CS 4701: Winning Bot for "Dots And Boxes"

Hangil Chung(hc685) and Taehoon Lee(tl353)

December 12, 2016

1 Introduction

"Dots and Boxes" is a simple pencil-and-paper game involving two players. The game includes dots, boxes and dot to dot grid with the following rules:

1. Given empty grid of dots (N by M), two players alternatively connect two dots by connecting unjoined adjacent dots using either vertical or horizontal line until no lines can be placed.
2. Each player earns one point for completing the fourth side of a 1x1 box and if a box is completed, that player goes again.
3. The player with more points at the end wins!

We want to create a bot that can find winning strategies for "Dots and Boxes" by implementing a simple base line Decision Tree algorithm and Minimax as part of Adversarial Search. Specifically, we wanted to measure how well these methods perform in winning the game taking into account its computational feasibility in real life game where players must make a move within certain amount of time(ex. speed chess). The questions we wanted answer through our experiments are:

1. What are the trade-offs of each algorithm proposed?
2. Does the bot take reasonable amount of computation time?
3. Does $\alpha - \beta$ pruning significantly optimize runtime?
4. What are some heuristics for this game?

By comparing different approaches in creating winning bots for "Dots and Boxes," we want to explore numerous aspects of practical applications of Decision Tree, Minimax, $\alpha - \beta$ pruning and heuristics.

2 Problem Statement and Goals

"Dots and Boxes" presents interesting problems for us to solve, such as "how does one go about creating a winning bot for the game?" As winning strategies, we decided to look at Decision Tree as a baseline algorithm and Minimax as an improved algorithm that can provide a better performance in winning the game. Theoretically, Minimax can always choose the optimal move and be able to win given that it has all the computing power and time needed to carry out the computations. During our initial prototyping phase, we realized how heavy Minimax computations are for board sizes larger than 4 and search depths larger than 5. As an extension we wanted to explore a heuristic that can significantly improve Minimax's run time while preserving the optimality of the algorithm through heuristics and $\alpha - \beta$ pruning.

The goals of our experiments are to answer the following questions:

1. What are the main trade-offs of Decision Tree, Minimax, Minimax with pruning and Minimax with heuristics
2. How much computation is feasible for Minimax? (ex. on a regular student laptop)
3. How does depth limit affect performance in Minimax?
4. How efficient is $\alpha - \beta$ pruning and how much performance boost can it provide?
5. What are some heuristics we can apply? And how effective are they?

2.1 Extension

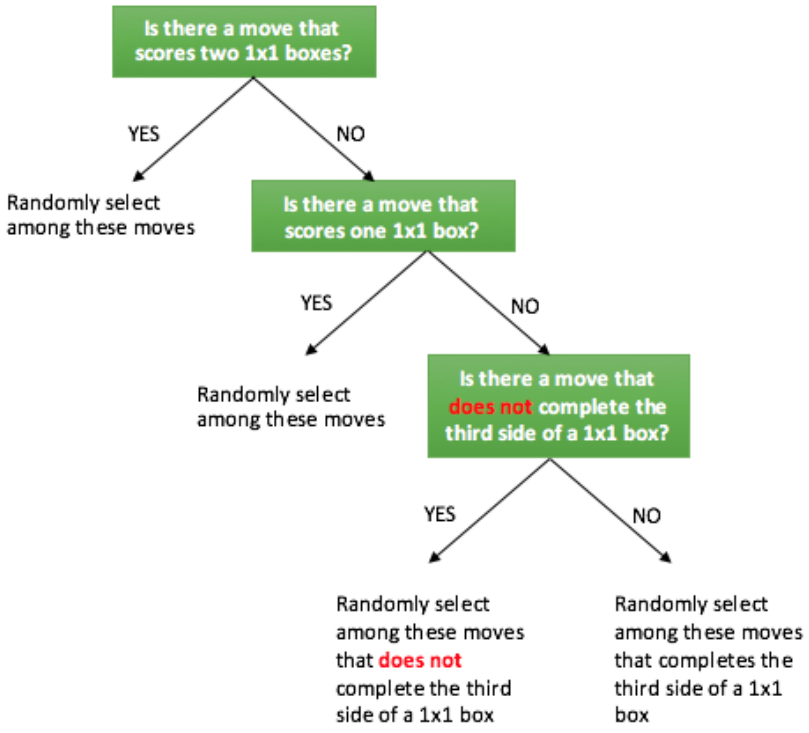
As we were experimenting different methods to create a winning bot for the game, we quickly realized how large of a computation is needed to run full Minimax algorithm for game board sizes larger than 4x4. To get around this limiting factor, we decided to explore a couple of heuristics:

1. Decision Tree + Minimax with Pruning: combining the pros of the two algorithms to create a hybrid algorithm which runs much faster while sacrificing minimal optimality.
2. Dynamic Depth: Using a dynamic depth calculated from board state and available resources to compute the minimax algorithm. In other words, run as much calculation as possible within fixed time window.

