Final Report for Team 16 Project

Cassidy Crouse (1), Candace Campbell (2), Matthew Ferry (3), and Chidi Ede (4)
Department of Computer Science and Electrical Engineering
University of Maryland, Baltimore County
Email: (1) pm26690@umbc.edu, (2) candcam1@umbc.edu,
(3) mferry1@umbc.edu, and (4) chidede1@umbc.edu

***Abstract*—After further research into solving Connect Four using a minimax algorithm, there are several options and optimizations available for varying degrees of precision in our AIs decision making process. It will be shown that applying minimax to any given Connect Four board requires a node traversal that grows exponentially with respect to the iteration depth of the future game state. With this in mind, it will also be shown that for any generally unoptimized AI that would like to solve Connect Four with minimax, there are far too many computations that must be done in order to generate a look-up table that could be used to play a perfect game of Connect Four from the beginning to end in a reasonable amount of time. It is our belief that a reasonable approach to the problem of computational time required for a perfect solver is to instead make an imperfect solver with a utility function and then fine tune the weights and attributes of this utility function until we have a product that we believe can mimic the ability of a human to play Connect Four.**

I. Findings on the Nature of Connect Four

The specifics of Connect Four were given in our first report, but we would like to describe the game more in terms of game theory. The existence of a perfect solution given a certain game state (an empty board) will also be explored as well as approximations into the number of possible game states and other interesting statistics. We have also included more formal definitions of the algorithms and structures that we may use in the implementation of a Connect Four solver.

A. Connect Four and Game Theory

Connect Four is a game in which two players alternate in placing disks in one of seven columns of a game board that is seven columns wide by six rows high in such a way that they are able to line up four consecutive disks either horizontally, vertically, or diagonally. A column with six disks inside of it is a full column, and players cannot place their disks in such a column. Players must place a disk when it is their turn unless the game board is full. If the game board is full and no player has lined up four disks consecutively in any winning configuration, the game is a draw.

Because their is no apparent score, and any given player either wins, loses, or draws Connect Four is a fully-competitive game [2]. As the decisions of both players are visible to each other it is also said that Connect Four is a game of perfect information [2]. With both of these qualities in mind, Connect Four is a zero-sum game because any one players ”score” is the opposite of their opponent's [2]. For example, if the ”score” was how many turns it would take you to achieve four consecutive disks given the other player played a perfect game, then their score would be the negative of your score.

Because it is a zero-sum game with many possible permutations of game states, we have chosen to apply a, more or less, brute force minimax algorithm to solving Connect Four from a given game state onwards. This algorithm would look ahead as far as possible into the moves that could be made by both players, and choose the one that maximizes the ”score” of the AI player, thus minimizing the ”score” of the other player.

In order to maximize a ”score”, there must be some notion of what this ”score” actually is, we have not yet decided a way to measure this score but the value itself is certainly related to how many turns it would take the AI to force a win (if the other player plays perfectly) and the total number of turns that have already passed. If the AI finds that its maximum ”score” for all possible moves is zero then it has found that if the other player also plays perfectly then it is guaranteed to fill the board and draw. If any given move could not possibly lead to a win if the other player plays perfectly, the ”score” is then negative and the AI is likely to lose. Choosing to maximize a negative score will ensure that the AI loses in as many turns as possible.

B. A Perfect Solution

If the Connect Four game board is empty and you are the first player, victory is always possible. In other words, if the first player makes the right decision on every turn of play from beginning to end, they are capable of winning the game every single time regardless of the decisions of the second player. The most interesting subset of these winning games is that in which the second player also plays a perfect game (which is to say again, that they make all the right decisions on every turn), and loses on the second to last (41st) turn of play [1]. These perfect games end in such a way that there is only one column left for both players to play their disks inside of, and every instance includes a forced loss by player two as they must give player one the win.

C. Computing the Perfect Solution

Our chosen algorithm, minimax, is entirely capable of producing a perfect solution for the first player at every turn of play given we allow it to run for as long as it requires. The

problem with this is that it will require a length of time that is unreasonable for our purposes. The total number of valid Connect Four game permutations is over 4.5 trillion [1] for a game where the first player has yet to make their first move. Our minimax algorithm would have to sift through all possible permutations in order to come to the conclusion that it can win a perfect game on the second to last turn assuming that the second player will also play a perfect game, as our minimax will.

