terry Bossomaier and Allan Snyder April 2007. Computational Intelligence and Games.

Evolution of Neural Networks to Play the Game of Dots-and-Boxes Lex Weaver and Terry Bossomaier May 1992.

Dots-and-Boxes Analysis Index David Wilson 2014. University of Wisconsin.

Improving Monte-Carlo tree search for dots-and-boxes with a novel board representation and artificial neural networks Zhuang, Shuqin Li, Tom Peters, Chenguang Zhang November 2015. 2015 IEEE Conference on Computational Intelligence and Games (CIG).