3 Approaches

3.1 Decision Tree - Base Line Approach

The goal of the game is to score more points than the opponent; this means that a simple winning strategy can be implemented by prioritizing moves that grant the most score and while avoiding moves that grants the opponent a chance to score. A relatively simple Decision Tree can translate this logic into an algorithm.



This decision tree is a baseline algorithm that we can compare more advanced algorithm against.

3.2 Minimax

The Minimax search algorithm is specifically designed for calculating the best move possible in two player games where all the information is available, which applies to "Dots and Boxes." It essentially creates a game tree that captures all the possible moves in the game where two players alternatively take turns to maximize its own gain. For example, say Player A is running the algorithm

against Player B; each level of the game tree alternates between Player A's move, maximizing the score for A, and Player B's move, minimizing the score for A. In other words, the game tree assumes that the opponent will play his/her optimal move. Therefore, the goal of Minimax naturally aligns with the goal of a winning bot that we want to create.

However, Minimax has a few shortcomings. Specifically, representing all the possible moves for relatively simple game like "Dots and Boxes" becomes computationally expensive as board size increases. The branching factor of this game is proportional to the number of moves remaining in the game, which is bounded by $O(M \cdot N)$ where M is width and N is height of game board. So with just a few depths into its search, the search space grows exponentially in respect to the branching factor, which quickly become extremely large and expensive to compute. To overcome such limitation, we decided to look at pruning and heuristics to Minimax.

3.3 Minimax with $\alpha - \beta$ Pruning

$\alpha - \beta$ pruning provides a way to reduce the number of nodes evaluated in Minimax algorithm during its game tree search. Specifically, it eliminates exploring an entire subtree of moves when at least one possible move that proves any outcome of this subtree yields worse score; therefore, moves in such subtree are not considered entirely.

Optimality of Minimax is not sacrificed by $\alpha - \beta$ pruning, because subtrees that are eliminated from search do not contain the optimal moves. So the optimal move at each level of game tree is still maintained. The most significant benefit of $\alpha - \beta$ pruning is that it reduces the search time and space. Without pruning, Minimax needs to explore $O(b^d)$ where b is the branching factor and d is depth. $\alpha - \beta$ pruning can cut the time and space up to $O(b^{d/2})$.

3.4 (Heuristic Extension) Decision Tree + Minimax with $\alpha - \beta$ Pruning

This algorithm attempted to focus on what we observed to be the pros of the two algorithms, decision tree and minimax with pruning. As we analyzed the game further, we came up with two important observations. First, we realized that the first half of the moves do not have pressing impact on the outcome of the game. Unlike other strategy games like Go or Chess, "Dots and Boxes" is a relatively simple game that does not have much delayed effects of moves placed in earlier time. Second, we observed that the branching factor during the first half of the game is much larger than the latter half, because the branching factor corresponds to the number of moves remaining. Upon further observation of the resulting data, we observed that the first half of the moves performed by both minimax and decision tree never result in a score change. Both algorithms are designed such that while there is a move option which doesn't allow the

opponent to score, it chooses that move.

Furthermore, from our previous results we observed that the latter half of the moves are critical to the ending score and that minimax tends to outperform decision tree for larger board sizes and depths. However, decision tree naturally performs much faster. Thus we reasoned combining the pros of both algorithms would help us create a hybrid algorithm which runs much faster and also performs at the level of an original minimax.

These two observations led us to experiment with a heuristic that uses simple Decision Tree algorithm described above for the first half of the game and then use Minimax with $\alpha - \beta$ pruning for the latter half. This allows the first half of the game to be played relatively fast and make computation for Minimax feasible on a reasonable hardware.

