Game Tree: Checkers

Nick Gilliam Ronald Stempien

December 8, 2019

Checkers is a fun game. It's like Chess, but easier. The only issue is that in order to play Checkers, you have to have friends that want to play Checkers. To complicate things further, you need to have friends in the first place, which is a lot to ask of a computer science student. This leaves us with only one option: to make an artificial intelligence to play Checkers with.

For board games like Checkers, Chess, Othello, et cetera, most AIs utilize a data structure called a "game tree". A game tree is a tree of possible states in a game. Each node in the tree would be a possible game state, with each edge being a possible move from one state to the next. There are two types of game trees, complete and partial. A complete game tree is a game tree that has all possible game states that can occur from the starting position of the game. While a partial game tree will only have a certain amount of possible game states from a given node. Each leaf on a complete game tree would be an ending state for a game (win, loss, tie, et cetera), where a partial game tree may have leaves that are incomplete game states.

When we have a partial game tree, we are able to create engines that will play the game. An engine works by giving each move a rating, using an evaluation function. Using these ratings, we can determine which move is best by selecting the highest rating among a node's children. The minimax algorithm is a popular implementation in a two player game. It works by having one of the players be the "maximizing" player and their opponent be the "minimizing" player. Every level of the tree alternates players, so if it's the players turn, it will select the move with the highest score. Therefore, the opponent's next turn will select the move with the lowest score. The score of a node also depends on how close it is to the root.

We simplified the rules of Checkers in order to make it simpler to create a game tree for. First, and most obvious, we shrunk the board size to 6x6 rather than the standard 8x8 board. We also simplified the rules on capturing, where chain captures are not allowed and you must capture if you can. These rules lower the overall complexity of the game, and lower the time it takes to generate a tree and how much memory the tree takes up.

We first created a game tree of Tic-Tac-Toe to get a better understanding of how to construct one for a more complex game like Checkers. There are a few similarities between the two trees - the board state is stored in binary strings and it's populated using a similar recursion method - but it is smaller in size because of the nature of the game. Creating a whole tree with every possible move took little to no time. The maximum depth of this tree would always be 9, since there are only 9 turns in each game. In Checkers, the depth could be infinite if we left two AIs to battle it out. To get around this, we had to limit what moves the AI could make. For example, if there was a capture possible, that would be the only move the AI could make.

The basic unit of our tree is the node. A node is the representation of a board state with other information attached. The most important element of the node is the 3 integers used to

store the current state of the board. One for white pieces, black pieces, and king pieces. Each integer is a binary string 18 bits in length. Each bit represents a black square on a 6x6 Checkers board. A character is used to represent the last player that went. There is also a "Move" object associated to each node. The move object for a node represents the move that had to be made to get from the node's parent to the node itself. The move object holds information about the source and destination of a move, the color of the piece being moved, and whether the move was a capture or not. The node also holds an arraylist of the node's children.

The tree class holds the root node of the tree, difficult of the AI, the engine the AI uses, and a lot of useful functions. The tree is populated using a recursive method - it calls the root's population method, which will create a new node for each possible move up to a certain depth. If a node contains a completed board, then the tree will stop populating. Obviously, the deeper the tree is, the more information the AI has to work with. By limiting the depth, we can control how smart it is.

Since we store the state of the board in binary strings, we had to create methods to make it easier to manipulate. Methods like checking if a piece exists or getting which player is at a certain spot on the board use bit shift operators. Combining these smaller methods makes our code cleaner. When writing more complex methods like one to get all legal moves, all we have to do is subtract a certain number from the piece's current location and check if that space is open.

The populate method is the method used to create the tree. It's a recursive function within the node class. Since we are creating a partial game tree, we are able to set bounds on the run time of the method. There are three conditions upon which the method ends: the nodes depth is larger than the limit (in our case five), the node has reached an end state (white or black win), or there have been too many moves since a piece has been captured (stalemate/tie). From the current node, all the possible children are created based off of the possible moves for the board (this takes $\mathcal{O}(b)$, where b is the board size). It takes linear time to iterate through each possible move for a board $\mathcal{O}(m)$ where m is the number of possible moves for this node. So to generate all the possible moves and loop through them it takes $\mathcal{O}(b+m)$. Then for each move, the populate function is called again. The population function will continue to be called until one of the ending conditions is met, in which the worst case scenario is to exceed the depth limit, d. Meaning that the functions complexity is $\mathcal{O}(b+m*d)$.

The random engine is by far the simplest. It takes linear time, as all it needs to do is generate all the possible moves for a given node and choose one at random, which takes $\mathcal{O}(b)$ time, where b is the size of the board. Our minimax engine is more complicated though. The method is recursive, as it loops through all the children of the current node it is on, evaluate its score (which takes constant time), and then recuses on each child. This is just like preorder traversal, which takes $\mathcal{O}(n)$ time, with n being the size of the game tree.