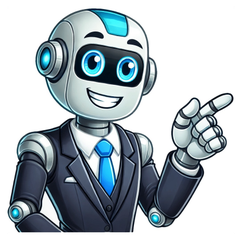
I'm not a bot



Share — copy and redistribute the material in any medium or format for any purpose, even commercially. Adapt — remix, transform, and build upon the material for any purpose, even commercially. The licensor cannot revoke these freedoms as long as you follow the license terms. Attribution — You must give appropriate credit , provide a link to the license, and indicate if changes were made . You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use. ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits. You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation . No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material. Problem : Compute the minimax value of each node in the tree below. Squares represent max nodes and circles represent min nodes. Top-to-bottom, left-to-right, the minimax values are 7;6,7,4;9,6,9,7,6,7,4. Problem : Show the operation of alpha-beta pruning on the tree shown below. Show the (alpha, beta) windows passed to each node as the tree is traversed, the values returned from each node, and which branches are pruned. Labelling the nodes A through K top-to-bottom, left-to-right, the (α, β) windows passed to each node and the values returned are Node(α, β)Value A $(-\infty, \infty)$ 7 B $(-\infty, \infty)$ 4 C $(4, \infty)$ 7 D $(7, \infty)$ 6 E $(-\infty, \infty)$ 4 F $(-\infty, 4)$ [prunes 2nd child] 6 G $(-\infty, 4)$ [prunes 2nd child] 7 H $(4, \infty)$ 7 I $(4, 7)$ [prunes 2nd, 3rd children] 9 J $(7, \infty)$ 9 K $(7, 9)$ 6 Problem : Show the operation of Scout on he tree from the previous problem. Show the (alpha, beta) windows passed to each node as the tree is traversed, the values returned from each node, and which branches are pruned. Labelling the nodes A through K top-to-bottom, left-to-right, the (α, β) windows passed to each node (nodes that are examined multiple times are given with each window passed to them and each returned value) and the values returned are Node(α, β)Value A $(-\infty, \infty)$ 7 B $(-\infty, \infty)$ 4 C $(4, 5)$$(7, \infty)$ [prunes second child] 77 D $(7, 8)$ 6 E $(-\infty, \infty)$ 4 F $(3, 4)$ [prunes 2nd child] 6 G $(3, 4)$ [prunes 2nd child] 7 H $(4, 5)$ [prunes 2nd child]$(7, \infty)$ 77 I $(4, 5)$ [prunes 2nd, 3rd children] 99 J $(7, 8)$ [prunes 2nd, 3rd children] 9 K $(7, 8)$ 6 Problem : Show what would be stored in the transposition table after using alpha-beta pruning on the tree from above. Assume that the transposition table is initially empty and there are no collisions. Labelling the nodes A through K top-to-bottom, left-to-right, the depths and bounds stored in the transposition table are NodeDepthBound A 3 =7 B 2 =4 C 2 =7 D 2 =6 E 1 =4 F 1 ≥6 G 1 ≥7 H 1 =7 I 1 ≥9 J 1 =9 K 1 =6 Problem : Condsider the multi-armed bandit problem with 3 arms with payoffs determined by the outcomes of rolling a fair 6-, 8-, or 10-sided die. Show that the greedy strategy does not achieve zero average regret for this setup. If we roll on 1 on the 10-sided die anything other than a 1 on either of the other dice then the average regret for each subsequent roll will be non-zero ,and so the limit of the average regret is non-zero. That happens with probability $\frac{1}{10} \cdot (1 - \frac{1}{8}\cdot\frac{1}{6}) > 0$, so the probability that the average regret goes to zero in the limit is less than 1. Problem : Describe modifcations that would be necessary to standard MCTS for 2-player Yahtzee. The tree policy would have to account for the stochastic element. At random event positions, the tree policy could randomly choose the next position according to the probability distribution of that event. Problem : Pick an two enhancements to standard MCTS and describe how they might be applied to NFL Strategy, Kalah, or Yahtzee. Predicate Average Sampling Technique: for NFL Strategy, use predicates "trailing by more than 21 points" and "leading by more than 21 points" and keep track of the success of various plays in those situations during random playouts. Then bias play selection towards the plays that are better in those situations when in positions matching those situations. Search Seeding: use a heuristic function for Kalah (perhaps our simple score margin heuristic) to initialize visit count and reward statistics for newly expanded nodes in the hopes of reducing the amount of time determining that bad moves are in fact bad. Problem : Consider using coevolution to optimize a set of numeric parameters in a heuristic for chess. Describe some ways to get useful information from playing two members of the population against each other multiple times when ordinarily the resulting players are deterministic so each game would follow the same sequence of moves. As described