3.5 (Heuristic Extension) Minimax with Dynamic Depth

This heuristic relies on the idea that a player needs to make a move within fixed time, which translates to total computations a computer can do in a fixed amount of time. Specifically, given a fixed time limit in which a bot must make a move, we attempt to calculate the maximum number of calculations possible before submitting a move. There are two possible options. One is to fix the depth, and run the minimax algorithm for as long as possible before returning the best solution thus far. The other is to calculate the max-depth that we can fully compute the minimax algorithm in the fixed time limit, and run the bot with this dynamic depth. As the first option doesn't necessarily guarantee an optimal solution for a specific depth, we reasoned it would be better to attempt the second option of using a dynamic depth algorithm. Specifically, at each turn, using the current board state information of number of possible moves available, the fixed time limit, we calculate the optimal solution at the max depth at which we could compute all calculations. When experimenting and presenting data below, we fixed the number of "nodes" considered to be 2.5 million nodes. This corresponded to around a 5 second time limit for each move.

Using this heuristic, we are able to ensure that our bot will compute a move within the given time limit regardless of board size and state. This heuristic essentially lets us take into account the current resources available as well as the real-life constraints on such an AI bot. For example, if this bot was to compete in an actual game where a time limit is placed on each move selection (as is common in most competitive games), this dynamic depth algorithm ensures our bot will meet the time constraints.

4 Implementation

In our implementation of "Dots and Boxes," we modularized the game into five major components:

1. *GameBoard*
2. *GameGUI*
3. *Player*
4. *BotDecisionTree*
5. *BotMinimax*

GameBoard maintains all internal states of the game and exposes a *play* routine that places a move for a player. *GameGUI* encapsulates all the graphical interface components of the game. Lastly, *Player* module encapsulates how a bot player interacts with the game and the different algorithms implement *Player* module. And *BotDecisionTree* and *BotMinimax* are extensions of *Player* module that correspond to different winning bot strategies.

4.1 GameBoard

Represents and maintains all internal states of the game

states - The internal states of *GameBoard* maintains all the necessary information about the game. This includes board width, board height, lines placed on the board, players, squares scored by each player.

play - This is the main function that drives the game forward. Specifically, this function places a particular move on the board after checking for validity. Then returns a list of squares captured by that particular move.

unplay - This is mainly used by Minimax to play out different scenarios of the game to construct the game search tree. It undos whatever action was taken by the above method *play*

4.2 GameGUI

Encapsulates all the graphical compnent of the game using `Tkinter`, which is Python's standard GUI package

_nextturn - This is the main driver of GUI that gives control to the next turn's player. If the next player is a human, it enables click. If the next player is a bot, it makes a move according to bot's decision.

4.3 Player

Specific implementations of our methods(Decision Tree, Minimax, Minimax with Pruning, Decision Tree + Minimax with Pruning) extend this module.

getNextMove - This is the main driver of Player, which returns the next move for this specific instance of Player.

4.4 BotDecisionTree

This module extends *Player* and implements the decision tree outlined in the prior section.

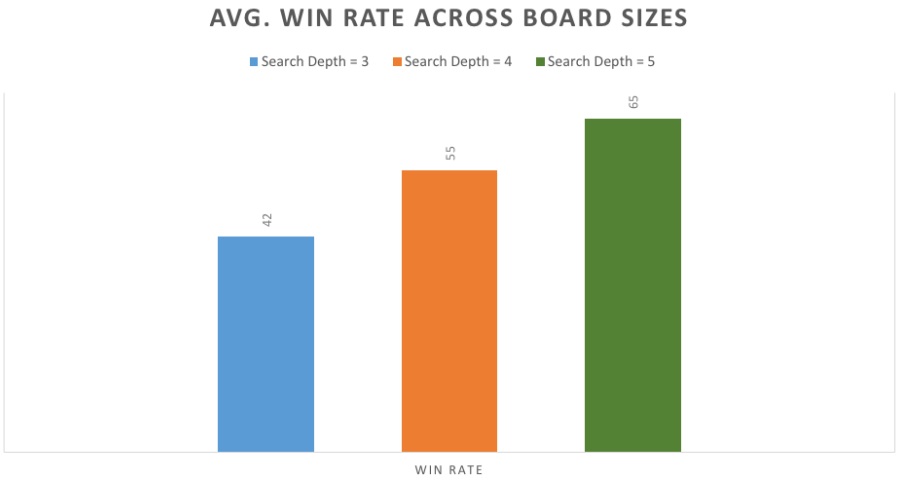
4.5 BotMinimax

This module extends *Player* and implements Minimax, Minimax with Pruning and Decision Tree + Minimax with Pruning approaches.

