INDIAN INSTITUTE OF TECHNOLOGY, GOA

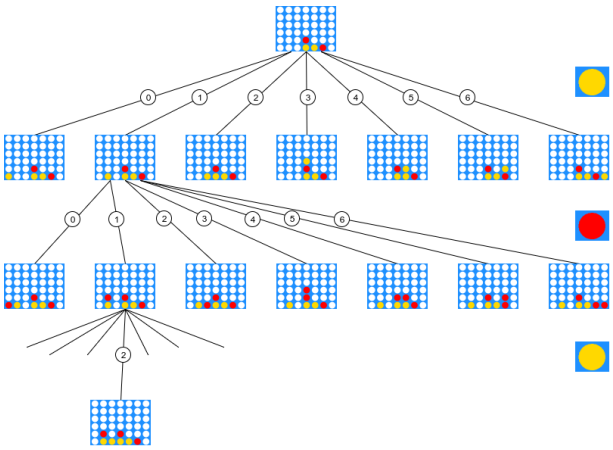
Game CONNECT-4 AI

Authors: *Priyanshu Singh, Naresh Kaushal*

Topic: *Game AI Algorithms* – Professor: *G. Siva Kumar Reddy*
Submission Date: *March 28th, 2025*

Theory

Game trees, minimax and alpha-beta pruning



Main Idea. For every game that has a discrete action space (or a finite number of possible actions in every move), we can construct a game tree in which each node represents a possible game state. The internal nodes at an even depth represent either the initial game state (the root) or a game state which resulted from a move made by the opponent. The internal nodes at an odd depth represent game states resulting from moves made by us. If a state is game-ending (four tokens connected or board is full), it is a leaf node. Each leaf node is awarded a certain score and not further expanded. Below is an example of a game subtree for the connect-four game.

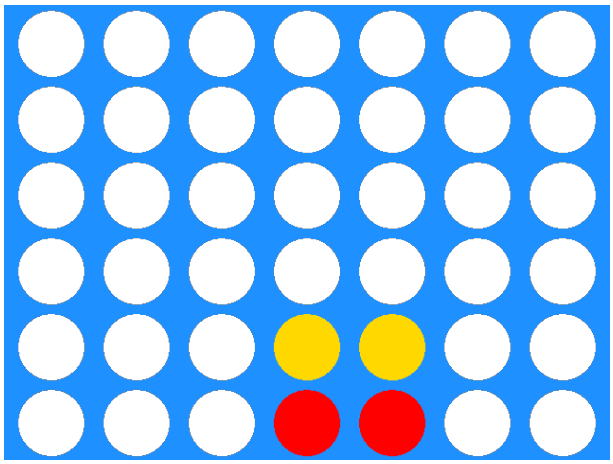
In total, there are **4531985219092** possible leaf nodes and thus a much larger number of total nodes in the tree. Even with the optimized bitboard operations, it is computationally infeasible to traverse the entire game tree. We will need techniques to efficiently find a winning path in this tree.

Now, while the path 1–1–2 in the game tree from the figure above leads to a game-ending state where the yellow player wins (it is a winning path), it rests on the assumption that red is a stupid bot and screws up his move (he does not block you).

Since we do not know how 'good' the bot is we are going to play against, we have to assume the worst-case: what if our opponent is the best possible bot and thus makes the best possible move every time? If we can find a winning path against such a worst-case bot, then we can definitely take this path and be sure that we win the game (the real bot can only do worse, making the game end earlier). For the connect-four game, such a path exists if you are the starting player. Here is where the minimax algorithm comes in play.

Before we can apply this algorithm to game trees, we need to define a scoring function for each node in the tree.

1. +100 if a string of 4 horizontal, vertical or diagonal dots of same colour is found which belongs to us.
2. +5 if the string of 3 horizontal, vertical or diagonal dots of same colour which belongs to us is found and 1 empty valid slot is also present in particular orientation.
3. +2 if the string of 2 horizontal, vertical or diagonal dots of same colour which belongs to us is found and 2 empty valid slots are also present in particular orientation.
4. -4 if the string of 3 horizontal, vertical or diagonal dots of opponent is found and 1 empty valid slot is also present in particular orientation.