This implementation of a brute force minimax is not ideal and we have decided to optimize it in any way that we can. Although we have yet to implement minimax in our vision, the most obvious way to optimize the traversal of all of the possible permutations is to use an Alpha-beta method of trimming off depth iterations that are either unrealistic or inherently flawed. If the AI is aware, as it should be, that it will be guaranteed to be able to win if it is the first player and move has yet been made then it should be able to ignore permutations where victory is impossible or defeat is certain unless it finds that victory is in fact impossible or defeat is in fact certain.

There are also other important pieces of information such as the fact that a perfect game (a game where both players make all of the most ideal decisions when it is their turn) can only be played if the first player makes their first move in the middle column of the board [1]. This fact alone trims the number of possible permutations roughly by a factor of seven. Our overall goal for this Connect Four AI will be that if we pit two versions of it against each other, it should play a perfect game that ends with the first player winning on the second to last turn on as many instances as possible. In our case, we may have to trade precision for computation time by either reducing the lookahead depth of our minimax tree structure or developing some other optimization or set of optimizations to the minimax algorithm.

II. Proposed Implementation and Preliminary Results

A. Structure of a Connect Four Game Permutation

As per the rules of Connect Four, at any given point in time a player has the choice of placing a disk in one of up to seven valid columns. At the beginning of the game, all seven columns are available, and the total number of possible game permutations is at its maximum of 4.5 trillion. Upon deciding one of the seven columns, the game has advanced to the depth below, cutting the number of permutations by a factor of roughly seven. As columns can eventually be filled, and not all placements of disks are valid as the game ends when four disks are lined up consecutively in a winning configuration, the game of Connect Four can be thought of as a multiway tree [3] of height 42, with all of the leaves (the bottom nodes of the tree) representing either a victory, loss, or draw in the case of the very bottom leaves at depth 42. Although the game of Connect Four is a multiway tree, the number of leaves at a particular child node is between zero and seven.

Minimax must traverse this structure of Connect Four in such a way that it would return to the AI either the certainty of a perfect move, or at the least it's best approximation up to a certain depth. Although we know that the true number of permutations is just over 4.5 trillion, it is possible to think of Connect Four as being bounded by 7^n where $n \leq 42$ is the number of turns that have passed. For example, up to the sixth turn of play no columns may be full and no wins are possible, so the number of permutations is bounded by $7x7...x7 = 7^6 = 117,649$ where the true number of permutations is 22100 according to Numberphile [1]. This is calculated assuming that at every turn up to seven columns can possibly be chosen, and the next turn up to seven more can be chosen, and so on. An implementation of minimax with an exponential upperbound of $O(n) = 7^n$ is not ideal or even computationally possible at a depth of 42, nor is it practical when examining the true upperbound of 4.5 trillion computed permutations.

B. More on Minimax

This algorithm originates from a theorem known as the minimax theorem which was proved by John von Neumann, an accomplished mathematician, physicist and computer scientist, in 1928. his theorem first introduced the idea that in any given two player, zero-sum game with perfect information, there are strategies for each player that allows them to minimize their maximum losses. It also showed that both players minimaxes are equal in absolute value and opposite in sign. he original proof was underdeveloped and did not include any equations to be solved. Sixteen years later, in 1944, von Neumann provided a completely new proof for this theorem where he had a full understanding of the mathematical context. Once this theorem was established, it largely influenced the development of game theory which soon after became an active field of research.

The Mini-Max Algorithm was first created in 1958 for use in computer chess games as part of a larger research interest in Artificial Intelligence. This AI created a tree detailing all possible moves by using heuristics to determine what moves were legal. The tree was pruned using an Alpha-Beta algorithm to keep complexity down and reduce the number of moves needed to explore.[7]

In order to properly define a the mini-max algorithm, we must define two other terms first. A *Finite Game* is a game which is guaranteed to finish. There can be no potential endless games. Every move makes the game move one step closer to completion. A *Zero-Sum Game* is a game in which the payoffs for all players (determined by the result of the game) sum up to zero. This means that after adding up every potential outcome of the game, where every outcome is assigned a value based on its' utility, the end result is zero.[6]

The Min-Max Theorem is derived from these concepts. For every finite two-person zero-sum game there exists at least one optimal mixed strategy. Therefore, there exists a game value v, such that by applying the optimal strategy the first player guarantees for himself a payoff not worse than v, while the

second player guarantees for himself a payoff not worse than $--v$.