5 Results and Evaluation

5.1 Winning Rates

Board Size/ Depth	3	4	5
3	0.30	0.30	0.30
4	0.40	0.65	0.75
5	0.25	0.55	0.65
6	0.55	0.55	0.80
7	0.60	0.55	0.75



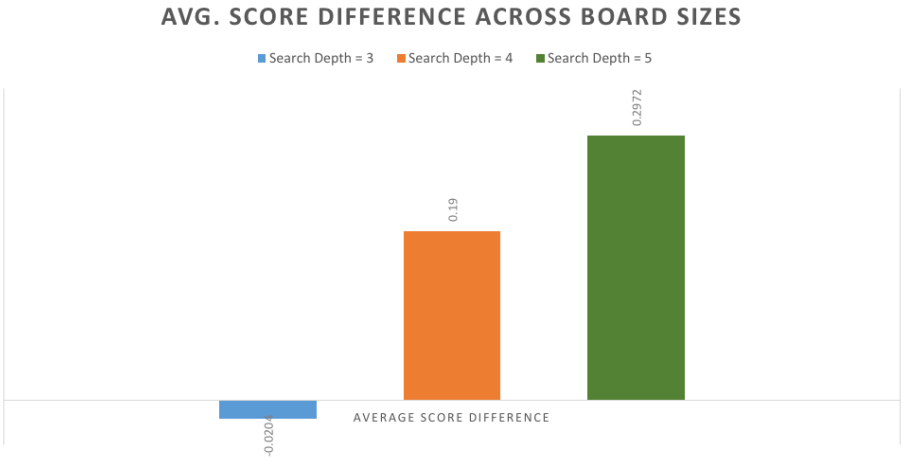
Based on the table above, we can clearly see a general trend of improving scores going across the table from left to right as the depth increases, and a weaker trend of improving scores going from top to bottom as the board size increases. This matches our intuition that the lower in the minimax tree we go down, the better the algorithm performs. Particularly, the depth corresponds to the number of moves we look forward to before choosing our optimal move. Naturally, the more number of moves in the future we take into consideration the better the alogrithm performs.

An interesting point is that for the case of board size (3,3), the minimax algorithm only wins 30% of the time regardless of the depth. This seems to occur because when the board size is small enough as in the case of a (3,3) board, there aren't too many choices available in the number of moves and the decision tree can perform just as well. This leads to the minimax and decision tree bots to draw in the game leading to a low win rate.

5.2 Average Score Difference

The win rate is a good course measure of which bot won each game. However to look at a finer-grained measurement of the performance of the minimax algorithm in comparison to the decision tree, we decided to look at the average score difference between the minimax algorithm and the decision tree algorithm. To be able to compare across board sizes, we normalized the scores by the total number of boxes one could have possible scored in that particular board sized game.

Board Size/ Depth	3	4	5
3	-0.125	0.025	0.100
4	0.166	0.266	0.466
5	-0.156	0.131	0.200
6	-0.112	0.464	0.412
7	0.125	0.064	0.308