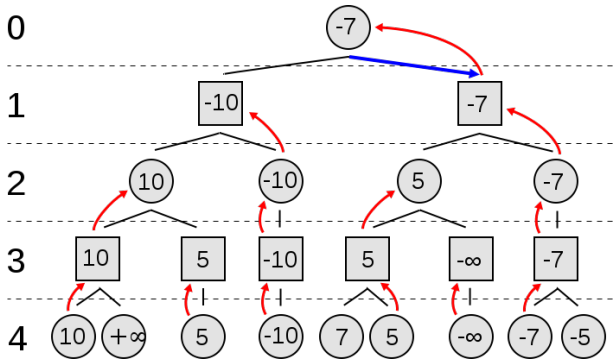
In the picture above, if we assume that we are the red player, we can assign a score of +2 to the game board, since the yellow player can't win because still 2 more yellow stones need to be placed.

In practice, it is hard to assign a score when the game has not yet finished. That's why we explore our tree until we reach a leaf node, calculate the score and propagate this score back upwards to the root. Now, when we are propagating these values upwards, internal nodes within our game tree will receive multiple values (a value for each of its children). The question is what value we should assign to the internal nodes. Now we can give the definition for the value of an internal node:

1. If the internal node is on an odd depth, we take the minimum value of the children their values (as an opponent, we want to make the final score as negative as possible, since we want to win)

2. If the internal node is on an even depth, we take the maximum value of the children their values (we want our score to be as positive as possible)

Here's an example, taken directly from wikipedia:



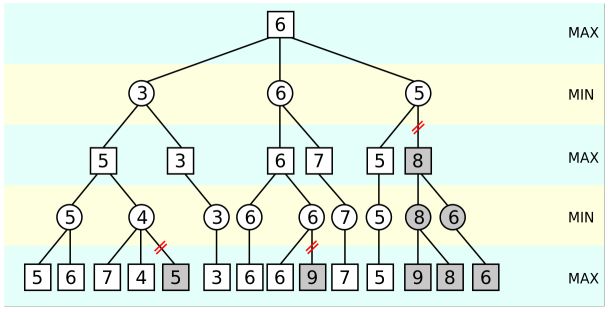
So now we have a way to find the optimal path within a game tree. The problem with this approach is that, especially in the beginning of the game, it takes a very long time to traverse the entire tree. We only have 1 second to make a move! Therefore, we introduce a technique that allows us to prune (large) parts of the game tree, such that we do not need to search it entirely. The algorithm is called alpha-beta pruning.

We define the following variables:

1. **alpha**: the current best score on the path to the root by the maximizer (us)

2. **beta**: the current best score on path to the root by minimizer (opponent)

What we do is update our alpha and beta values every time we see a new value from our children (depending on whether we are on an even or odd depth). We pass these alpha and beta along to our other children, now when we find a value that is either higher than our current beta, or lower than our current alpha, we can discard the entire subtree (since we are sure that the optimal path will not go through this). Let us take a look at another example, again taken from wikipedia:



1. We traverse the game tree depth first, from left to right. The first leaf we encounter is the left-most leaf (with value 5). The leaf propagates this value to its parent, where the beta-value is updated and changed into 5. It also checks the second left-most leaf (which has a value of 6) but this does not update any of the values (since 6 is not better than 5 if we are in the perspective of the minimizer).