in Genetic Algorithms for Evolving Computer Chess Programs (O. David et al; see reading list), we can play multiple games with randomly selected opening move sequences. Problem : Describe how you would set up the inputs to a neural network that chooses plays for NFL Strategy. Describe how you would use the outputs to choose a play. Inputs: distance to goal line, normalized to $[0.0, 1.0)$ time left in quarter, normalized to $[0.0, 1.0)$ current quarter (1.0 = 1st, 0.66 = 2nd, 0.33 = 3rd, 0.0 = 4th) current down (1.0 = 1st, 0.66 = 2nd, 0.33 = 3rd, 0.0 = 4th) yards to first down divided by 10 (this one is interesting: we should normalize somehow, but dividing by 100 would result in compressing the most frequently seen positions to a small range; normalizing to standard deviations away from the mean seen during actual play may be better) score margin divided by 20 (again, the best normalization may be determined by experimentation) (and field position as -1 for left, 0, for center, 1 for right if you wish to include that feature of the commercial game) Outputs: one output for each play, interpreted as the probability of selecting that play (so the output layer should use a softmax activation function) Problem : How did DeepMind address the issue of overfitting when training the value network they used in AlphaGo? The value network was trained using positions reached in games played by the policy network obtained after reinforement learning against itself. Instead of training on every position reached during every game, they trained using one position sampled from 30 million different games. Alpha Beta Pruning is an optimization technique of the Minimax algorithm. This algorithm solves the limitation of exponential time and space complexity in the case of the Minimax algorithm by pruning redundant branches of a game tree using its parameters Alpha(α\alphaα) and Beta(β\betaβ). Basic knowledge of space and time complexity Minimax algorithm In AI, an agent takes various decisions in an environment to maximize its gains/rewards. Adversarial search is a paradigm wherein multiple agents are competing against each other in the same environment to maximize their gains while minimizing that of others. The Minimax algorithm is an adversarial search algorithm that involves a Depth-First-Search(DFS) on the game tree.However, as the depth of the game tree increases, the number of states increases exponentially, leading to high time and space complexity. This is solved by the Alpha Beta Pruning algorithm, an optimization technique of the Minimax algorithm.The following sections present the Alpha Beta Pruning algorithm details for the case when two competing agents take alternate turns. One agent is a maximizer who wants to maximize the utility, whereas the other agent is a minimizer who wants to minimize the utility. The maximizer gets the first turn. Our goal is to search for an optimal sequence of actions for the maximizer. The following concepts can be extrapolated for other multi-agent environments as well.Interesting Fact: The literal meaning of pruning refers to the discarding of unwanted branches of a tree in gardening. Alpha Beta Pruning in AI is named as such because it involves pruning the unnecessary branches of a game tree by using two parameters, Alpha(α\alphaα) and Beta(β\betaβ), described in the next section. Alpha(α\alphaα): The best choice(highest utility/value) found till the current state on the path traversed by the maximizer. Beta(β\betaβ): The best choice(lowest utility/value) found till the current state on the path traversed by the minimizer. The critical points of Alpha Beta Pruning in AI are as follows. The initialization of the parameters Alpha(α\alphaα) is initialized with −∞-\infty−∞. Beta(β\betaβ) is initialized with +∞+\infty+∞. Updating the parameters Alpha(α\alphaα) is updated only by the maximizer at its turn. Beta(β\betaβ) is updated only by the minimizer at its turn. Passing the parameters Alpha(α\alphaα) and Beta(β\betaβ) are passed on to only child nodes. While backtracking the game tree, the node values are passed to parent nodes. Pruning Condition The child sub-trees, which are not yet traversed, are pruned if the condition α≥β\alpha \geq\betaα≥β holds. In the next section, we conglomerate all these properties and conditions into the pseudo-code of Alpha Beta Pruning in AI. Keeping all the pointers stated previously, we present the following pseudo-code for the Alpha Beta Pruning algorithm. To illustrate the working of Alpha Beta Pruning in AI, we use an example game tree shown in Figure-1. The numbers shown in terminal nodes represent their respective utilities. Figure-1The Minimax algorithm would have traversed the complete game tree leading to