Using Minimax with Alpha-Beta pruning to play Quarto

Thomas Marstrander

24.10.2012

Overview of my classes:

Map:

This class handles the state (S) of the board, including all its' pieces, the current player, board size and methods for modifying these values. It also contains methods for placing a piece and checking whether the given state is a terminal state.

MiniMaxD:

This class is the core algorithm for minimax. When making an instance of this class you give as arguments the current state of the game (Map), depth to search, the piece to place and which player is performing the minimax search.

The algorithm will then search depth-first through all possible moves, and for each move evaluate whether the new state caused by the move is a terminal state. If it is, then the terminal state is evaluated and propagated upwards to the last state. Here it is evaluated against all the other children of this state, and chosen or discarded depending on whether the state is a maxnode or a minnode, and its' comparison to the other children. If it is not a terminal state, the next child of this state is evaluated using the recursion of the minimax evaluation function until a terminal state is found, or the specified total depth has been reached.

Piece:

Each Piece has a size, shape, color and hole. These values together defines a piece. A piece can be constructed using these values, or a string format : Piece(String s), where s is a 4 bit number describing whether the value is "on" or "off". The format is color ,shape, size, hole, meaning a string s of 0011 = (R*) = a red, circle, big, with hole piece.

Player:

Each player has a playerNumber, for determining when it is his turn to place a piece, a playerMode, which determines the difficulty of a computer opponent(random, novice or minimax d) or whether the player is a human. The player class contains methods for placing a given piece given its' playerMode and chosing a new piece for its' opponent, turning the Map state (S) into the next State (S*).

Run:

This class handles the input from the user.

You can choose which player mode each of the two players will have among:

1. Human
2. Random
3. Novice
4. MiniMax D

If you choose 4 – minimax you will be prompted to type in the depth of your search as well.

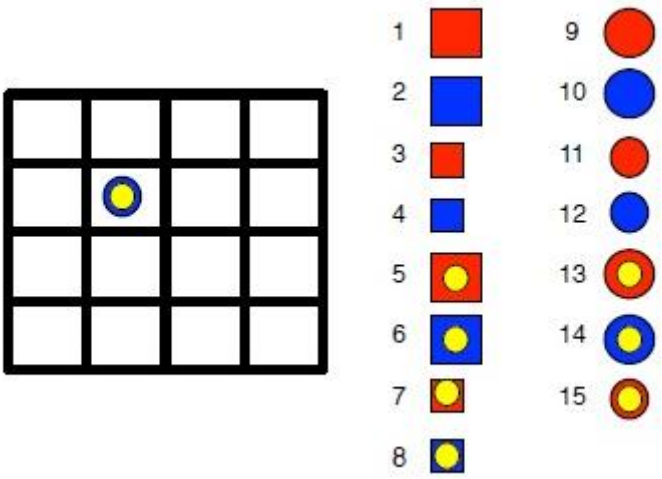
Next you will be able to choose how many simulations you want to perform. The simulations are then executed and the program finishes with a statistic of how many wins, losses and ties there were for the first player.

Evaluation of state function:

General about the algorithm:

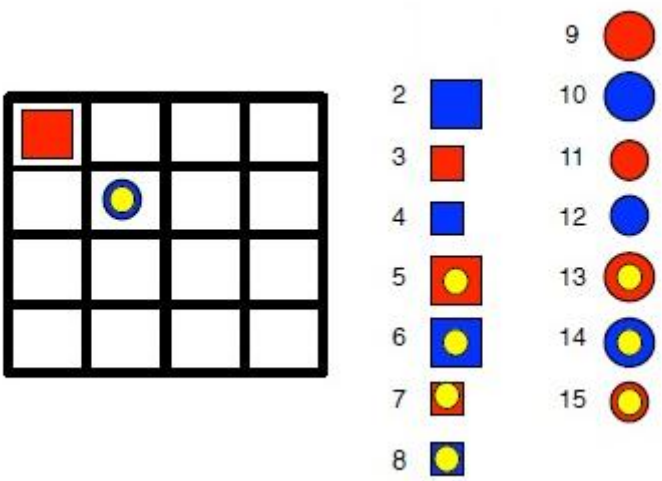
Each state is a combination of pieces placed on the board, and remaining pieces outside the board.