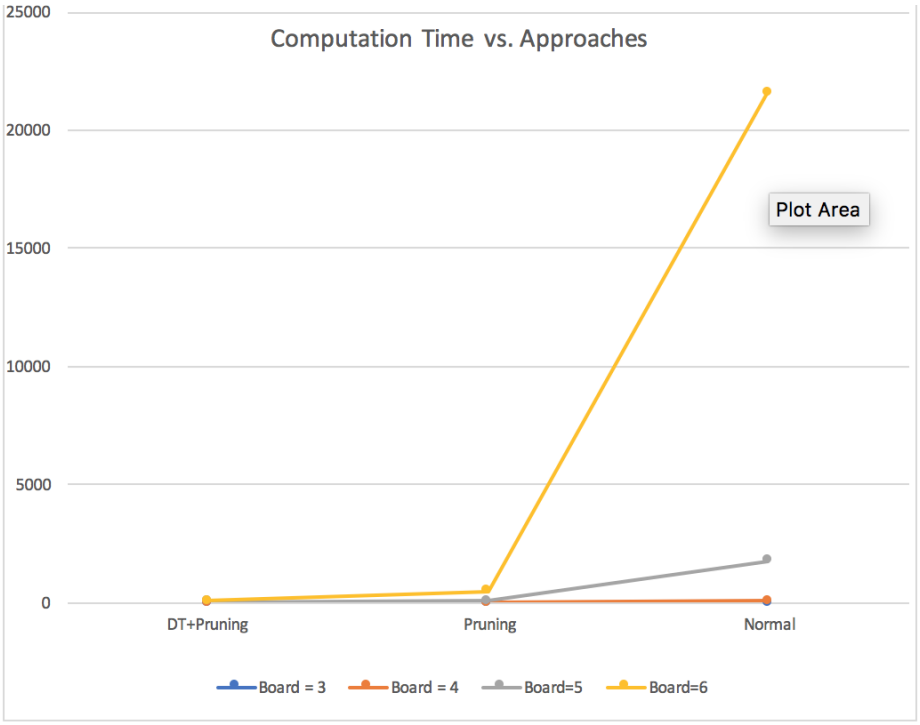


Looking at the average score differential we see a strong positive trend in increasing depth. However, we no longer see the positive trend as we increase board sizes. Particularly, the minimax seems to perform best at board sizes 4 and 6 over the larger board size of 7, while at board size 5 it seems to perform particularly worse. Thus there doesn't seem to be a particular relationship between the ratio of board size to the depth. This actually goes against our original intuition. We hypothesized that as the ratio of board-size:depth increased, the minimax algorithm would perform worse. This is because as the board size increases, the number of moves left till game finishes also increases, meaning that for a fixed depth, the minimax algorithm is covering a smaller portion of the full game future moves.

However, based on the data there doesn't seem to be a clear relationship. There are three possible explanations for this. One is that there indeed is no re-

lationship between board-size and depth. However, this seems unlikely. Rather, a better explanation seems to be either that we didn't run enough trials to converge to the true data, or the decision tree also performs worse at a similar rate, resulting in a overall no net change in performance for either algorithms.

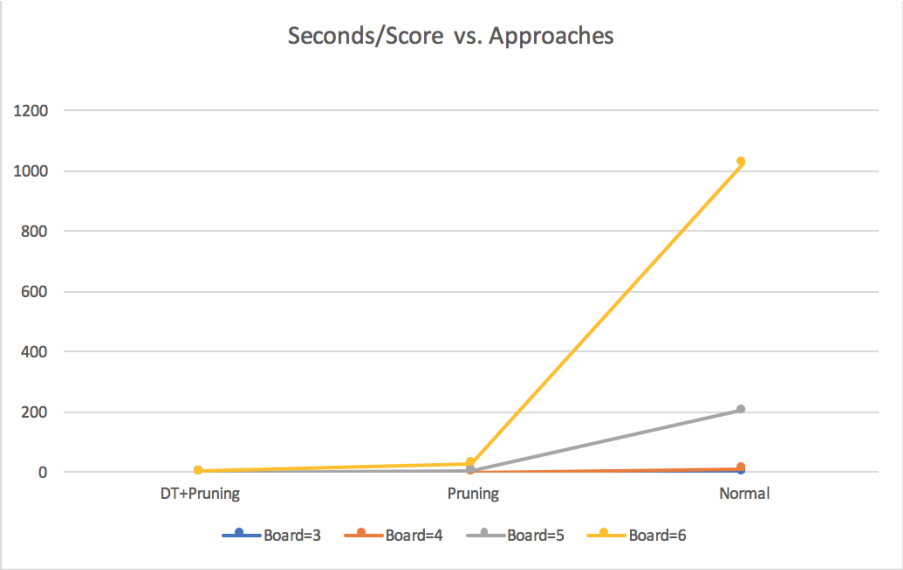
5.3 Time Difference



Above we can see the absolute computation time taken for the three approaches across the four different board sizes: 3, 4, 5, 6. Specifically, the three approaches are: hybrid minimax using decision tree for the first half of the moves and also pruning, minimax with pruning, and finally normal minimax with not pruning. Clearly we can see the difference in computation time for the three approaches. Looking at just the points for normal minimax, you can observe the exponential increase in time across the board sizes. For the only pruning minimax approach, the growth can still be seen but is very small compared to the regular minimax. Finally, the approach using decision tree for the first half and then minimax with pruning seems to grow very little in computation time as the board size increases.

