Alpha-Beta Pruning

Carl Felstiner

May 9, 2019

Abstract

This paper serves as an introduction to the ways computers are built to play games. We implement the basic *minimax* algorithm and expand on it by finding ways to reduce the portion of the game tree that must be generated to find the best move. We tested our algorithms on ordinary Tic-Tac-Toe, Hex, and 3-D Tic-Tac-Toe. With our algorithms, we were able to find the best opening move in Tic-Tac-Toe by only generating 0.34% of the nodes in the game tree. We also explored some mathematical features of Hex and provided proofs of them.

1 Introduction

Building computers to play board games has been a focus for mathematicians ever since computers were invented. The first computer to beat a human opponent in chess was built in 1956, and towards the late 1960s, computers were already beating chess players of low-medium skill.[1] Now, it is generally recognized that computers can beat even the most accomplished grandmaster.

Computers build a tree of different possibilities for the game and then work backwards to find the move that will give the computer the best outcome. Although computers can evaluate board positions very quickly, in a game like chess where there are over 10^{120} possible board configurations it is impossible for a computer to search through the entire tree. The challenge for the computer is then to find ways to avoid searching in certain parts of the tree.

Humans also look ahead a certain number of moves when playing a game, but experienced players already know of certain theories and strategies that tell them which parts of the tree to look. However, as computers become faster in terms of raw processing power and humans become better at programming the computers to search more efficiently, even the most skilled humans will be defeated.

In this project, we explore different ways to reduce the number of nodes in the game tree that the computer has to look at in order to find the best possible move. We start with a basic algorithm called *minimax* that searches through the entire tree, then add the following components:

- Alpha-Beta Pruning
- Forced Moves
- Random Searches
- 1/-1 Termination

These algorithms serve as the building blocks of modern game-playing computers. Computers that play games with large trees such as chess typically have evaluation functions that can assess values to non-terminal nodes. This allows them to apply the basic algorithms without reaching the bottom, which is almost impossible with large trees. The evaluation function can even be "learned" by the computer using neural networks [2].

We tested our algorithms on the following three games:

- Ordinary Tic-Tac-Toe
- Hex
- 3-D Tic-Tac-Toe

With these algorithms, the computer was able to find the best possible moves in Tic-Tac-Toe, Hex on boards up to 4x4, and 3-D Tic-Tac-Toe on boards up to 3x3x3.

2 The Minimax Algorithm

We will first explore how the computer can find the best possible move given a game tree. This algorithm is a basic concept in game theory and involves working up from the leaf nodes. Consider the tree shown in Figure 1. The Computer is making a move at the root A and is choosing between nodes B and C. The leaf nodes, in green, are values that correspond to outcomes of the game. The goal

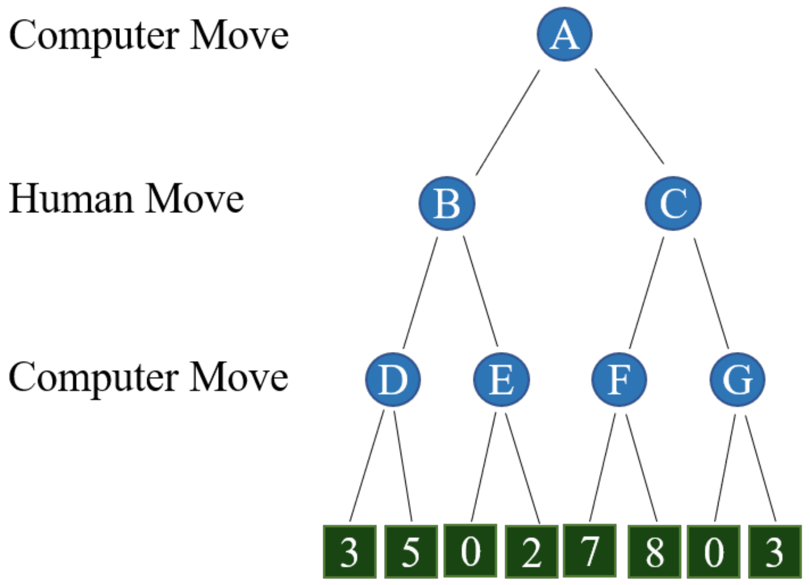
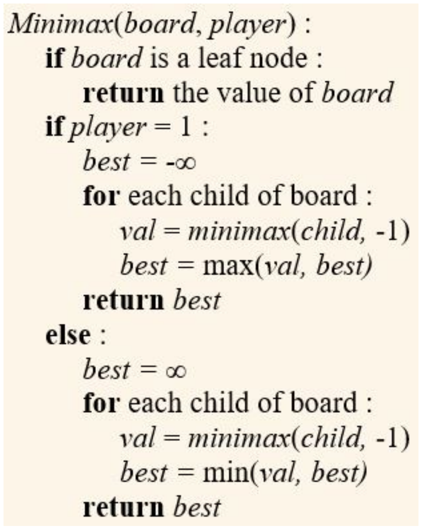


