Search- and Learning-Based Game Playing for Dots and Boxes

BY EVAN LIU & DEBNIL SUR

Abstract

Dots-and-Boxes, a popular children's game, also has deep mathematical properties that make it amenable to a general game-playing approach. In this paper, we implement a game-playing agent primarily based upon search heuristics, with basic learning. We compare its performance to Dabble, an existing agent based upon game-playing techniques and using TD learning. This faciliates a comparison of search-based and learning-based approaches to the game. We find that a learning-based approach is substantially better when playing second but almost comparable to our advanced search-oriented agent when first. This thus provides insight into the comparative efficacy of search- and learning-based techniques as well as optimal side-based strategy in Dots and Boxes.

1 Introduction & Motivation

The advancement of artificial intelligence has resulted in fundamental questions about the differences between the human mind and Turing computation in elementary cognitive tasks. Simple games taught to young children serve as testbeds for the difference of human analysis—chess problems rapidly solved by novices are incorrectly handled by Deep Thought, a chess computer of grandmaster rank [1]. We will study the application of artificial intelligence techniques to the game Dots and Boxes. This game starts with an $m \times n$ grid of dots; players connect adjacent dots via horizontal or vertical lines; the player who draws the final connecting line to create a box is rewarded with that box and gets to repeat her turn; and the game ends when all possible boxes have been created. Though simple, the game is analytically unsolved for any dimension greater than a 4×5 grid. The primary challenge arises in its massive state space. For instance, one of the largest solved problems, the 4×4 grid, has 40 edges, a state space of 2^{40}, and a naive search space of 40! [1]. Comparing human and computer play can thus provide greater insight into game playing strategy at large.

2 Model

We model Dots and Boxes as a two-player zero-sum game. By definition, a game involves turn-taking with full observation. This is a type of state-space model. For all games, there exist s_{start}, a starting state; Actions(s), possible actions from state s; Succ(s, a), a resulting state from choosing action a in state s; IsEnd(s), whether s is an end state representing the end of the game; Utility(s), the agent's utility for a state s; and finally, Player(s), the player who plays at a state s. By definition, this game has two players; furthermore, as this is a zero-sum game, the sum of both players' utilities must be zero.

For our model, we define these parameters as following: the players for this game are "agent," our game-playing AI, and "opp," the opponent game-player. Each state s contains the current edges on the grid, each player's score, and whose turn it is. The actions at state s, or Actions(s), contain the possible edges that Player(s) can draw. IsEnd(s) is true if s has all possible edges drawn. Utility(s) is defined for a particular agent to be the number of boxes that the agent has drawn minus the number of boxes that the opponent has drawn. In an end state, an agent has won if its utility is positive, otherwise the opponent has won. Finally, Player(s) will either be the agent or the opponent, and it represents the player who is playing at state s. This satisfies the basic requirements of a two-player zero-sum game. We evaluate our agent against Dabble, another game-playing agent written by JP Grossman. While it utilizes rudimentary domain knowledge, the bulk of its performance is based upon game-playing techniques. It has experienced great success against human players, as it was used to accomplish a top-15 global rating in the popular Yahoo! Dots adaptation of Dots and Boxes [1]. It thus serves as an effective metric of evaluation for our approach.

Notably, the Sprague-Grundy Theorem effectively solves many two-player zero-sum games similar to Dots and Boxes [1]. The important distinction in the case of Dots and Boxes is that the winner of Dots and Boxes is not decided by who makes the last move, but rather, who has the most boxes at the end of the game. Because of this, the Sprague-Grundy Theorem does not apply to Dots and Boxes, and the game is not solved for grids larger than 5x5.

3 Approach

We developed the following infrastructure to allow us to sanity-check our progress. First, we developed a random agent who would randomly select an edge from the set of possible moves and draw that edge. We expected any agent that we would write to always win against the random agent. Next, we created a human agent, that would allow us to input moves manually to the game via command line, allowing us to match our AI against humans and Dabble. This infrastructure creates the necessary foundation for development and testing our agent.