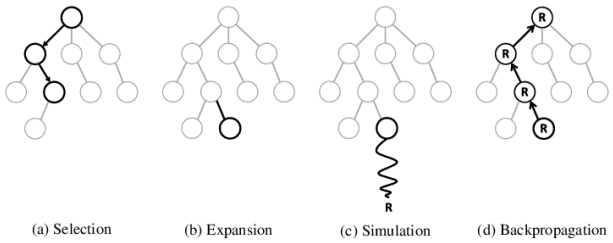
2. The best found value in this node (again, 5) gets propagated to its parent, where the alpha-value is updated. Now, we are at the the right child of this parent, and first checks its left child with value 7, and updates the beta-value (we now have alpha=5 and beta=7). We check the following child: with value 4, this is a better value for the minimizer so we now have beta=4 and alpha=5.

3. Since now beta $<=$ alpha, we can prune all the remaining children. This is because we will ALWAYS have a value $<=$ 4 now in that internal node (we are a minimizer and only update our value when we encounter a value smaller than the current one), BUT the parent node will only update values when they are $>=$ 5 (since we are a maximizer in that node). So whatever the value we will be after traversing all the nodes, it will never be chosen by the maximizer node, since it will must be larger than 5 in order for that to happen.

4. This process continues for all remaining nodes...

Theory

Introduction to Monte Carlo Tree Search



(a) Selection (b) Expansion (c) Simulation (d) Backpropagation

Main Idea. The subject of game AI generally begins with so-called perfect information games. These are turn-based games where the players have no information hidden from each other and there is no element of chance in the game mechanics (such as by rolling dice or drawing cards from a shuffled deck). Tic Tac Toe, Connect 4, Checkers, Reversi, Chess, and Go are all games of this type. Because everything in this type of game is fully determined, a tree can, in theory, be constructed that contains all possible outcomes, and a value assigned corresponding to a win or a loss for one of the players. Finding the best possible play, then, is a matter of doing a search on the tree, with the method of choice at each level alternating between picking the maximum value and picking the minimum value, matching the different players' conflicting goals, as the search proceeds down the tree. This algorithm is called Minimax.
The problem with Minimax, though, is that it can take an impractical amount of time to do a full search of the game tree. This is particularly true for games with a high

branching factor, or high average number of available moves per turn. This is because the basic version of Minimax needs to search all of the nodes in the tree to find the optimal solution, and the number of nodes in the tree that must be checked grows exponentially with the branching factor. There are methods of mitigating this problem, such as searching only to a limited number of moves ahead (or ply) and then using an evaluation function to estimate the value of the position, or by pruning branches to be searched if they are unlikely to be worthwhile. Many of these techniques, though, require encoding domain knowledge about the game, which may be difficult to gather or formulate. And while such methods have produced Chess programs capable of defeating grandmasters, similar success in Go has been elusive, particularly for programs playing on the full 19x19 board.
However, there is a game AI technique that does do well for games with a high branching factor and has come to dominate the field of Go playing programs. It is easy to create a basic implementation of this algorithm that will give good results for games with a smaller branching factor, and relatively simple adaptations can build on it and improve it for games like Chess or Go. It can be configured to stop after any desired amount of time, with longer times resulting in stronger game play. Since it doesn't necessarily require game-specific knowledge, it can be used for general game playing. It may even be adaptable to games that incorporate randomness in the rules. This technique is called Monte Carlo Tree Search.
Here we build a variant of it , called the Upper Confidence bound applied to Trees(UCT), and then will show you how to build a basic implementation in Python.

UCB1

Firse We construct Confidence Intervals

UCB1. Imagine, if you will, that you are faced with a row of slot machines, each with different payout probabilities and amounts. As a rational person (if you are going to play them at all), you would prefer to use a strategy that will allow you to maximize your net gain. But how can you do that? For whatever reason, there is no one nearby, so you can't watch someone else play for a while to gain information about which is the best machine. Clearly, your strategy is going to have to balance playing all of the machines to gather that information yourself, with concentrating your plays on the observed best machine. One strategy, called UCB1, does this by constructing statistical confidence intervals for each machine

$$\bar{x}_i \ \pm \sqrt{\frac{2\log n}{n_i}} \tag{1}$$

Where

1. $\bar{x}_i$: the mean payout for machine i
2. n_i : the number of plays of machine i
3. n : the total number of plays