Figure 1: An example game tree. The value of each node is determined by the maximum of the node's immediate children when it is the computer's move, and taking the minimum of the node's immediate children when it is the human's move. In this way, the game is solved from the bottom up.

for the computer is for the game to end in the outcome with the highest possible value, while the goal for the human is the opposite. The *minimax* algorithm is done by asserting a value to each of the nodes, starting from the bottom of the tree. The value of a node when it is the computer's turn (nodes A, D, E, F, and G) is equal to the maximum of the values of the children, while the value of a node at a human turn is equal to the minimum of the values of the children. With this tree, the value of node D is equal to the maximum of 3 and 5, which is 5. Similarly, the values of nodes E, F, and G are 2, 8, and 3, respectively. Working up the tree, the human now chooses between the minimum of nodes D and E (5 and 2), which is 2, so the value of node B is 2. In the same way, the value of node C is 3. Finally, the computer at A should choose the maximum value out of nodes B and C, which is 3 for node C. So the best move for the computer at A is C.



```
Minimax(board, player) :
    if board is a leaf node :
        return the value of board
    if player = 1 :
        best = $-\infty$
        for each child of board :
            val = minimax(child, -1)
            best = max(val, best)
        return best
    else :
        best = $\infty$
        for each child of board :
            val = minimax(child, -1)
            best = min(val, best)
        return best
```

Figure 2: The full *minimax* algorithm in pseudocode. The initial call will be called upon the root, with player equal to 1, corresponding to the computer. The algorithm traverses the entire game tree in post-order.

2.1 The Algorithm

From here on we will denote the computer as player 1 and the human as player -1. The full algorithm in pseudo-code is shown in Figure 2 [3][4].

The reader can verify that this algorithm traverses the game tree in post-order and that it follows the logic described before. The main disadvantage with this procedure is that this algorithm requires the computer to traverse the entire tree, which can require a lot of time and computing power. However, the algorithm does succeed in finding the best possible move for the computer. Another thing to note is that the algorithm assumes that the human will make the best possible move when given the opportunity, but in reality that is not always the case. The algorithm assumes that the human will play to make the outcome for the computer as bad as possible, so if the human is not playing perfectly, the outcome for the computer will be at least as good as if the human was playing perfectly.

2.2 Implications

The *minimax* algorithm leads to an interesting result that will be used later on when exploring the Hex game.

Theorem 1. *In any two-player, finite game with no draws, one player must have a winning strategy.*

Proof. In the minimax algorithm, the entire game tree is searched and the algorithm finds the best possible move by working from the bottom up and assuming both players are playing perfectly. Since the values of all the leaf nodes are either 1s or -1s, corresponding to wins and losses respectively, every parent node is also either a 1 or a -1. Therefore, at the beginning of the game, the root node must have a value of either 1 or -1. If it is a 1, then the first player has a winning strategy, and if it is a -1, the second player has a winning strategy. □