the game tree shown in Figure-2. Figure-2Alpha Beta Pruning algorithm restricts the agent from visiting redundant sub-trees leading to faster average search time complexity. To visualize its working, we refer to the same game tree in Figure-1. We start from node N1N_1N1 with α=−∞\alpha=-\infty α=−∞ and β=∞\beta=\infty β=∞. From N1N_1N1, we move to its child node N2N_2N2 with the same α\alphaα and β\betaβ. Similarly, we move from N2N_2N2 to its child N4N_4N4, as shown in Figure-3. Figure-3From N4N_4N4, we move to its first terminal child and get back its utility as 111. Since N4N_4N4 is a maximiser, we update α=max(−∞,1)=1\alpha=max(-\infty,1)=1α=max(−∞,1)=1. The pruning condition α≥β\alpha\geq\betaα≥β remains unsatisfied. Therefore, we move to the second terminal child of N4N_4N4 and get its utility as 444. Further, we update α=max(1,4)=4\alpha=max(1,4)=4α=max(1,4)=4. After this step, we get the node value of N4N_4N4 as max(1,4)=4max(1,4)=4max(1,4)=4 and return it to its parent N2N_2N2. Since N2N_2N2 is a minimiser, it updates β=min(+∞,4)=4\beta=min(+\infty,4)=4β=min(+∞,4)=4. The game tree after this step is shown in Figure-4. Figure-4At N2N_2N2, the pruning condition α≥β\alpha\geq\betaα≥β is still unsatisfied. Therefore, we visit its next child node N5N_5N5 with α=−∞\alpha=-\infty α=−∞ and β=4\beta=4β=4. At N5N_5N5, we get the utility of its first terminal child as 777. Since N5N_5N5 is a maximiser, we update α=max(−∞,7)=7\alpha=max(-\infty,7)=7α=max(−∞,7)=7. Now, the pruning condition gets satisfied (7≥4)(7\geq4)(7≥4), leading to the pruning of the other child node. After that, the node utility of N5=7N_5=7N5=7 is further sent back to the parent node N2N_2N2. N2N_2N2 updates β=min(4,7)=4\beta=min(4,7)=4β=min(4,7)=4. Further, the node value of N2=min(4,7)=4N_2=min(4,7)=4N2=min(4,7)=4 is returned to its parent N1N_1N1. At N1N_1N1, we update α=max(−∞,4)=4\alpha=max(-\infty,4)=4α=max(−∞,4)=4. After this step, the game tree looks like Figure-5. Figure-5The pruning condition is still unsatisfactory at N1N_1N1. Therefore, we continue towards the next child N3N_3N3, with the same values of α\alphaα and β\betaβ. Similarly, we propagate from N3N_3N3 to its first child N6N_6N6. At N6N_6N6, we get back the utility of its first terminal child as 333 and update α=max(4,3)=4\alpha=max(4,3)=4α=max(4,3)=4. Since the pruning condition is still unsatisfied, we get the utility of the second terminal child as 000 and update α=max(4,0)=4\alpha=max(4,0)=4α=max(4,0)=4. After this, the node value of N6=max(3,0)=3N_6=max(3,0)=3N6=max(3,0)=3, is returned to the parent N3N_3N3. Since N3N_3N3 is a minimiser, we update β=min(+∞,3)=3\beta=min(+\infty,3)=3β=min(+∞,3)=3. Now, the pruning condition gets satisfied (4≥3)(4\geq3)(4≥3). Therefore, the other child sub-tree(rooted at N7N_7N7) of N3N_3N3 is pruned, and the node value of N3N_3N3 becomes 333, which is returned to parent N1N_1N1. At N1N_1N1, we again update α=max(4,3)=4\alpha=max(4,3)=4α=max(4,3)=4 and the final utility of N1N_1N1 becomes 444. All these steps are shown in Figure-6. Figure-6Figure-6 depicts the final game tree of the Alpha Beta Pruning algorithm. As evident from the figure, Alpha Beta Pruning in AI yields the same result as the Minimax algorithm by visiting fewer states. The performance of the Alpha Beta Pruning algorithm is highly dependent on the order in which the nodes are traversed. For instance,Case-1(Worst Performance): If the best sequence of actions is on the right of the game tree, then there will be no pruning, and all the nodes will be traversed, leading to even worse performance than the Minimax algorithm because of the overhead of computing α\alphaα and β\betaβ. An example game tree can be seen in Figure-7. Figure-7Case-2(Best Performance): If the best sequence of actions is on the left of the game tree, there will be a lot of pruning, and only about half of the nodes will be traversed to get the optimal result. An example game tree can be seen in Figure-8. Figure-8 The best move occurs from the shallowest node. The game tree is ordered so that the best nodes are checked first. Domain knowledge is used to find the best move. Memoization is applied to prevent recomputation in case of repeating states. In this article, we learned about Significance of Alpha Beta Pruning algorithm Parameters of Alpha Beta Pruning in AI Properties and Conditions of Alpha Beta Pruning in AI Pseudo-code of Alpha Beta Pruning algorithm Working on Alpha Beta Pruning in AI Significance of order of nodes for Alpha Beta Pruning algorithm and rules to decide good orders Search algorithm that seeks to decrease the number of nodes in the minimax algorithm search tree For other uses, see Alphabeta (disambiguation). Alpha–beta pruningClassSearch algorithmWorst-case performance O (b d) {\displaystyle O(b^{d})} Best-case performance O (b d) {\displaystyle O\left({\sqrt {b^{d}}}\right)} Alpha–beta pruning is a search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree. It is an adversarial search algorithm used commonly for machine playing of two-player combinatorial games (Tic-tac-toe, Chess, Connect 4, etc.). It stops evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves need not be evaluated further. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.[1] John McCarthy during the Dartmouth Workshop met Alex Bernstein of IBM, who was writing a chess program. McCarthy invented alpha–beta search and recommended it to him, but Bernstein was "unconvinced".[2] Allen Newell and Herbert A. Simon who used what John McCarthy calls an "approximation"[3] in 1958 wrote that alpha–beta "appears to have been reinvented a number of times".[4] Arthur Samuel had an early version for a checkers simulation. Richards, Timothy Hart, Michael Levin and/or Daniel Edwards also invented alpha–beta independently in the United States.[5] McCarthy proposed similar ideas during the Dartmouth workshop in 1956 and suggested it to a group of his students including Alan Kotok at MIT in 1961.[6] Alexander Brudno independently conceived the alpha–beta algorithm, publishing his results in 1963.[7] Donald Knuth and Ronald W. Moore refined the algorithm in 1975.[8][9] Judea Pearl proved its optimality in terms of the expected running time for trees with randomly assigned leaf values in two papers.[10][11] The optimality of the randomized version of alpha–beta was shown by Michael Saks and Avi Wigderson in 1986.[12] A game tree can represent many two-player zero-sum games, such as chess, checkers, and reversi. Each node in the tree represents a possible situation in the game. Each terminal node (outcome) of a branch is assigned a numeric score that determines the value of the outcome to the player with the next move.[13] The algorithm maintains two values, alpha and beta, which respectively represent the minimum score that the maximizing player is assured of and the maximum score that the minimizing player is assured of. Initially, alpha is negative infinity and beta is positive infinity, i.e. both players start with their worst possible score. Whenever the maximum score that the minimizing player (i.e. the "beta" player) is assured of becomes less than the minimum score that the maximizing player (i.e., the "alpha" player) is assured of (i.e. beta < alpha), the maximizing player need not consider further descendants of this node, as they will never be reached in the actual play. To illustrate this with a real-life example, suppose somebody is playing chess, and it is their turn. Move "A" will improve the player's position. The player continues to look for moves to make sure a better one hasn't been missed. Move "B" is also a good move, but the player then realizes that it will allow the opponent to force checkmate in two moves. Thus, other outcomes from playing move B no longer need to be considered since the opponent can force a win. The maximum score that the opponent could force after move "B" is negative infinity: a loss for the player. This is less than the minimum position that was previously found; move "A" does not result in a forced loss in two moves. An illustration of alpha–beta pruning. The grayed-out subtrees don't need to be explored (when moves are evaluated from left to right), since it is known that the group of subtrees as a whole yields the value of an equivalent subtree or worse, and as such cannot influence the final result. The max and min levels represent the turn of the player and the adversary, respectively. The benefit of alpha–beta pruning lies in the fact that branches of the search tree can be eliminated.[13] This way, the search time can be limited to the 'more promising' subtree, and a deeper search can be performed in the same time. Like its predecessor, it belongs to the branch and bound class of algorithms. The optimization reduces the effective depth to slightly more than half that of simple minimax if the nodes are evaluated in an optimal or near optimal order (best choice for side on move ordered first at each node). With an (average or constant) branching factor of b, and a search depth of d plies, the maximum number of leaf node positions evaluated (when the move ordering is pessimal) is O(bd) – the same as a simple minimax search. If the move ordering for the search is optimal (meaning the best moves are always searched first), the number of leaf node positions evaluated is about O(b×1×b×1×...×b) for odd depth and O(b×1×b×1×...