The minimax algorithm is designed to find the optimal strategy for the first player, and to decide the best (first) move. According to the authors of *Artificial Intelligence: a Modern Approach* a formal five step application of minimax does the following [5]:

- Generate the whole game tree all the way down to the terminal states
- Apply the utility function to all of the terminal states to get its value
- Use the utility of all of the terminal states to determine the utility of the nodes one level higher. The utility of nodes not belonging to the first player is generated assuming that the second player will make the right move.
- Continue backing up utility values for the leaf nodes towards the root one layer at a time.
- Eventually the values have reached the root, and player one will choose the value that has the greatest utility.

Our implementation of this may include the use of the multi-way tree discussed in the last section, and a recursive traversal of the leaf nodes. Where computation limits the effectiveness of our algorithm, the depth of the traversal may be altered to decrease the amount of computations required while still giving an approximation of the next best move. [4]

C. More on Alpha-Beta Algorithm

Many different programmers independently discovered the Alpha-Beta algorithm. This algorithm aims to reduce the number of comparisons needed when computing the optimal move for a player.

The alpha a represents the guaranteed payoff for the maximizing player, as the computer has determined so far. The computer will continue to update this value as it traverses the tree. The beta b represents the guaranteed payoff for the minimizing player, as the computer has determined so far.

The algorithm starts the tree traversal with the $a = -infinity$ and $b = infinity$. As the algorithm descends the tree, the program updates the values of both for each node. The program then checks that $a > b$. If this is the case, the program will ignore descending down that subtree, as it will never need to be computed since the computer has already discovered a more optimal move than anything that that particular subtree could provide. As a result, the program can bypass many subtrees and save computational time.[6]

D. Graphical User Interface

Just as any game, Connect Four requires a board that visualizes the moves that are being made by both the computer and the player. For the user interface, we are using java Swing and AWT(Abstract Windowing Toolkit). In order to create the game board, we are using a two- dimensional array that consists of 42 JButtons which are formatted to accommodate a board that consists of six rows and seven columns. These JButtons that represent the game board do not have action listeners meaning that when they are clicked on there is no response. This is because in a physical game of Connect Four the disks are placed in an opening atop the game board and fall into the next available position.

Above the board is an additional row of seven JButtons aligned with the existing columns. These buttons are to be used by the player when placing their disks in the desired position. When the player selects one of these buttons, their piece will be placed in the corresponding column. In order to indicate that their piece has been placed, the background color of that position is then changed to be the color of the corresponding player. If the column in which a disk is being placed in is full, there is a message which indicates that that is an invalid move.

Once the player makes their move, control of the board is passed to the AI, which utilizes the implemented algorithms to determine the next move that should be made. When the AI has determined its move, an integer is returned to select which column to place a disk in, the game board is updated based on the value returned and the control of the game is passed back to the player. This process will continue until either there is a winner or every position has been occupied, resulting in a draw. Once the game terminates a new window with text appears stating the game results.

III. Actual Implementation of the Algorithms

A. The Call to the AI

The User Interface makes a call to the AI with a function called 'AI Loop' immediately after the player places their piece. This function call takes the current state of the board as a parameter and returns the column that AI shall place its' next piece. The user interface handles the AI's move, then waits for the next move from the player.

The AI is actually implemented as the aforementioned min-max tree, and all computations and computer logic is handled in this class.

B. Implementation of the Tree

The 'Tree' class was designed to take in the current game board and use it as the root of the tree. It makes a recursive function call to create the entirety of the tree from this root. From the root, it creates all next possible moves that the AI can place. From these newly created roots, it makes every possible move the player can make. It will continue this pattern of recursion until it reaches the specified depth.

To help reduce computational time, every created node is checked to be a win condition before creating subtrees for it. If a created node represents a win condition, creating subtrees from that node is unnecessary since the game will end if it ever reaches that game state.

Once the tree is fully created, the AI will weigh all subtrees in a post-order traversal. This means the AI will weigh all subtrees of a node in the tree before weighing the parent. The node will receive an initial positive weight if represents a win of the AI. Similarly, if the node represents a loss for the AI, the node is set to an initial negative weight.

A parent node, because of the way we created the tree, must either be a terminal state have children. If the node has children, we automatically know the weight of the tree must be determined by its' children. Depending on whose turn it is, the AI will either pick the highest value of its' children to represent its' weight if it represents the AI's move and the AI want to maximize its' utility, or it will pick the lowest weight of the children to represent the node since the node represents the players move and the player is assumed to pick the next move with the most utility to him/herself.

The AI will also keep track of what move the entity made to reach the child's game state so that when it gets to the root, it knows exactly what route the entire game will follow given perfect play from both entities. The value contained in the root is the value passed back to the User Interface for the next move of the AI.