Then, your strategy is to pick the machine with the highest upper bound each time. As you do so, the observed mean value for that machine will shift and its confidence interval will become narrower, but all of the other machines' intervals will widen. Eventually, one of the other machines will have an upper bound that exceeds that of your current one, and you will switch to that one. This strategy has the property that your regret, the difference between what you would have won by playing solely on the actual best slot machine and your expected winnings under the strategy that you do use, grows only as $O(\log n)$. This is the same big-O growth rate as the theoretical best for this problem (referred to as the multi-armed bandit problem), and has the additional benefit of being easy to calculate.
And here's how Monte Carlo comes in. In a standard Monte Carlo process, a large number of random simulations are run, in this case, from the board position that you want to find the best move for. Statistics are kept for each possible move from this starting state, and then the move with the best overall results is returned.

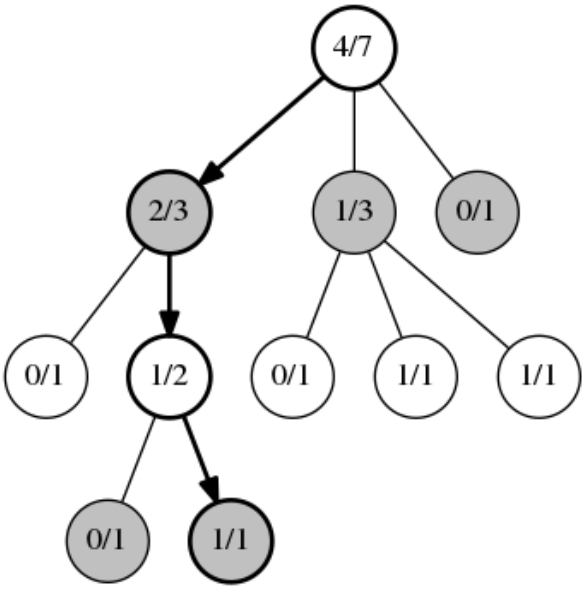
UCT

Problem with UCB

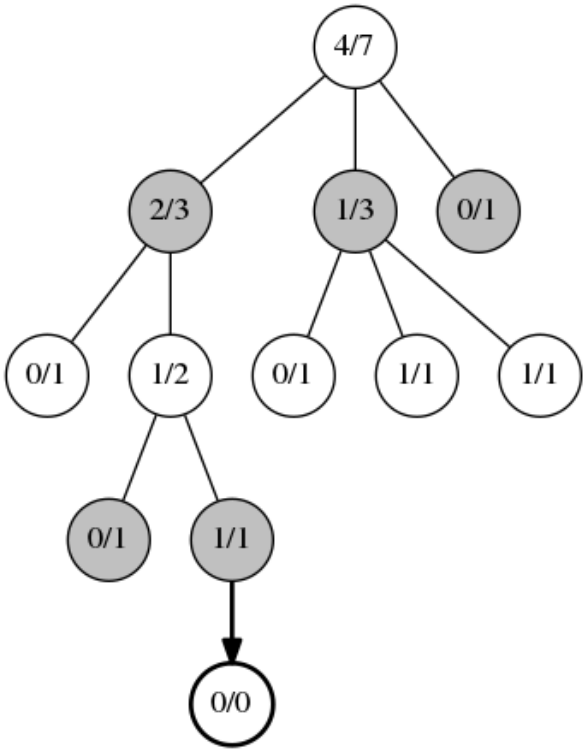
Problem. The problem with UCB1 is that for any given turn in the simulation, there may be many possible moves, but only one or two that are good. If a random move is chosen each turn, it becomes extremely unlikely that the simulation will hit upon the best path forward. So, UCT has been proposed as an enhancement. The idea is this: any given board position can be considered a multi-armed bandit problem, if statistics are available for all of the positions that are only one move away. So instead of doing many purely random simulations, UCT works by doing many multi-phase playouts.