×1) for even depth, or O (b d / 2) = O (b d) {\displaystyle O(b^{d/2})=O({\sqrt {b^{d}}})} . In the latter case, where the ply of a search is even, the effective branching factor is reduced to its square root, or, equivalently, the search can go twice as deep with the same amount of computation.[14] The explanation of b×1×b×1×... is that all the first player's moves must be studied to find the best one, but for each, only the second player's best move is needed to refute all but the first (and best) first player move—alpha–beta ensures no other second player moves need be considered. When nodes are considered in a random order (i.e., the algorithm randomizes), asymptotically, the expected number of nodes evaluated in uniform trees with binary leaf-values is Θ (((b − 1 + b 2 + 14 b + 1) / 4) d) {\displaystyle \Theta (((b-1+{\sqrt {b^{2}+14b+1}})/4)^{d})} .[12] For the same trees, when the values are assigned to the leaf values independently of each other and say zero and one are both equally probable, the expected number of nodes evaluated is Θ ((b / 2) d) {\displaystyle \Theta ((b/2)^{d})} , which is much smaller than the work done by the randomized algorithm, mentioned above, and is again optimal for such random trees.[10] When the leaf values are chosen independently of each other but from the [0 , 1] {\displaystyle [0,1]} interval uniformly at random, the expected number of nodes evaluated increases to Θ (b d / l o g (d)) {\displaystyle \Theta (b^{d/log(d)})} in the d → ∞ {\displaystyle d\to \infty } limit,[11] which is again optimal for this kind of random tree. Note that the actual work for "small" values of d {\displaystyle d} is better approximated using 0.925 d 0.747 {\displaystyle 0.925d^{0.747}} .[11][10] A chess program that searches four plies with an average of 36 branches per node evaluates more than one million terminal nodes. An optimal alpha-beta prune would eliminate all but about 2,000 terminal nodes, a reduction of 99.8%.[13] An animated pedagogical example that attempts to be human-friendly by substituting initial infinite (or arbitrarily large) values for emptiness and by avoiding using the negamax coding simplifications. Normally during alpha–beta, the subtrees are temporarily dominated by either a first player advantage (when many first player moves are good, and at each search depth the first move checked by the first player is adequate, but all second player responses are required to try to find a refutation), or vice versa. This advantage can switch sides many times during the search if the move ordering is incorrect, each time leading to inefficiency. As the number of positions searched decreases exponentially each move nearer the current position, it is worth spending considerable effort on sorting early moves. An improved sort at any depth will exponentially reduce the total number of positions searched, but sorting all positions at depths near the root node is relatively cheap as there are so few of them. In practice, the move ordering is often determined by the results of earlier, smaller searches, such as through iterative deepening. Additionally, this algorithm can be trivially modified to return an entire principal variation in addition to the score. Some more aggressive algorithms such as MTD(f) do not easily permit such a modification. The pseudo-code for depth limited minimax with alpha–beta pruning is as follows:[15] function alphabeta(node, depth, α, β, maximizingPlayer) is if depth == 0 or node is terminal then return the heuristic value of node if maximizingPlayer then value := −∞ for each child of node do value := max(value, alphabeta(child, depth − 1, α, β, FALSE)) if value ≥ β then break (* β cutoff *) α := max(α, value) return value else value := +∞ for each child of node do value := min(value, alphabeta(child, depth − 1, α, β, TRUE)) if value ≤ α then break (* α cutoff *) β := min(β, value) return value (* Initial call *) alphabeta(origin, depth, −∞, +∞, TRUE) Implementations of alpha–beta pruning can often be delineated by whether they are "fail-soft," or "fail-hard". The pseudo-code illustrates the fail-soft variation. With fail-soft alpha–beta, the alphabeta function may return values (v) that exceed (v < α or v > β) the α and β bounds set by its function call arguments. In comparison, fail-hard alpha–beta limits its function return value into the inclusive range of α and β. Further improvement can be achieved without sacrificing accuracy by using ordering heuristics to search earlier parts of the tree that are likely to force alpha–beta cutoffs. For example, in chess, moves that capture pieces may be examined before moves that do not, and moves that have scored highly in earlier passes through the game-tree analysis may be evaluated before others. Another common, and very cheap, heuristic is the killer