IV. Final Results

A. Notion of an "Easy" AI

Applying the usage of our recursive decent tree into the gamespace of all Connect Four moves available up to a certain depth, we are able to find the terminal states up to a depth of about 7 reasonably. Once the terminal states are found, the only value assigned to them is either positive or negative infinity for a win or loss for the given player respectively. If the AI does not have a clear Win, Loss, or draw before it, it simply plays a random valid column on the board. The completion of this Easy AI builds a foundation for implementing more computationally difficult, but hopefully more effective, methods of evaluating not only the terminal states but also intermediate nodes along the way.

B. Notion of a Utility-based AI

As previously stated it is common for minimax implementations that are not strong solvers of any type of zero-sum game to form some strategy or way of evaluating the game state permutations before them. This utility-based approach requires that a utility function be created and implemented that will give the AI some ability to evaluate a node that is not considered terminal (hereby referred to as an intermediate node). Terminal nodes are reached when the minimax function sees that we have found our target depth, another clear terminal state such as a win, loss, or draw has been found. because our "easy" AI implementation is not capable of evaluating these intermediate nodes, it is our belief that if we added some utility to these nodes it will allow minimax to far better evaluate the nodes that it sees. in our case, we propose the following attributes be considered for the utility-based AI model:

- The number of potential four-in-a-row connections, hereby referred to as "disjoint sets" (even in the case where a set is, in fact, not disjoint).
- The number of columns controlled by a given player where a win is forced if their opponent should play it, hereby referred to as "fatal" columns.

The proposed calculation of the utility value or score of a game board should be done within the minimax function for every node it comes across, and just before minimax alters max and min.

C. More on Disjoint Sets

Disjoint sets must be found before calculating the score of a game permutation to keep scores up to date. We have found there to be 69 total unique four stripe locations in which a disjoint set can be located, 24 horizontally, 24 diagonally, and 21 vertically. A more formal definition of a possible disjint set for our implementation is any unique four stripe on the Connect Four board. Suffice it to say (without an image of such a unique position) that any position in which a win can be located is also a location where a disjoint set can be located, and all such unique locations are of interest in counting the number of disjoint sets (see our presentation for more in depth imagery).

The method by which we count disjoint sets is essentially a 2-Dimensional search of the entire Connect Four board. This search should run in $O(n^2)$ time (For the pseudocode and more in depth imagery, see our power point presentation).

- For all of the locations on the board, see if the location at board[i][j] could contain a unique four disjoint set (two for loops $O(n^2)$)
- At each possible unique location, check in all directions to see if there is, in fact, an interesting disjoint set in that direction (implemented with if statements $O(1)$)
- If there is an interesting disjoint set, which is to say there is a set of the form {O, O, -, O} increment the counter for that particular set and continue counting (where 'O' is the designation of the player interested in counting the number of disjoint sets).

For example, {O, O, O, -} is a disjoint "threes" set, {O, -, -, O} is a disjoint "twos" set, {-, O, -, -} is a disjoint "ones" set, but the set {O, -, -, X} (although some part of it may be a subset of another unique set) is not interesting by itself as the opposing player has a piece here as well. This set is not counted among our interesting sets whatsoever, as it cannot be used to make any four-in-a-row connections in the future.

D. More on Fatal Columns

Fatal Columns must also be found before calculating the score of a game permutation, but can be found much more simply than disjoint sets. The method that we use to find fatal columns is simply playing a column twice alternating players, and seeing if the second player has won. This is done with a linear search of the top row of the board, and placing valid pieces to see if a win has occured. If a player controls a fatal column that is very good for that player as it constrains the movements of their opponent. It can be shown that a fatal column does not actually come into usage until many dozens of turns in the future, so it is an attribute that we think may be valuable for an AI that cannot strongly solve connect four (see our presentation for the usage of a fatal column in defeating one's opponent).

E. The Utility Function

Now that there is some notion as to what the attributes use to evaluate a given board will be, the proposed method of calculating the score of the board is

- $score(board, player) = player.num_o nes * w\_ones + player.num\_twos * w\_twos + player.num\_threes * wthrees + player.num\_fatals * w\_fatal$

F. Performance and Statistics

Below are the results of running both the Easy AI at varying levels of lookahead depth and values of games played. Notice that the Utility- based AI takes far, far longer to make decisions due to the increased level of computation added by computing the score at each node as opposed to only terminal states.