Different Phases in UCT

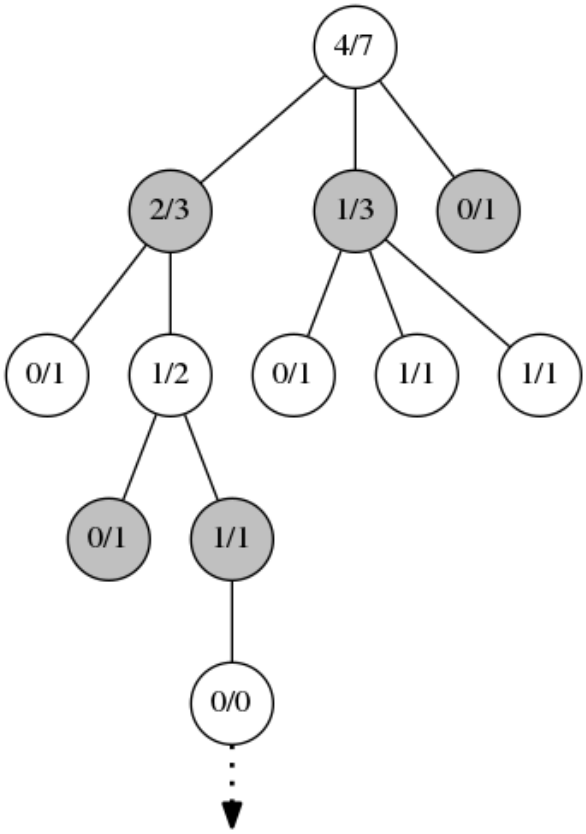
0.1. SELECTION. The first phase, selection, lasts while you have the statistics necessary to treat each position you reach as a multi-armed bandit problem. The move to use, then, would be chosen by the UCB1 algorithm instead of randomly, and applied to obtain the next position to be considered. Selection would then proceed until you reach a position where not all of the child positions have statistics recorded.



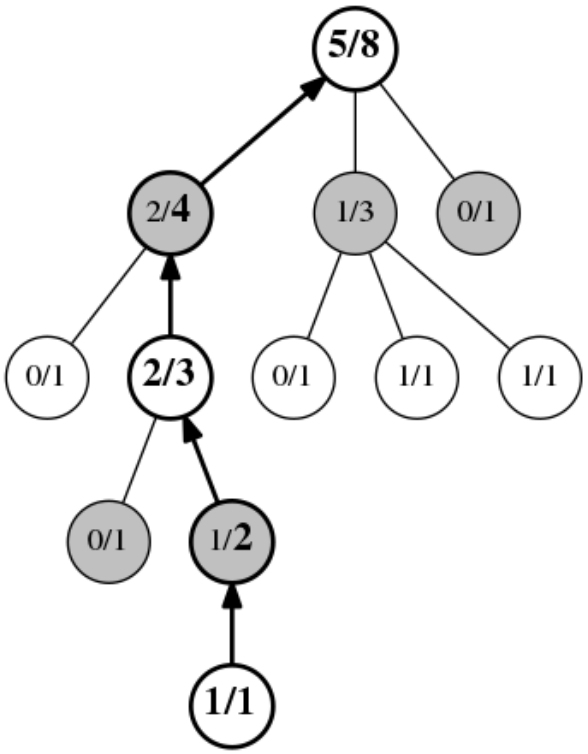
0.2. EXPANSION. The second phase, expansion, occurs when you can no longer apply UCB1. An unvisited child position is randomly chosen, and a new record node is added to the tree of statistics.



0.3. SIMULATION. After expansion occurs, the remainder of the playout is in phase 3, simulation. This is done as a typical Monte Carlo simulation, either purely random or with some simple weighting heuristics if a light playout is desired, or by using some computationally expensive heuristics and evaluations for a heavy playout. For games with a lower branching factor, a light playout can give good results.