3.1 Baseline and Oracle

We defined a baseline and an oracle in order to get a rough sense of the problem at hand, giving us an upper bound and a lower bound on expected performance of our AI. The baseline is defined to be an average human player. In this case, Debnil played 20 games against Dabble on a 3x3 grid, 10 as the first player and 10 as the second player. Debnil lost all of these games, surprisingly, giving us a 0/20 as the baseline. We expect that any AI written should at least perform better than an average human, although expert humans have been known to outplay Dabble [2].

We defined the oracle to be a third game-playing agent known as PRsBoxes written by Paul Stevens. This agent utilizes several extremely game specific techniques that allows it to analyze positions far more effectively than Dabble, which uses general Artificial Intelligence techniques and a couple of game-specific optimizations. In this paper, we seek to achieve the best results using primarily general Artificial Intelligence techniques, so we expect this utilization of Dots and Boxes specific techniques to perform better than our AI. We choose to use only general techniques for to restrict the scope of our project, although we may use more specific techniques in the future. PRsBoxes defeated Dabble 44 to 16, giving us an upperbound of a 73.3% win rate [3].

3.2 Minimax

We employ the Minimax algorithm as the basis of our approach. We chose the Minimax algorithm over Monte-Carlo Tree Search for several reasons, although the choice may appear counter-intuitive at first. Minimax is known to be better suited for precise and tactical games, whereas Monte-Carlo Tree Search is better at more strategic games, where individual moves matter less [4]. Monte-Carlo Tree Search is notably a particularly good choice for games that have huge branching factors. Dots and Boxes is a game that branches significantly, but the inability of Monte-Carlo Tree Search to perform as well tactically, forced us to choose the Minimax algorithm. The Minimax algorithm traverses the game tree as follows. It expects that its opponent makes the optimal moves, and chooses the move that yields the highest score given that the opponent moves optimally.

3.3 Pruning

Since Minimax in general requires full search of the tree, we employ Alpha-Beta pruning to make the problem more tractable. Alpha-Beta pruning allows us to prune out branches that are known to be sub-optimal by keeping track of the current best and worst possible scores. Alpha-Beta pruning can potentially prune out all but the best branches if the best moves are examined first. We use the follow move heuristic in order to attempt to prune the most branches. Moves that do not complete a third edge, and hence don't allow the opponent to create a box are considered first. Barker and Korf have shown this simple ordering heuristic to be very effective [1].

We also employ Transposition Tables to further decrease the amount of computation required to make moves. Transposition Tables take advantage of the fact that many positions may be reached in different ways and stores the evaluation of that position, so that if it is reached again in some other move ordering, the value of the position does not need to be re-calculated. This method is particularly useful for Dots and Boxes as the Transposition Tables in Dots and Boxes requires only the score and the configuration of the edges, and not who owns each box. Hence, we can collapse a lot of states into one Transposition Table entry. Experimentally, we have found Transposition Tables to decrease computation time by up to 100 times. We evict from the Transposition Table based on which entries are least frequently used. More specifically, each entry of the Transposition Table stores the score of a position based on the depth of evaluation and the position.

3.4 Learning

We also implemented several Reinforcement Learning aspects to our agent. We implemented a simple mechanism that updates an agent's Transposition Table based on losses. If the agent loses, it will subtract 1 from the evaluation of every entry of the Transposition Table associated with the moves that it makes. This discourages the agent from playing the exact same moves again, which is a risk of the heuristic ordering of the moves for Alpha-Beta pruning. In a direct comparison between a version of our agent that used this reinforcement learning aspect vs our agent that did not use this reinforcement learning aspect, the version that did use the reinforcement learning won about 55% of the games, a slight improvement.

We also attempted to evaluate TD-Learning as our evaluation function in Minimax, instead of the basic evaluation function that we were originally using, which only returned the score of the position based on number of boxes. The main idea is that over time, the agent learns weights for certain features and assigns values to positions based on those weights and features. However, we encountered a problem with our approach in that our usage of TD-Learning is not computationally feasible within our current framework, as it took far too long for the agent to evaluate a position. This is due to several factors. First, we wrote our agent in Python, which is well known not to be the highest performing language, over others such as C++. Second, there are a few implementation quirks, that slow down positional evaluation that we used in order to rapidly prototype our ideas. In the future, we intend to change these and expect TD-Learning to significantly impact our AI's strength.