heuristic, where the last move that caused a beta-cutoff at the same tree level in the tree search is always examined first. This idea can also be generalized into a set of refutation tables. Alpha–beta search can be made even faster by considering only a narrow search window (generally determined by guesswork based on experience). This is known as an aspiration window. In the extreme case, the search is performed with alpha and beta equal; a technique known as zero-window search, null-window search, or scout search. This is particularly useful for win/loss searches near the end of a game where the extra depth gained from the narrow window and a simple win/loss evaluation function may lead to a conclusive result. If an aspiration search fails, it is straightforward to detect whether it failed high (high edge of window was too low) or low (lower edge of window was too high). This gives information about what window values might be useful in a re-search of the position. Over time, other improvements have been suggested, and indeed the Falphabeta (fail-soft alpha–beta) idea of John Fishburn is nearly universal and is already incorporated above in a slightly modified form. Fishburn also suggested a combination of the killer heuristic and zero-window search under the name Lalphabeta ("last move with minimal window alpha–beta search"). Since the minimax algorithm and its variants are inherently depth-first, a strategy such as iterative deepening is usually used in conjunction with alpha–beta so that a reasonably good move can be returned even if the algorithm is interrupted before it has finished execution. Another advantage of using iterative deepening is that searches at shallower depths give move-ordering hints, as well as shallow alpha and beta estimates, that both can help produce cutoffs for higher depth searches much earlier than would otherwise be possible. Algorithms like SSS*, on the other hand, use the best-first strategy. This can potentially make them more time-efficient, but typically at a heavy cost in space-efficiency.[16] Minimax Expectiminimax Negamax Pruning (algorithm) Branch and bound Combinatorial optimization Principal variation search Transposition table ^ Russell & Norvig 2021, p. 152-161. ^ McCarthy, John (2006-10-30). "The Dartmouth Workshop--as planned and as it happened". www-formal.stanford.edu. Retrieved 2023-10-29. ^ McCarthy, John (27 November 2006). "Human Level AI Is Harder Than It Seemed in 1955". Stanford University. Retrieved 2006-12-20. ^ Newell, Allen; Simon, Herbert A. (1 March 1976). "Computer science as empirical inquiry: symbols and search". Communications of the ACM. 19 (3): 113–126. doi:10.1145/360018.360022. ^ Edwards, D.J.; Hart, T.P. (4 December 1961). The Alpha–beta Heuristic (Technical report). Massachusetts Institute of Technology. hdl:1721.1/6098. AIM-030. ^ Kotok, Alan (2004) [1962]. "A Chess Playing Program". Artificial Intelligence Project. RLE and MIT Computation Center. Memo 41. Retrieved 2006-07-01. ^ Marsland, T.A. (May 1987). "Computer Chess Methods" (PDF). In Shapiro, S. (ed.). Encyclopedia of Artificial Intelligence. Wiley. pp. 159–171. ISBN 978-0-471-62974-0. Archived from the original (PDF) on 2008-10-30. ^ Knuth, Donald E.; Moore, Ronald W. (1975). "An analysis of alpha-beta pruning". Artificial Intelligence. 6 (4): 293–326. doi:10.1016/0004-3702(75)90019-3. S2CID 7894372. ^ Abramson, Bruce (1 June 1989). "Control strategies for two-player games". ACM Computing Surveys. 21 (2): 137–161. doi:10.1145/66443.66444. S2CID 11526154. ^ a b c Pearl, Judea (1980). "Asymptotic Properties of Minimax Trees and Game-Searching Procedures". Artificial Intelligence. 14 (2): 113–138. doi:10.1016/0004-3702(80)90037-5. ^ a b c Pearl, Judea (1982). "The Solution for the Branching Factor of the Alpha-Beta Pruning Algorithm and Its Optimality". Communications of the ACM. 25 (8): 559–64. doi:10.1145/358589.358616. S2CID 8296219. ^ a b Saks, M.; Wigderson, A. (1986). "Probabilistic Boolean Decision Trees and the Complexity of Evaluating Game Trees". 