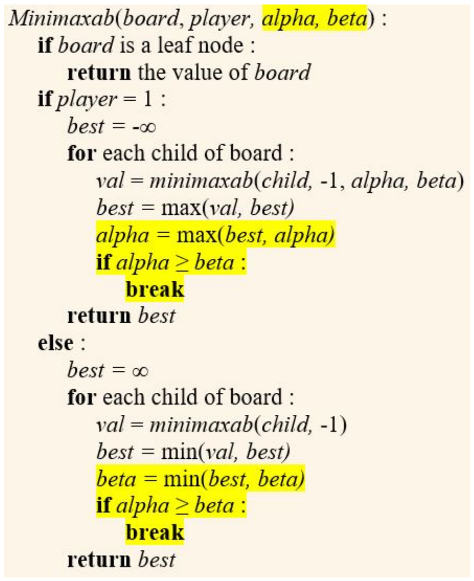
2.3 Alpha-Beta Pruning

We will now expand upon *minimax* to make it so that the computer does not have to look at the entire tree. Consider the gametree of Tic-Tac-Toe displayed in figure 4. Here, the computer is playing X's and the human is O's. The value of the game is 1 for a computer win, -1 for a human win, and 0 for a draw. Note that the obvious move would be for the computer to play in the left-most subtree where it would block the human from getting three in a row along the diagonal. However, ordinary *minimax* would still need to traverse the whole tree to be able to make that decision. It would first take the minimum of the children of B, which is 0, and then compare that to the minimum of C's children, which is -1. It would then find the maximum of the value of B (0) to the value of C (-1). But wait! Since C's left child is -1, and C looks for the minimum of it's children, the value of C is guaranteed to be less than or equal to -1 regardless of the value of its right child. And because -1 is less than 0, C is guaranteed to be a worse move than B just because of C's left child. So the computer does not even need to look at C's other children. This is the basis of what is called alpha-beta pruning.

Consider the following addition to the code in the *minimax* algorithm in Figure 3. In this algorithm, we introduce the new parameters $alpha$ and $beta$:

- $Alpha$ is the maximum value that can be attained at the current level or above.
- $Beta$ is the minimum value that can be attained at the current level or above.

The initial call to *minimaxab* will be *minimaxab*(*board*, *player*, $-\infty, \infty$). With this algorithm, the for-loop is broken whenever beta is less than or equal to alpha, meaning that the computer does have to look at the remaining children. We will



```
Minimaxab(board, player, alpha, beta) :
    if board is a leaf node :
        return the value of board
    if player = 1 :
        best = $-\infty$
        for each child of board :
            val = minimaxab(child, -1, alpha, beta)
            best = max(val, best)
            alpha = max(best, alpha)
            if $alpha \geq beta$ :
                break
        return best
    else :
        best = $\infty$
        for each child of board :
            val = minimaxab(child, -1)
            best = min(val, best)
            beta = min(best, beta)
            if $alpha \geq beta$ :
                break
        return best
```

Figure 3: The full algorithm for *minimaxab* in psuedocode, with the differences from *minimax* highlighted in yellow. The additional parameters *alpha* and *beta* are used to help prune the game tree by breaking the for-loop.

now break down exactly how the algorithm works after the initial call with the Tic-Tac-Toe game tree in Figure 4:

- The traversal starts on node A, where it is the computer's turn, so *player* is equal to 1. *Minimaxab* is then called on A's first child, node B, where *player* is -1.
- *Minimaxab* is then called on E, the first child of node B. We call on E's only child, which will return a 1. E then returns 1 and *val* = 1 at B.
- Now the computer finds min(*val*, *best*), so *best* = min(∞, 1) = 1.
- *beta* = min(*beta*, *best*) = min(∞, 1) = 1. Since *alpha* is still $-\infty$, $beta \nleq alpha$, and the for loop continues.
- The computer looks at the value of B's right child, which is 0, then compares it to *val*, which is 1. Since $0 < 1$, *val* = 0, and *beta* = 0. B is out of children, so *minimaxab* returns 0 to node B.