3.5 Domain Knowledge

In addition to all of the above general game playing techniques, we implemented one game-specific improvement taking advantage of the structure of chains in Dots and Boxes. There are several possible chain structures that arise in Dots and Boxes, displayed in Figure 1 in the Appendix. The chain labeled A is called a half-opened chain, since it is open only at one end. With half open chains, there are only two possible optimal moves – to either complete each box in the chain or to complete all but two boxes, leaving the two boxes incomplete for the next player to take in order to maintain control over the game. The chain labeled B is called a closed chain. With closed chains, there are again only two possible optimal move sequences. One possible move sequence is to again complete all of the boxes. The other possible move sequence is to complete all but four boxes, sacrificing those to the opponent to maintain control. We take advantage of these chain structures by massively pruning the game tree when these structures arise, by only considering the possible optimal moves.

3.6 Example Play

Finally, we provide a short example of our desired approach. Here, it's important to note that even a small game has a massive state space. For instance, while small, the 4×4 game has 40 edges, a state space of 2^{40}, and a naive search space of $40!$ [1]. For this reason, it's one of the largest solved instances of this game. As a result, drawing the entire state space for a small game is extremely difficult. In lieu of this, we will consider a portion of the moves in two levels of the game tree for a 2×3 game, using a minimax approach, to demonstrate how the game-playing agent makes decisions. Specifically, we will examine the first two moves; a sample opening sequence of the game is shown in Fig. 2 in the Appendix. Here, both player 1 and player 2 are minimax agents operating with a lookahead of 2 levels. As the heuristic for evalulation will be equal for all initial moves, the temporary evaluation function will return 0 for all games. Consequently, player 1 draws an edge at random from the edges with score 0. Similarly, Player 2 looks 2 levels deep and sees a score of 0 for effective opening actions. Therefore, she also draws an edge at random from these edges. At future levels, the agents will also utilize the basic ordering heuristic explained above, in which they avoid drawing the third edge in any box unless absolutely necessary. However, at both beginning and later stages, the general principle of minimax will further prune the search graph as more positions are strongly desirable than others. This example illustrates how each agent applies that technique to drawing its initial edges.

4 Data and Experiments

4.1 Experimental Design

As mentioned above, the selected baseline and oracle provide a lower and upper bound on the performance of our algorithm. We hope to consistently defeat a human player, but we also expect to perform reasonably well against Dabble. We expect Dabble to still be significantly better, but we attempt to control its advantages in implementation to allow for a fair test. First, it is important to note that Dabble has a few general optimizations implemented that we were not able to successfully implement. In terms of general game-playing, while both our approaches utilize alpha-beta pruning, Dabble makes use of a learned evaluation function as opposed to our chosen heuristic. However, both programs have the same game-specific optimizations. Thus, this comparison of approaches will provie valuable insight into how significantly a learned function affects play, versus the approximated heuristics that we used.

Game speed also significantly affects the design of our experiment. Because Dabble is written in C++, it traverses the game's massive state space much faster than our Python agent. This speed difference is best seen at depth 3: while our game playing agent still takes a few seconds to return a turn at early stages (due to the exponential state space), Dabble returns moves practically instantaneously. At search depths higher than 3, the time it took to return moves made manually interfacing our system with Dabble infeasible with respect to time. As a result, in order for our program to return its move in a reasonable time, we limited the depth of search for both agents to 3. Finally, also due to excessive time, we could not test our AI against Dabble at higher dimensions (above 4×4). This has both pros and cons. On one hand, because Dabble has been used to solve games fully at the dimensions we have tested, we know that perfect play is feasible; it just becomes a question of the margin of victory or defeat. On the other hand, because Dabble's evaluation function is learned from already solved games, its probability of winning is also much higher for these boards.