27th Annual Symposium on Foundations of Computer Science. pp. 29–38. doi:10.1109/SFCS.1986.44. ISBN 0-8186-0740-8. S2CID 6130392. ^ a b c Levy, David (January 1986). "Alpha-Beta Soup". MacUser. pp. 98–102. Retrieved 2021-10-19. ^ Russell & Norvig 2021, p. 155. ^ Russell & Norvig 2021, p. 154. ^ Pearl, Judea; Korf, Richard (1987), "Search techniques", Annual Review of Computer Science, 2: 451–467, doi:10.1146/annurev.cs.02.060187.002315, Like its A* counterpart for single-player games, SSS* is optimal in terms of the average number of nodes examined; but its superior pruning power is more than offset by the substantial storage space and bookkeeping required. Russell, Stuart J.; Norvig, Peter. (2021). Artificial Intelligence: A Modern Approach (4th ed.). Hoboken: Pearson. ISBN 9780134610993. LCCN 20190474. Heineman, George T.; Pollice, Gary; Selkow, Stanley (2008). "7. Path Finding in AI". Algorithms in a Nutshell. Oreilly Media. pp. 217–223. ISBN 978-0-596-51624-6. Pearl, Judea (1984). Heuristics: Intelligent Search Strategies for Computer Problem Solving. Addison-Wesley. ISBN 978-0-201-05594-8. OCLC 1035596197. Fishburn, John P. (1984). "Appendix A: Some Optimizations of α-β Search". Analysis of Speedup in Distributed Algorithms (revision of 1981 PhD thesis). UMI Research Press. pp. 107–111. ISBN 0-8357-1527-2. Retrieved from " DSA to Development: A Complete GuideBeginner to AdvanceJAVA Backend Development - LiveIntermediate and AdvanceTech Interview 101 - From DSA to System Design for Working ProfessionalsBeginner to AdvanceFull Stack Development with React & Node JS - LiveBeginner to AdvanceJava Programming Online Course [Complete Beginner to Advanced]Beginner to AdvanceC++ Programming Course Online - Complete Beginner to AdvancedBeginner to AdvancePage 2Our website uses cookiesWe use cookies to ensure you have the best browsing experience on our website. By using our site, you acknowledge that you have read and understood our Cookie Policy & Privacy Policy Open-Source Internship opportunity by OpenGenus for programmers. Apply now. Reading time: minutes | Coding time: minutes Like we discussed in earlier article, a hard coded AI can be used to create an intelligent opponent which you can challenge to a game of classical tic-tac-toe! But, that algorithm could still be optimised further. But how do we achieve that? The Alpha Beta Pruning is a search algorithm that tries to diminish the quantity of hubs that are assessed by the minimax algorithm in its search tree. It is an antagonistic search algorithm utilized usually for machine playing of two-player recreations (Tic-tac-toe, Chess, Go, and so forth.). It stops totally assessing a move when no less than one probability has been observed that ends up being more regrettable than a formerly analyzed move. Such moves require not be assessed further. At the point when connected to a standard minimax tree, it restores an indistinguishable move from minimax would, however prunes away branches that can't in any way, shape or form impact an official conclusion! You must be thinking what does this figure really signifies. Well, it shows how the alogorithm ignores the sub trees which are not really desireable in our game moves. The algorithm keeps up two qualities, alpha and beta, which speak to the base score that the expanding player is guaranteed of and the greatest score that the limiting player is guaranteed of separately. At first alpha is negative infinity and beta is sure infinity, i.e. the two players begin with their most exceedingly terrible conceivable score. At whatever point the most extreme score that the limiting player(beta) is guaranteed of turns out to be not as much as the base score that the expanding player(alpha) is guaranteed of (i.e. beta alpha) alpha = score end if alpha >= beta break end return alpha end else #player is minimizing player best_score = +infinity for each child score = minimax(depth - 1, player, alpha, beta) if (score < beta) beta = score end if alpha >= beta break end return beta end end end #then you would call it like minimax(2, computer, -inf, +inf) Complexity Worst case Performance : O(|E|) = O(b^(d/2)) Worst case space complexity : O(|V|) = O(b * d) Implementations Next Best Move Guesser- Python import numpy as np import sys from copy import copy rows = 3 cols = 3 board = np.zeros((rows,cols)) inf = float("inf") neg_inf = float("-inf") def printBoard(): for i in range(rows): for j in range(cols): if board[i, j] == 0: sys.stdout.write(' _ ') elif board[i, j] == 1: sys.stdout.write(' X ') else: sys.stdout.write(' O ') print '' # The heuristic function which will be evaluating board's position for each of the winning POS heuristicTable = np.zeros((rows+1, cols+1)) numberOfWinningPositions = rows + cols + 2 for index in range(rows+1): heuristicTable[index, 0] = 10**index heuristicTable[0, index] =-10**index winningArray = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8], [0, 3, 6], [1, 4, 7], [2, 5, 8], [0, 4, 8], [2, 4, 6]]) def utilityOfState(state): stateCopy = copy(state.ravel()) heuristic = 0 for i in range(numberOfWinningPositions): maxp = 0 minp = 0 for j in range(rows): if stateCopy[winningArray[i,j]] == 2: maxp += 1 elif stateCopy[winningArray[i,j]] == 1: minp += 1 heuristic += heuristicTable[maxp][minp] return heuristic def minimax(state,alpha,beta,maximizing,depth,maxp,minp): if depth==0: return utilityOfState(state),state rowsLeft,columnsLeft=np.where(state==0) returnState=copy(state) if