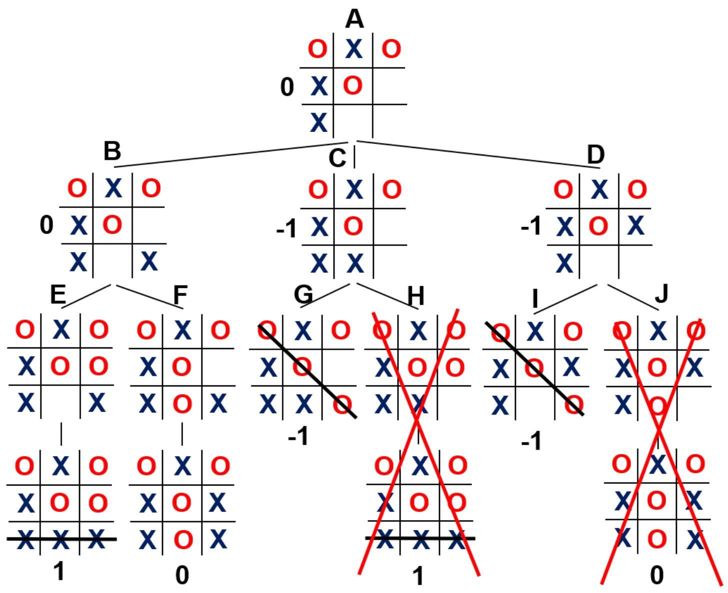


Figure 4: An example Tic-Tac-Toe game tree. With the *minimaxab* algorithm, nodes H and J are pruned. While pruning just two nodes may seem like a small accomplishment, if a node is pruned higher up in the game tree it would be a much bigger feat.

- At node A, player = 1, so best = max(val, best) = max(0,$-\infty$) = 0. Then alpha = max($-\infty$, 0) = 0.

- *Minimaxab* is called on the left child of node C, with *alpha* = 0 and *beta* = ∞, and with *player* = -1. Since at C we are starting a new for-loop, best = ∞, then *val* = -1, *best* = min(*val*, *best*) = -1, and *beta* = min(∞, -1) = -1. Here *beta* $\leq$ *alpha* is true, so the code breaks the for-loop and moves to node D with the same *alpha* = 0.

- At D, *best* is still ∞. We call upon D's left child, where *val* = -1, so *best* = min(*best, val*) = -1, and *beta* = min(*best, val*) = -1. Since *alpha* is still equal to 0, we find that *beta* $\leq$ *alpha*, so for loop is broken and D's right child is ignored.

- Finally, A finds max(0, -1) = 0 and returns 0. A is out of children so we are done.

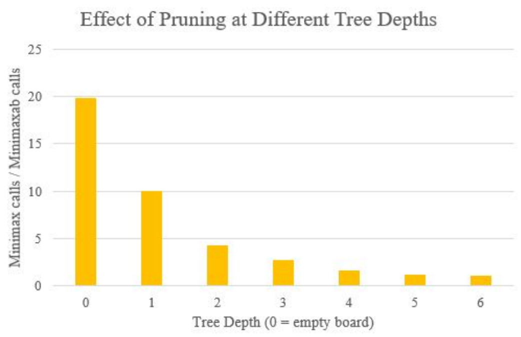
In this algorithm, we were able to ignore C's right child and D's right child. This may seem like a relatively small feat, but with larger trees, being able to prune an entire subtree when near the top is a great accomplishment for the computer. Figure 5 shows the number of board positions the two algorithms look at for the function call at different tree depths. Actual code for the two algorithms will be located in the appendix. Note that pruning has a much greater impact the larger the tree is. One can expect that for games more complicated than Tic-Tac-Toe pruning will be even more crucial to be able to find the best move while conserving computing power.

2.4 Other Improvements

We can further improve the algorithm for Tic-Tac-Toe by adding in more features. These include:

1. Check to see if there is a space that will immediately win the game for the computer and play there.

2. If the game cannot be immediately won, check to see if there is a space that the human will win at the next turn and play there to block the human from playing there.



# Spaces Filled	No Pruning	Pruning	Multiple
0	549945	27662	19.9
1	61004	6112	10.0
2	7637	1779	4.29
3	997	364	2.74
4	180	110	1.63
5	35.2	29.5	1.21
6	16.0	14.5	1.10

Figure 5: A graph showing the number of nodes generated by *minimax* divided by the number of nodes generated in *minimaxab* at different tree depths. The table shows the number of nodes generated by each algorithm. At the beginning of the game when no spaces are filled, pruning improves the search algorithm by almost 20x.

3. If the algorithm finds a move that will lead to a win, stop searching and play there since there are no better outcomes than a win.

4. Randomize the search order in the tree.

Improvements (1-3) are self-explanatory, but (4) requires some further explanation.

In the ordinary *minimax* algorithm, child nodes are evaluated in the same order in every part of the tree using a simple for loop. In Tic-Tac-Toe, each child node is evaluated 1-9, starting with moves in the top left, then working left to right, top to bottom. With randomness turned on, the order in which each of the nine children are evaluated would be random. Of course, in ordinary *minimax* randomness will not help because the entire tree must be created anyway. But with alpha-beta pruning, the order in which the children are evaluated is important because it could lead to more or less pruning depending on whether the computer gets "lucky" in finding a way to short-circuit parts of the tree. In the worst case, both random and non-random algorithms will still need to evaluate the entire tree. But with randomness turned on, the computer has a better chance to get "lucky" by searching through children in a random order at every node. This is because with 1/-1 termination, when the computer finds something, it stops looking. Without randomness, it is more likely for the computer to repeat unsuccessful searches.