TABLE I
EASY AI: 10000 GAMES, LOOKAHEAD = 5

Type	Value
Total Run Time	8:53
Avg. Time per Game	.053 s
Avg. Time per Turn	1.287 ms
Player 1 Win Rate:	.3263
Player 2 Win Rate:	.6007
Draw Rate:	.073

TABLE II
EASY AI: 1000 GAMES, LOOKAHEAD = 7

Type	Value
Total Run Time	39:22
Avg. Time per Game	2.363 s
Avg. Time per Turn	78.051 ms
Player 1 Win Rate:	.287
Player 2 Win Rate:	.629
Draw Rate:	.084

TABLE III
UTILITY-BASED AI: 1000 GAMES, LOOKAHEAD = 5

Type	Value
Total Run Time	6:56
Avg. Time per Game	.416 s
Avg. Time per Turn	18.039 ms
Avg. Search Depth	4587.38
Player 1 Win Rate:	.632
Player 2 Win Rate:	.317
Draw Rate:	.051

TABLE IV
UTILITY-BASED AI: 100 GAMES, LOOKAHEAD = 7

Type	Value
Total Run Time	21:05
Avg. Time per Game	12.65 s
Avg. Time per Turn	580.54 ms
Avg. Search Depth	155,534.44
Player 1 Win Rate:	.63
Player 2 Win Rate:	.28
Draw Rate:	.09

V. FURTHER IMPROVEMENTS

A. Attempts with a Data Set

The original design of the tree included a reference to a data set. This data set contained every game state in which perfect play from an entity could force the other entity into a win. The plan was to reference every game state in the created tree against this data set. This way, the AI could figure out which game states were better than others without having to fully traverse the depth of tree. There are some issues that make this approach unfeasible at this stage of development.

The data set contained an array of almost 68,000 game states. Originally, we stored these in a single array and searched through the array linearly to determine if a given game state was in this array and what terminal state it led to if so. This proved to be an issue, as it added an additional 68,000 comparisons to each node in the tree, meaning our tree could only work up to a depth of 5 before the computer would take too long to generate a result. A depth of 5 was tested and determined to be not enough look-ahead ability of the AI to be good at winning games.

The array could not be sorted easily, so our only option for cutting down this amount of comparisons was to store the data set in a hash table to reduce search time to O(1). Unfortunately, with our time constraints, this was judged to be too much to implement before the project deadline.

Additionally, it did not solve the most important requirement of the AI; to attempt to win the game. The AI was capable of getting to a winning game state as determined by the data set if it was 5 turns or less in the future. Once it was there, the AI did not know how to finish the game, especially if the forced win was more than 5 turns in advanced, meaning that it would not be able to see how it could win from the winning game state and would pick its' move randomly instead, thus ruining the winning game state.

This addition to the program is left as a possible improvement and has potential to drastically improve performance if implemented correctly.

B. Optimization of the Weights

although we have not gotten this far in the project, we still have decided to put forward some method as to how to optimize the weights. It seems that the best way to optimize the weights of the proposed attributes is to set them to some set of values such that

- $w\_ones < w\_twos < w\_threes < w\_fatals$

and then have the Utility-based AI play itself or the easy AI and observe changes in its ability to win (including how quickly it is able to win) and make minute adjustments independtly for each attribute.

VI. CONCLUSION

As previously stated the overall goal of the Connect Four solver(s) was to first implement a minimax solution up to a certain depth, and then incrementally optimize our algorithm with alpha-beta trimming and whatever other set of optimizations

we deem fit. The ideal Connect Four solver is able to generate the dominant nash equilibrium for the first player that leads to a game that lasts 41 turns, and ends with the second players forced defeat on their last turn of play. Our implementation should approach as close to this ideal as possible in such a way that does not sacrifice our understanding of the process by which we are generating our results.

In the end, we were able to implement our idea of an "Easy" AI that operated without the usage of a utility function and made random decions when it could not decisively win. We then attempted to expand upon this notion of an Easy AI by implementing the same algorithm with a utility or scoring function that would examine attributes of the Connect Four board and assign score values to intermediate nodes as well as the terminal nodes as it did before. In concept, this would have improved the ability of our AI to mimic the behaviour of a human, but logic errors and lack of creating good way to optimize the weights of our attributes got in the way of finishing what could be called a complete Utility-based AI.

REFERENCES

[1] Haran, Brady. Connect Four - Numberphile. *YouTube*, Numberphile, 1 Dec. 2013, www.youtube.com/watch?v=yDWPi1pZ0Po.