4.2 Results

On a 4 by 4 board, our algorithm was able to win 100% of games against a random agent and 100% against a human agent. Against Dabble (with the settings adjusted as explained above), we won 25% of games. Due to the time required to manually interface with Dabble effectively, 20 games were played against each agent. The specific winning percentage of our agent against each type of player is seen in the table below.

	vs Random	**vs Human**	**vs Dabble**
Our Agent First	100%	100%	10%
Our Agent Second	100%	100%	40%

5 Conclusions

5.1 Analysis

As expected, our algorithm far-outperformed the random and human agents. This demonstrates that even basic lookahead and a hardcoded heuristic function, as opposed to a learned evaluation function, are sufficient to significantly improve performance when compared to non-intelligent game-playing tactics. Along the same lines, our performance against Dabble also provides a valuable quantitative insight into the performance of ordering heuristics versus learned metrics like traditional evaluation functions. In particular, our agent performed far more comparably to Dabble when playing second than playing first. In the context of the implemented heuristic, this makes sense. We only implemented a basic ordering heuristic that avoided drawing a third edge; obviously, this only becomes useful when there are already edges on the board. When playing first, especially in the opening series of turns, this heuristic will not be used. Thus, the opening of the game is functionally random until enough edges are drawn. This explains our agent's drastic improvement in going second, as opposed to going first. Along these lines, our agent's comparative performance also demonstrates the strategic utility of going second in the game. Even though a human player does not have the same look-ahead capability of an AI, the basic ordering heuristic is still one that can be utilized to avoid letting the opponent draw boxes. When initial moves are made by the adversary, the space of possible moves is substantially reduced; it is far easier to memorize optimal strategy, as expert human players have done in past [2]; and playing optimally becomes much easier.

We illustrate an example of this opening strategy in the Appendix on a 4×4 game. A sample opening sequence of the game is shown in Fig. 3 in the Appendix. Here, Player 1 is our minimax agent, playing at depth 3. Player 2 represents inputted moves from Dabble, being played on another computer also at depth 3. As the heuristic for evalulation will be equal for all initial moves, the temporary evaluation function will return 0 for all games. Consequently, Player 1 draws an edge at random from the edges with score 0. Player 2, representing Dabble, has the advantage of a learned evaluation function for each possible move. Thus, it chooses the subsequent move with the highest likelihood of victory. On the other hand, if Player 1 were in this situation, he would look 3 ahead to decide his next move. While he would avoid drawing edges that would likely result in him drawing a third edge, he would not have the full benefit that Dabble has in this situation. Notice that as more edges are drawn, the ordering heuristic increasingly becomes a powerful way of narrowing the search space. Later in the game, it becomes comparable to learned evaluation functions in choosing advantageous moves [1]. Of course, if the player is significantly behind from the beginning, this late-game equivalence becomes far less relevant. Along those lines, the limitations it places on effective early play are illustrated by the substantially weaker performance when our agent goes first, as opposed to second.

Finally, although displaying a full game would be optimal, the massive state space of the game makes an entire display infeasible in the space of this paper. The game logs that accompany this paper provide a record of the games played that constitute our data. They can be examined to see the increasing efficacy of the ordering heuristic later in the game, as discussed above.

5.2 Further Work

As mentioned above, further work primarily centers upon enhancing the performance of our program. This will both improve the speed of computation and facilitate necessary simulations to learn an evaluation function. First, we must move our Python agent into C++. Doing this will significantly improve program speed and performance, as it accelerates searching the state space. Second, we will run Monte Carlo simulations using the optimized code to build a large data set of played games. This can be used to learn evaluation functions and therefore determine intermediate evaluated values for the variety of states corresponding to different board formations. This approach is the same that the creator of Dabble used, and by learning the evaluation function, performance should be comparable to Dabble's. Finally, to improve the testing procedure, we will programmatically interface our C++ version with Dabble; because both are in the same language, it should be substantially easier than our previously failed attempts with the Python/C API.

Apart from computational improvements, there are a few other game-based optimizations that we can also add that have been detailed in other work. One key improvement would be an opening book. This would hardcode optimal edges to draw in the first few turns. As mentioned above, one of the biggest flaws in our current strategy is opening sequences. This primarily stems from the hard-coded evaluation function, so replacing that with a learned function should lead the agent to choose optimal moves to start the game. Another significant improvement that could significantly reduce the size of the state space is analysis of symmetries. The mirror image of a state is also a game with the same optimal strategy; all instances have both horizontal and vertical symmetry. Storing by symmetry could reduce the size of the search space by a factor of 4 [1]. Adding other game-specific optimizations akin to those in Paul Stevens' implementation (referenced above as an oracle) could also significantly improve performance.