With all four of these features turned on, the number of nodes generated in the opening move of a 3x3 Tic-Tac-Toe game was cut to 1,893, down from a total tree size of 549,945 generated with ordinary *minimax*. However, this was assuming that checking to see if there is a "forced move" (items 1 and 2) is done for free and does not count as generating a node. We can still conclude that the runtime was reduced drastically from the original *minimax*.

3 Hex

We will now turn to a different game: Hex. Similar to Tic-Tac-Toe, Hex is a two-player zero-sum game played by sequentially placing tokens on a board. The game board is shown in Figure 7. The object of the game is to connect a chain of tokens from one side of the board to the other. In the figure, the red player is trying to connect red tokens from the bottom right side of the board to the top left, while the blue player is trying to connect blue tokens from the top right of the board to the bottom left. The game can theoretically be played on a board with any size, but traditionally an 11x11 board is used. Some interesting mathematical aspects of Hex are:

1. The game cannot end in a draw.
2. Having an extra piece of your own color on the board cannot hurt you.
3. The player who moves first has a winning strategy.
4. The two "acute corners" that are only connected to two hexes are not winning first moves.

Statement (3) was first proved by John Nash in 1952 and relies upon (1) and (2) as lemmas[5]. We will prove (2), (3) and (4). Statement (1) is beyond the scope of this paper, but the proof can be found in [6]. One can think about the result by imagining that there is a river flowing between the two sides of the board, and the only way to dam the river is to establish a path through it.

Lemma 1. *In the game of Hex, having an extra piece of one's own color on the board is not disadvantageous.*

Proof. Suppose that a player has an extra piece at position a on the board. If a is part of the player's winning strategy, then on the turn when the player would play at position a, the player plays anywhere else on the board. If a is not part of the player's winning strategy, then that square does not matter. □

Theorem 2. *In the game of Hex, the player who moves first has a winning strategy.*

Proof. Suppose instead that the second player has a winning strategy. On turn 1, the first player plays at any random place on the board. The first player then pretends that this piece is not there, effectively making the second player the "first" player. The actual first player then proceeds to use the second player's winning strategy, playing in all of the hexes that the second player was going to play at. By Lemma 1, the first player's first move cannot be disadvantageous, so this strategy prevents the second player from winning. This is a contradiction because we assumed that the second player has a winning strategy. Since the second player does not have a winning strategy, and the game cannot end in a draw, then by Theorem 1 we conclude that the first player has a winning strategy. □

Note that although this proof concludes that the first player has a winning strategy, it does not give any indication as to what that strategy is. The proof that acute corners are not winning opening moves involves a similar strategy-stealing argument.

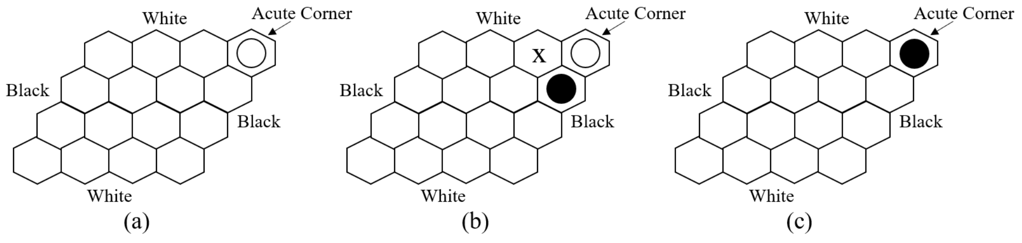


Figure 6: Suppose white has a winning strategy by playing in the acute corner (a). Black then plays as shown in (b). Black then pretends that the white stone is black and that the black stone is not there (c). Black then uses white's winning strategy and win, a contradiction.