rowsLeft.shape[0]==0: return utilityOfState(state),returnState if maximizing==True: utility=neg_inf for i in range(0,rowsLeft.shape[0]): nextState=copy(state) nextState[rowsLeft[i],columnsLeft[i]]=maxp #print 'in max currently the Nextstate is ',nextState,'' Nutility,Nstate=minimax(nextState,alpha,beta,False,depth-1,maxp,minp) if Nutility > utility: utility=Nutility returnState=copy(nextState) if utility>alpha: alpha=utility if alpha >=beta : #print 'pruned' break; #print 'for max the best move is with utility ',utility,' n state ',returnState return utility,returnState else: utility=inf for i in range(0,rowsLeft.shape[0]): nextState=copy(state) nextState[rowsLeft[i],columnsLeft[i]]=minp #print 'in min currently the Nextstate is ',nextState,'' Nutility,Nstate=minimax(nextState,alpha,beta,True,depth-1,maxp,minp) if Nutility < utility: utility=Nutility returnState=copy(nextState) if utility< beta: beta=utility if alpha >=beta : #print 'pruned' break; return utility,returnState def checkGameOver(state): stateCopy=copy(state) value=utilityOfState(stateCopy) if value >=1000: return 1 return -1 def main(): num=input('enter player num (1st or 2nd) ') value=0 global board for turn in range(0,rows*cols): if (turn+num)%2==1: #make the player go first, and make the user player as 'X' r,c=[int(x) for x in raw_input('Enter your move ').split(' ')] board[r-1,c-1]=1 printBoard() value=checkGameOver(board) if value==1: print 'U win.Game Over' sys.exit() print '' else: #its the computer's turn make the PC always put a circle' #right now we know the state if the board was filled by the other player state=copy(board) value,nextState=minimax(state,neg_inf,inf,True,2,2,1) board=copy(nextState) printBoard() print '' value=checkGameOver(board) if value==1: print 'PC wins.Game Over' sys.exit() print 'Its a draw' if __name__ == "__main__": main() Applications Its major applications are in game of chess. Stockfish (chess) is a C++ open source chess program that implements the Alpha Beta pruning algorithm. Stockfish is a mature implementation that is rated as one of the strongest chess engines available today as evidenced by it winning the Top Chess Engine Championship in 2016 and 2017. One more interesting thing to notice is that implementations of alpha-beta pruning can often be delineated by whether they are "fail-soft," or "fail-hard." The pseudo-code illustrates the fail-soft variation. With fail-soft alpha-beta, the alphabeta function may return values (v) that exceed (v < α or v > β) the α and β bounds set by its function call arguments. In comparison, fail-hard alpha-beta limits its function return value into the inclusive range of α and β. Further Reading 17 Mar 2025 | 5 min readAlpha-beta pruning is a modified version of the minimax algorithm. It is an optimization technique for the minimax algorithm.As we have seen in the minimax search algorithm that the number of game states it has to examine are exponential in depth of the tree. Since we cannot eliminate the exponent, but we can cut it to half. Hence there is a technique by which without checking each node of the game tree we can compute the correct minimax decision, and this technique is called pruning. This involves two threshold parameter Alpha and beta for future expansion, so it is called alpha-beta pruning. It is also called as Alpha-Beta Algorithm.Alpha-beta pruning can be applied at any depth of a tree, and sometimes it not only prune the tree leaves but also entire sub-tree.The two-parameter can be defined as:Alpha: The best (highest-value) choice we have found so far at any point along the path of Maximizer. The initial value of alpha is -∞.Beta: The best (lowest-value) choice we have found so far at any point along the path of Minimizer. The initial value of beta is +∞.The Alpha-beta pruning to a standard minimax algorithm returns the same move as the standard algorithm does, but it removes all the nodes which are not really affecting the final decision but making algorithm slow. Hence by pruning these nodes, it makes the algorithm fast.The main condition which required for alpha-beta pruning is:α>=β The Max player will only update the value of alpha.The Min player will only update the value of beta.While backtracking the tree, the node values will be passed to upper nodes instead of values of alpha and beta.We will only pass the alpha, beta values to the child nodes.function minimax(node, depth, alpha, beta, maximizingPlayer) is if depth ==0 or node is a terminal node then return static evaluation of node if MaximizingPlayer then // for Maximizer Player maxEva= -infinity for each child of node do eva= minimax(child, depth-1, alpha, beta, False) maxEva= max(maxEva, eva) alpha= max(alpha, maxEva) if beta