[2] Resnik, Michael D. GAME THEORY. *Choices: An Introduction to Decision Theory*, NED - New edition ed., University of Minnesota Press, 1987, pp. 121176. *JSTOR*, www.jstor.org/stable/10.5749/j.ctttshgd.9.

[3] Black, Paul E. "multiway tree", *Dictionary of Algorithms and Data Structures*, Vreda Pieterse and Paul E. Black, eds. 27 October 2005, xlinux.nist.gov/dads/HTML/multiwaytree.html.

[4] Rajiv Bakulesh Shah, "minimax", *Dictionary of Algorithms and Data Structures*, Vreda Pieterse and Paul E. Black, eds. 10 January 2007, xlinux.nist.gov/dads/HTML/minimax.html.

[5] Russell, Stuart J., and Peter Norvig. *Artificial intelligence : a modern approach*. Englewood Cliffs, N.J: Prentice Hall, 1995. Print.

[6] Klesk, Przemyslaw, *MIN-MAX algorithm history, variants, refinements* LaTeX, Department of Methods of Artificial Intelligence and Applied Mathematics

[7] *The Minimax Algorithm and Alpha-Beta Pruning*, Mastering The Game, A History Of Computer Chess

REFERENCE REVIEWS

Numberphile

Although an informal source, the calculations provided by the youtube video posted under the name "Connect Four - Numberphile" are instrumental in providing insight into the number of possible permutations that would be explored by minimax at the beginning of any given game. As the total number of valid permutations is computed by making every possible move at every possible turn, it is essentially an application of minimax on the first turn of play and represents the total number of iterations of the depth tree. The concept of a perfect game in which both players make ideal moves and end with the first player's victory on the 41st turn of play also provides some optimum goal for our AI to achieve (whether or not we are able to do so).

Choices: An Introduction to Decision Theory

A more formal source that explores all manner of two-person competitive and non-competetive games and provides proofs for the completeness of a (mixed strategy) solution for a two-person competitive zero-sum game such as tic-tac-toe. Used to provide a formal definition of Connect Four as a fully-competitive zero-sum game in which the ideal utility function of the first player on their turn results in the negative ideal of the same utility function of the second player on their turn. The author also provides a formal definition of the minimax theorem with respect to the utility matrix of a given game.

Multiway Tree

A standard definition of a multiway tree structure that may be followed by our minimax algorithm, the tree would be traversed with a depth-first recursive search. The root of the tree represents the maximum score for a given player, the children of this node (bounded by 0-7) represent the possible moves for the given player on the next turn, the children of these nodes (bounded by 0-49 total children for all previous nodes) represent the moves of the opposite player.

Minimax

A standard definition of a minimax algorithm that may be used by our Connect Four solver, the minimax algorithm would perform a recursive traversal of our multiway tree that stores the game state of all possible upcoming moves. The minimax implementation will assume all moves made are perfect, and calculate its maximum score for the given player at the given depth based on this fact.

Artificial Intelligence: A Modern Approach

Although somewhat dated in its calculation of the amount of time it would take for algorithms such as minimax and alpha-beta to complete down to their terminal states, the authors still provide relevant definitions and examples of the algorithms in action. A figure very similar to our application of minimax to the problem of Connect Four is presented for the similar game, Tic-Tac-Toe. This figure demonstrates the terminal states with utility values, and also the backwards recursion to the root node ultimately deciding MAX's game decision.

Min-Max algorithm history, variants, refinements

A formal power-point presentation from a university that discusses game theory. In particular, it heavily details the Min-Max Formula and the Alpha-Bets Algorithm, giving many needed formal definitions. It also includes complexity estimates and mathematical formulas for every algorithm.

The Minimax Algorithm and Alpha-Beta Pruning

The alpha-beta pruning is a specific case of the minimax algorithm that reduces the search time of the multiway tree. Each node in the tree would have two scores, alpha and beta, that would represent the value of the move. Alpha represents a value associated with a move favorited by the A.I. while beta represents a value associated with a move favorited by the user and therefore dangerous for the A.I. A move that is more closely associated with a win state for the A.I. would have a high alpha value and a low beta value. A move that is more closely associated with a win state for the user would have the opposite. As you add children to the tree that include

different permutations, these values would be updated based on whether there is a win state or not. A subtree containing a win state would have a high alpha value in the parent node, it would also increase depending on how quickly the win state can be achieved. When looking at a parent node's children, you would just have to find the child with the highest alpha value in order to determine which subtree contains the optimal move. This reduces the search time finding the optimal move in the entire tree, seeing that you wouldn't have to check every single node in the entire tree, just one subtree.