Theorem 3. *In Hex, a move in an acute corner on the first turn is not a winning strategy.*

Proof. Suppose that playing in an acute corner on the first move is a wining strategy. White plays in one of them, and black moves as shown in Figure 6b. Black then pretends the black stone is not there, and that the white stone is black. Black then uses white's winning strategy as if black had played in the acute corner on the first move. By Lemma 1, the stone that black pretends is not there cannot be disadvantageous. So the only way for this strategy to fail is if the white stone that black pretends is black is part of the winning strategy. However, for this to happen, the hex labeled x must also be part of the winning strategy. If that is the case, then black wins anyway because of the black stone that black is pretending does not exist. This means that both black and white have a winning strategy, a contradiction, so a first move in an acute corner is not a winning strategy. □

The game has been completely solved in boards up to 7x7, and some openings have been solved (meaning the best move can be found given any board position) in an 8x8 board [2]. We applied *minimaxab* to hex on boards up to 4x4 with the all of the features in the Tic-Tac-Toe program turned on. On a 3x3 board, ordinary *minimax* generated 11107 nodes, while our improved *minimaxab* generated 57 nodes on average (an average is used due to the randomized search). With the 4x4 board, an average of 369,349 nodes were generated with the improved algorithm. Both algorithms assumed that playing in an acute corner on the first move is not a winning strategy. The winning opening moves on boards up to 4x4 are shown in figure 8.

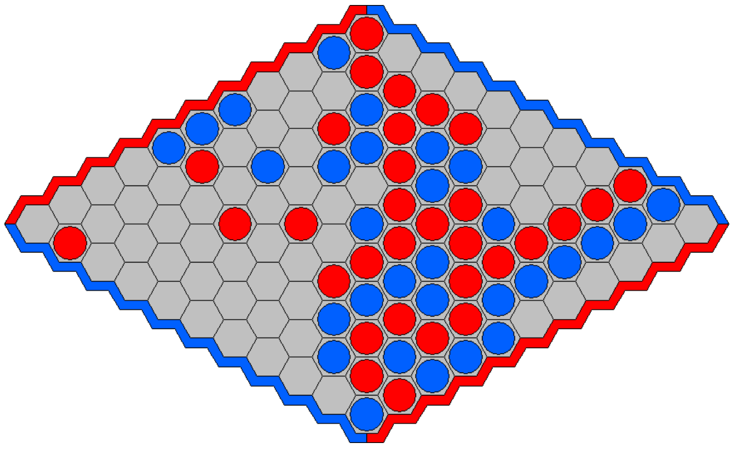


Figure 7: An example hex board. The red player is trying to establish a chain of tokens from the bottom right side to the top left side, while the blue player is trying to connect the other two sides. Red has won in this game.

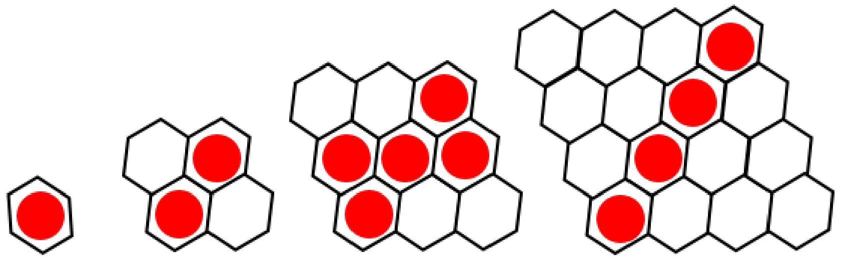


Figure 8: The winning opening moves on boards up to 4x4.

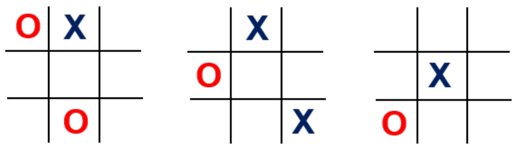


Figure 9: In this game, the O's have won with a three in a row on the left side of the cube.

4 3-D Tic-Tac-Toe

We will now move on to study a third game, three dimensional Tic-Tac-Toe. The rules of the game are similar to 2-D Tic-Tac-Toe in that the goal is to make three in a row. On a computer screen, we can decompose the cube into three 2-D Tic-Tac-Toe boards. Three in a row can be made within each of these three boards, or involving one square in each of the three boards. An example of a winning position is shown in Figure 9.