To actually examine the trade-offs between better computation time and loss in performance, we examined the seconds per score measure. Specifically, we took the absolute total time taken by the algorithm and divided by the number of points it scored. This allowed us to compare across the three approaches

how much we gained in terms of time performance and lost in terms of score performance. The graph is below:



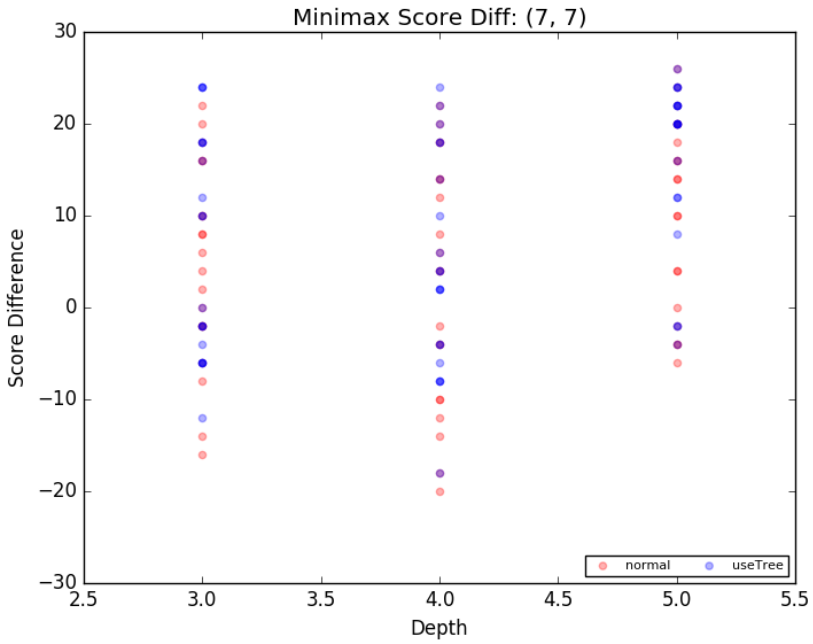
Clearly, the graph trend is essentially the same as the above. This suggests that our gain in time performance came with almost or little loss in actual score performance. Specifically, we can see that the average computation time required to score one point is exponentially smaller for the hybrid minimax approach and minimax with just pruning compared to the regular minimax algorithm.

This shows us two things. First it shows us the power of A-B pruning in the minimax algorithm. Without sacrificing any optimality, our minimax with pruning is able to significantly reduce the computation run time. Second, the graph confirms our hypothesis that using decision tree for the first half of the moves does not result in a significant loss in performance.

5.4 Comparing Hybrid Minimax

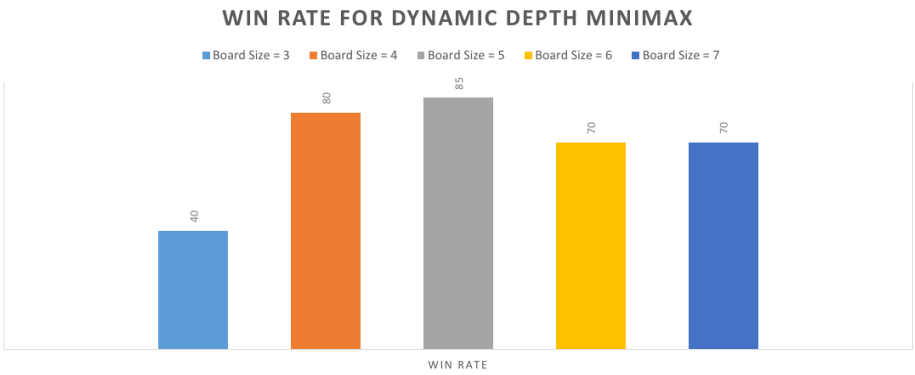
As mentioned above, one of the extensions we performed was implementing a hybrid minimax which uses the decision tree heuristic for the first half of the moves, then relies on regular minimax with pruning for the more critical moves. To actually compare the scoring performance only, rather than the seconds per score measure, we produced a scatter plot of the normalized score difference across all the trials for both the hybrid minimax which uses the "useTree"

heuristic and the normal minimax. Below is the scatter plot.



