Dots and Boxes

by Debnil Sur & Evan Liu

Alpha-beta

Transposition tables

Chaining

Learning feature – loses, updates good and bad features

TD learning

TD learning – should improve lots

Mediate between game and dabble

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. 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 an end 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.

Let us now define each of these parameters for this game. 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) checks if s has no possible actions. Utility(s) is only defined if IsEnd(s) and will be $+\infty$ if the agent wins, 0 if a draw, and $-\infty$ if the opponent wins. 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 can also view this game as a system. The input is an empty playing board (with mutually agreed upon dimensions m and n), and the output is a fully connected board with a final score, winner, and loser.

To measure our algorithm's efficiency, we will check its performance against three classes of metrics. The first, a purely random agent, just arbitrarily draws edges. A human can beat this agent, so we expect even a rudimentary AI to perform excellently. The second is a human agent. While able to think strategically, a human does not have the ability to look ahead that a game-playing agent does. As such, we expect our AI agent, with some look-ahead, to consistently defeat a human. The final is playing against another game-playing agent, namely Dabble. Though successful, it does not utilize the minimax/TD learning approach, so we will iterate upon our approach until we can defeat this game. If we cannot, then it seems to indicate that the approach Dabble utilizes is preferable to ours for navigating this particular game's solutions.

3 Approach

Before applying any of the variety of techniques from AI, we built the infrastructure for the program. This entailed the creation of a dots-and-boxes game in Python, which is playable through a command-line interface. The three-pronged approach to testing also necessitated the creation of several internal agents in the system. First, a purely random agent observes the sequence of edges on the board and arbitrarily draws an edge. Second, a human agent necessitated the creation of a two-player option, where both players play via the command line. Finally, we attempted to interface with the source code of Dabble via the Python/C API. Due to the difficulty of interfacing directly, we instead utilized a parallel approach of playing the same game on both programs and interfacing moves through each game's respective command-line interface.

Measuring the performance of our approach also requires the identification of a baseline and oracle.

4 Old Progress for Reference

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. Comparing human and computer play can thus provide greater insight into game playing strategy at large.

Model

We will model Dots and Boxes as a two-player zero-sum game. We will briefly review the primary characteristics of such a 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 an end 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.

Let us now define each of these parameters for this game. 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) checks if s has no possible actions. Utility(s) is only defined if IsEnd(s) and will be $+\infty$ if the agent wins, 0 if a draw, and $-\infty$ if the opponent wins. 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.

Finally, we can apply certain heuristics to our game to reduce the state space and thus improve the efficacy of search algorithms. We will utilize evaluation functions via Monte Carlo approximation to approximate the utility function at each state, thus accelerating computation of the minimax value of a game. Using this, we will also apply alpha-beta pruning to ensure that we only search feasible parts of the game space. Though such techniques have been demonstrated to significantly improve search, they have not been utilized to a significant extent in prior studies of Dots and Boxes that applied similar principles [1]. Thus, we hope that such techniques, when applied to our model, will vastly improve the performance of subsequent algorithmic techniques.

Algorithm

Once we have characterized the space of potential states, we will apply algorithms discussed in class to efficiently search the state space and return the optimal move at each turn. This can be broadly segmented into two parts: first, establishing a general strategy to play the game, and second, learning the best policy through simulations. Note that in a zero-sum game, the opponent plays in an adversarial manner. Her goal is to maximize her utility, which trades off with the agent's. As a result, we will take a minimax approach to choosing the next move, whereby we assume the adversary will take the step that minimizes our utility and therefore choose the step which returns the maximum of these minimal values.

Next, we must learn the best game-playing policy in all. Specifically, we will apply temporal difference, or TD, learning. By running a multitude of Monte Carlo simulations, we generate potential data. Then, we learn weights $\mathbf{w}$ of the evaluation function from this data. A large quantity of simulations will help us learn effective weights. In turn, we can use the evaluation function to help our agent quickly compute the best potential action at each state, vastly improve its ability to traverse the game space, and increase its probability of winning the game.

On top of optimizations for general zero-sum games, we also make use of several algorithmic optimizations specific to Dots and Boxes. First, there are a couple chain structures that we can take advantage of displayed in Figure 1 in 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. In our evaluation metric, this second case appears rarely, since it is rarely optimal to sacrifice four boxes on a 4x3 grid.

Another optimization that we will make is to use an opening book. This will sacrifice some of the strength of the agent since it will no longer search during the opening, but it will allow the agent to move more quickly in the opening, which is the slowest period, since in an $m \times n$ grid, there are $(n(m-1)\,m(n-1))!$ possible move sequences. Finally, we will utilize symmetry on the board in order to decrease the number of states that must be searched. Symmetry can reduce the search space by a factor of four [1].

Example

We will 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 there is no learned evaluation function as of yet, the temporary evaluation function is simply the score, which is 0 for all moves of depth 1. Consequently, player 1 draws an edge at random. Similarly, Player 2 looks 2 levels deep and sees a score of 0 for any subsequent action. Therefore, she also draws an edge at random. The addition of a learned evaluation function will add intermediate utilities to these states and therefore eliminate the total randomness in the beginning game. However, the general principle of minimax is common to both the assumed zero value and the intermediate utility, and this example illustrates how each agent applies that technique to drawing its initial edges.

Initial Results

Thus far, we have implemented minimax without alpha-beta pruning and capable of arbitrary search depth; in the interest of speed, we have only tested with search depth 2. We have not utilized Monte Carlo or TD learning yet. We have only tested our algorithm rigorously against a random agent due to insufficient time to test against a human player. On all dimensions up to 4×4, our approach is winning the game with 98% probability. Thus, we have seen that at lower dimensions, which have been analytically solved, our rudimentary algorithm is performing very well even without significant lookahead or pruning of the state space. Our next steps will be to test our game against human players, implement TD learning, and finally play it against Dabble.

Conclusion

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, our work to this point has truly emphasized the symbiotic relationship of search and learning discussed in lecture. The computational intractability of searching the huge state space of Dots and Boxes forced us to utilize an evaluation function; but learning such a function will require searching enough possible futures to model the game's likely outcome.

References

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