5.3 Takeaways

Simple for toddlers and complex for machines, games serve as a fascinating case study of potential difference in human analysis. Dots and Boxes has been analytically solved at lower dimensions by alternative game-playing techniques. As such, deeper study of it using principles from class serve as an interesting endeavour in game playing strategy at large. In particular, given the scant literature on this game, our work provides a useful comparison of learning and search-based approaches to Dots and Boxes. Due to computational difficulties in our implementation, we implemented a series of improvements to searching, with naive learning and instead focused on comparing this with learned evaluation metrics. We found that when playing second, search-based heuristics can perform comparably to learning techniques. This provides insight into the performance of intelligent heuristics against learned functions, as well as rudimentary understanding into the benefits of going first, as opposed to second, in this game. Further computational power will improve our ability to play against Dabble more fully, but our work represents an important step in the right direction.

6 References

[1] Barker, J. K., & Korf, R. E. (2012, July). Solving Dots-And-Boxes. In *AAAI*.

[2] Wilson, D. (2010). Dots-and-boxes analysis index.

http://homepages.cae.wisc.edu/<sim>dwilson/boxes/.

[3] Roberts, P. (2010). Prsboxes. http://www.dianneandpaul.net/ PRsBoxes/

[4] Grossman, J. P. (2010). Dabble. http://www.mathstat.dal.ca/<sim>jpg/ dabble/

Appendix

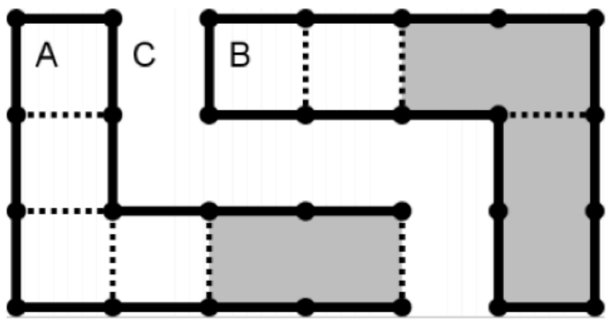
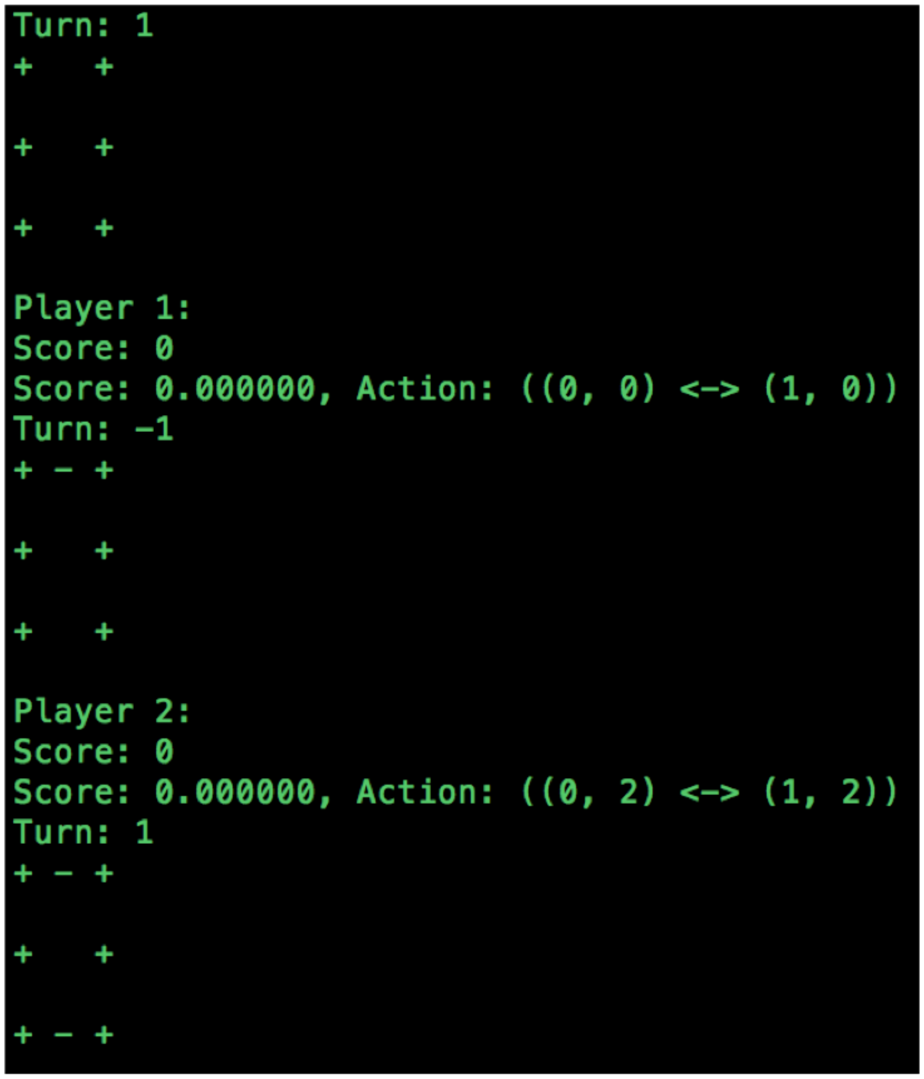
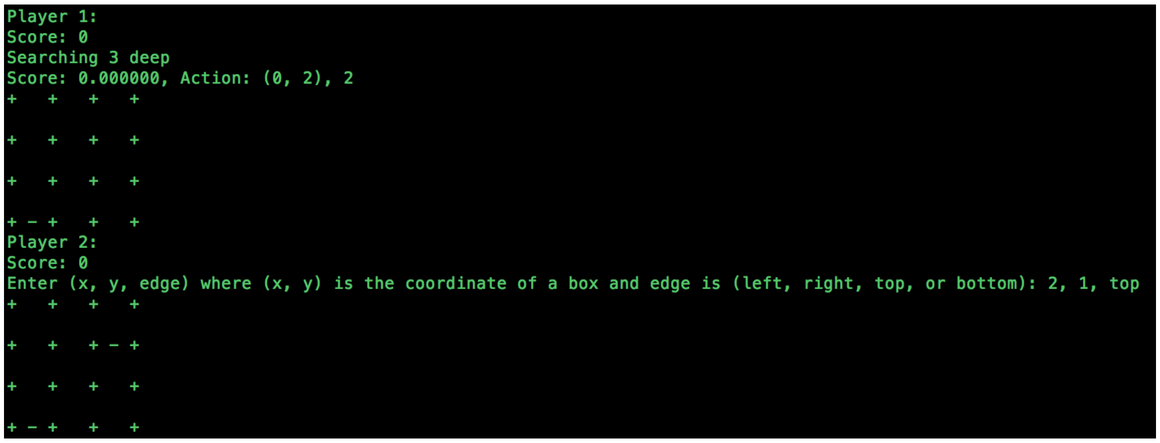


Fig. 1, an example of a chain [1].



```
Turn: 1
+   +

+   +

+   +

Player 1:
Score: 0
Score: 0.000000, Action: ((0, 0) <-> (1, 0))
Turn: -1
+ - +

+   +

+   +

Player 2:
Score: 0
Score: 0.000000, Action: ((0, 2) <-> (1, 2))
Turn: 1
+ - +

+   +

+ - +
```

Figure 2, the first move by either player in a running of the game.



```
Player 1:
Score: 0
Searching 3 deep
Score: 0.000000, Action: (0, 2), 2
+   +   +   +

+   +   +   +

+   +   +   +

+ – +   +   +
Player 2:
Score: 0
Enter (x, y, edge) where (x, y) is the coordinate of a box and edge is (left, right, top, or bottom): 2, 1, top
+   +   +   +

+   +   + – +

+   +   +   +

+ – +   +   +
```

Figure 3, where Player 1 is our agent and Player 2 is Dabble.