Observing the graph above, we can see that the blue and red dots are not linearly separable for all three depths. This suggests that there is no general loss in score performance using this hybrid minimax algorithm. Particularly, we saw a great gain in time performance in the graphs above, but this graph once again shows that there is no noticeable loss in score performance when using the hybrid minimax. This further strengthens our hypothesis that the first half of the moves are not critical and using the minimax algorithm for only the latter half of the moves still results in similar performance of the algorithm.

5.5 Comparing Dynamic Depth Minimax



Looking at the graph above, we observed that the dynamic depth minimax algorithm actually outperforms all previous algorithms. Particularly for board sizes over 3, the algorithm performs at almost a win rate of 80%. The intuition behind choosing a dynamic depth was to perform as many calculations as possible within a given time or computation limit. One possible reason for the boost in performance is due to the fact that the most critical moves of the game come late in the game. Particularly, the most critical decisions are made when there are few moves possible. And if there are few moves possible, the branching factor of our minimax is small, resulting in being able to perform minimax at a higher depth level than usual. Thus for example, upon examining the run of minimax on a 8x8 board, we observed that for the first half of the decisions, the minimax only ran to a depth of 2-4. However, for the latter half of the decisions, the more critical decisions, minimax was able to run the algorithm for depths larger than 4. Particularly, for the last 10 moves, we observed the algorithm performing up to depth 9 in the minimax search. Thus, by dynamically determining the depth based on the resources available, we were able to not only keep the minimax algorithm running within a time constraint, but also allow it to maximize its performance in the critical sections of the game.

For an actual application of an AI algorithm in the dots and boxes game, this final heuristic of using a dynamic depth would naturally be most suitable. Using an dynamic depth, the algorithm would be able to stay within the time constraints of the actual game. Furthermore, we see that it has the highest performing win rate amongst the various algorithms we implemented.

6 Conclusion and Moving Forward

Overall, using our experiments we were able to answer the questions we had started out with. Purely based on the scoring performance, we saw that Minimax performs better as the depth increases. This applied for both the Minimax

with pruning, and the hybrid Minimax with decision tree and pruning. Furthermore, we saw that there was no real scoring performance trade-off between the two algorithms. We also saw that there is no real trend in our data between the boardsize and depth ratio which went against our original intuition. This confirmed our hypothesis and heuristic that the first half of the moves made in the Dots and Boxes game are not critical, and simply using a decision tree algorithm suffices. However, we do see that using a dynamic depth minimax algorithm resulted in a much higher performance than the other algorithms across all board sizes. This was due to the ability of the algorithm to go to much higher depth levels in the late stages of the game due to the small number of moves left.

In terms of computational feasibility, we saw that even for a board size of 5 and depth 5, the original minimax algorithm had trouble computing a move within a reasonable amount of time. To address this problem, we tried both Alpha-Beta pruning and using our heuristic of using decision tree for the first half of the moves. Based on the data we learned that Alpha-Beta pruning almost exponentially reduces our run time, and combined with the decision-tree heuristic can help our minimax algorithm compute moves for board sizes up to 11 and depth 5 in a feasible amount of time. We also saw that even with a set time limit, using the dynamic depth algorithm we were able to perform at the highest win rate while staying within the time limit. This suggests that for a practical application of an Dots and Boxes AI, using the dynamic depth algorithm would be most useful, boosting both performance while staying within the time constraints of the game.

Moving forward, we want to explore concepts of Reinforcement Learning applied to Dots and Boxes. Such approach is different from Decision Tree or Minimax algorithms we experimented, because it uses reward feedback for a bot to learn behaviors for winning moves. Additionally, we want to further explore optimizing the computation needed for Minimax by memoizing previsouly solved cases by finding more efficient ways to represent game states. We believe that sections of Dots and Boxes are subproblems that the bot has already seen in its previous encouters, which can be memorized for cutting computation needs in the future.

7 References

1. “Dots and Boxes.” Wikipedia. Wikimedia Foundation, n.d. Web. 11 Dec. 2016
2. “Adversarial Search.” Wikipedia. Wikimedia Foundation, n.d. Web. 11 Dec. 2016
3. “Minimax” Wikipedia. Wikimedia Foundation, n.d. Web. 11 Dec. 2016
4. “Dots and Boxes.” Dots and Boxes. N.p., n.d. Web. 11 Dec. 2016. (https://hkn.eecs.berkeley.edu/ DYOO/python/dots-and-boxes/index.html)
5. Russell & Norvig, `Artificial Intelligence: A Modern Approach` $3^{rd} Edition$