The game is traditionally played in a 4x4x4 configuration, but we will start with the 3x3x3 case because it was easier for the computer to solve. Using all four additions to the ordinary *minimax* algorithm used in prior games, we found the only winning move on an empty board to be the center by evaluating 38024 board positions on average (an average was computed because of the random search). When searching by looking at the center first, we found this move to result in a win by evaluating only 243 board positions on average. Both of these searches were completed in under one second. Even though 3x3x3 Tic-Tac-Toe has 27 squares and 5x5 hex only has 25 squares, 3x3x3 Tic-Tac-Toe was much easier to solve because there are many more forced moves.

5 Randomization Results

The effects of randomness in Tic-Tac-Toe and Hex are shown in Figure 10. We noticed with forced moves turned on, randomness had an insignificant effect in Tic-Tac-Toe. This is because there are so many forced moves in Tic-Tac-Toe that pruning, does not do much. Without pruning, randomness does not help because there is no possibility of short-circuiting the tree. When the simulation was run without forced move functionality, randomness reduced the number of nodes evaluated by about 6-10% over 1000 trials. It seems that when more spaces are filled in the board, leading to a smaller tree, randomness had less of an effect.

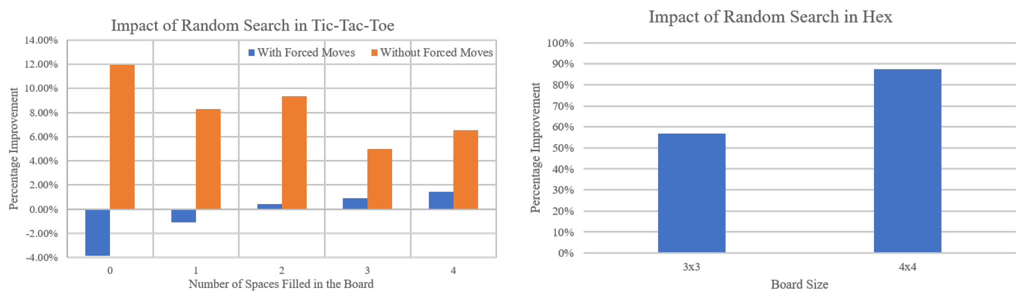


Figure 10: The impacts of randomness for Tic-Tac-Toe and hex. The percentage improvement was calculated as the percentage fewer nodes that were evaluated over 1000 trials.

In three dimensional Tic-Tac-Toe, we found randomness to have an insignificant effect. This was expected because there are even more forced moves in 3-D Tic-Tac-Toe than in 2-D Tic-Tac-Toe due to the higher number of ways to get three in a row.

With Hex, randomness improved the algorithm much more significantly. We found that 58% fewer nodes were evaluated in 3x3 Hex and 88% fewer nodes were evaluated in 4x4 Hex. We expect that in larger boards with larger trees, randomness will have even more of an effect in reducing the runtime.

6 Other Games

The *minimax* algorithm and its improvements are actually not that hard to apply to other finite board games that are similar to Tic-Tac-Toe and Hex. The only functions that must be written are those that check the board for win conditions. Besides that, only minor modifications must be made to the code to account for the different board size and rules. Connect Four is a good example of another game that could be explored using our algorithms in the future. We could also expand on our code by writing an evaluation function that can asses values to non-terminal nodes. Then, we could modify *minimax* so that if a node is in a non-terminal state a a certain depth, the evaluation function is run on it and that node is treated as a leaf with that value. A more sophisticated program would also try to determine the “volatility” of different board positions and search deeper in parts of the game tree where moves can change the state of the game more significantly.

7 Acknowledgments

Special thanks to Professor David Guichard for supervising the project and Taka Olds for peer-editing.

References

[1] Bill Wall. Early computer chess programs. `https://archive.is/20120721202324/http://www.chessville.com/BillWall/EarlyComputerChessPrograms.htm`, Jul 2012.

[2] Broderick Arneson, Ryan B. Hayward, and Philip Henderson. Solving hex: Beyond humans. In H. Jaap van den Herik, Hiroyuki Iida, and Aske Plaat, editors, *Computers and Games*, pages 1–10, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[3] Akshay L Aradhya. Minimax algorithm in game theory — set 4 (alpha-beta pruning). `https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/`, Dec 2018.

[4] Ellis Horowitz and Sartaj Sahni. *Fundamentals of data structures*. Sung Kung Computer Book Co., 1987.

[5] John F Nash. Some games and machines for playing them. Rand Corporation, 1952.

[6] Jing Li. Hex notes. MIT, Apr 2011.