0.4. PROPAGATION. Finally, the fourth phase is the update or back-propagation phase. This occurs when the playout reaches the end of the game. All of the positions visited during this playout have their play count incremented, and if the player for that position won the playout, the win count is also incremented.



This algorithm may be configured to stop after any desired length of time, or on some other condition. As more and more playouts are run, the tree of statistics grows in memory and the move that will finally be chosen will converge towards the actual optimal play, though that may take a very long time, depending on the game.

1. Components in the code

1. conevert2string(board)
2. write_to_plays_csv(plays)
3. write_to_wins_csv(wins)
4. read_from_plays_csv(plays)
5. read_from_wins_csv(wins)
6. convert2array(str_board)
7. create_board()
8. drop_piece(board, row, col, piece)

9. turn(board)
10. next_state_possible(board, col)
11. next_state(board, col)
12. who_won(state, turn)
13. is_valid_location(board, col)
14. get_next_open_row(board, col)
15. print_board(board)
16. to_tuple(arr)
17. to_arr(tup)
18. winning_move(board, piece)
19. is_terminal_node(board)
20. get_valid_locations(board)
21. draw_board(board)
22. class MonteCarlo
    - member variables
      - board
      - states
      - seconds
      - calculation_time
      - max_moves
      - wins
      - plays
      - history
    - Functions
      - update(self, state)
      - get_play(self)
      - run_simulation(self)

2. Meaning of the various components

1. convert2string : Converts the given board argument into string form which can be stored into log-File
2. write_to_plays_csv : Writes the dictionary 'plays' into plays.csv file
3. write_to_wins_csv : Writes the dictionary 'wins' into wins.csv file

4. read_from_plays_csv : Reads into dictionary 'plays' from plays.csv file
5. read_from_wins_csv : Reads into dictionary 'wins' from wins.csv file
6. convert2array : Converts the input string into it's equivalent board type
7. create_board : Creates a board type instance for the game
8. drop_piece : Drops a piece corresponding to the user into the board, there are two types of pieces in the game
9. turn : Tells who's turn it is
10. next_state_possible : Returns the next board state instance on dropping the piece in the column given to it in the arguments
11. next_state : Also returns the next board state instance on dropping in the column specified in the arguments, but, this changes the original board we are playing on. That is, next_state_possible is used for simulation purposes, while the other is used for playing the game
12. who_won : Returns true or false of whether the player has won or not. And if true, it returns the player who won
13. is_valid_location : Returns whether the column is available for dropping our piece
14. get_next_open_row : Returns the row into which we have to drop our piece in the specified column
15. print : Prints our board
16. to_tuple : Converts the board from list of lists to tuple of tuple form, for indexing in the dictionaries
17. to_arr : Converts the board from tuple of tuples to list of lists for our manipulation
18. winning_move : Returns true of false whether someone has won or not after the current move
19. is_terminal_node : Tells if the game has ended and there are no moves to play ahead
20. get_valid_locations : returns the possible places for dropping the piece
21. draw_board : draws the board on the window using pygame
22. MonteCarlo : Class for Monte Carlo
23. board : Member variable which abstracts a Board of our game as a list of lists
24. states : stores the list of states that we visit in the event of execution of the program

25. seconds : Something required for timekeeping of the simulation phase
26. calculation_time : Same as above
27. max_moves : Used for keeping a bound on the number of moves that are simulated
28. wins : Dictionary which stores as keys the tuples (tuples of tuples specifying the state of the game, number of wins while simulating the game through this node)
29. plays : Dictionary which stores as keys the tuples (tuples of tuples specifying the state of the game, number of plays while simulating the game through this node)
30. history : Stores the history of the moves made, as a list of states
31. Update : Function to update the state of the board in MonteCarlo class
32. get_play : Function which returns the move from the current state of the game for the AI player, using the UCT algorithm, this also calls the simulation function for simulating up to the given time limit or the maximum moves specified by the user
33. run_simulation : Simulates the game, using UCB1 algorithm, starting from the current board state