State 1:



The algorithm will first check if the current state is a terminal state. A terminal state is a state where one of the players has won, or the given depth (usually 4) of the minimax algorithm has been reached. If the state is not a terminal state, the evaluation function will generate a child from this state. It does so by choosing one of the remaining pieces outside the board, and placing it on an empty spot on the board, thereby generating a new state.

State 2:



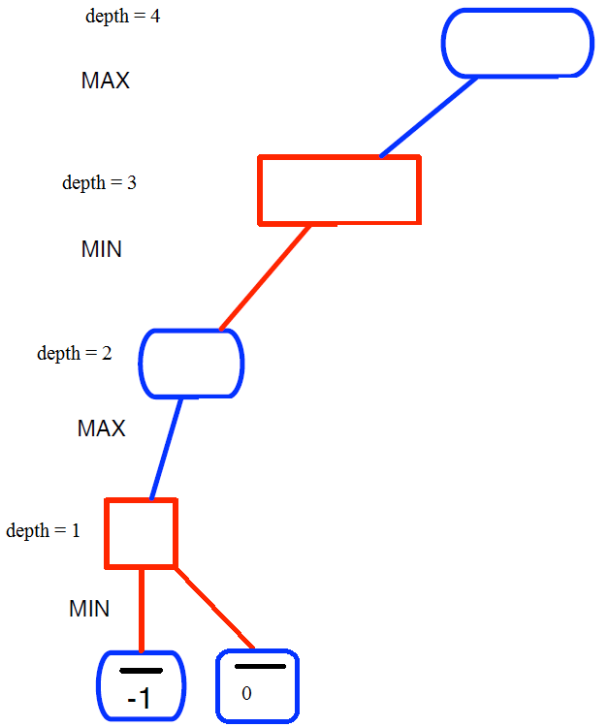
The algorithm is run recursively on the child node until it reaches a terminal node.

Node evaluation:

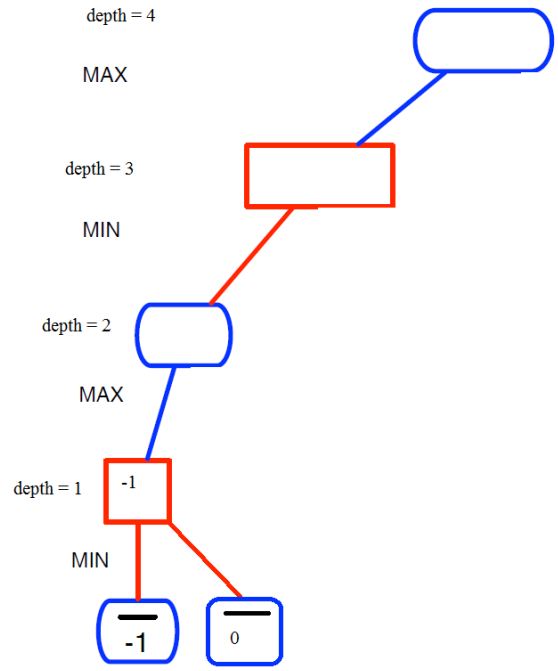
The terminal node is evaluated by checking if the current state is a finished state, that is one of the players has won. If one of the players has won, it figures out who won by seeing which player placed the last piece on the board. If the player who placed the last piece on the board is the player which is running the evaluation function/the maxing player, a positive value is returned, otherwise if the other player won, a negative value is returned. If none of the players has won, that is if the depth of the minimax searching algorithm has been reached, or the board is full/no pieces left, the evaluating function returns 0.

The positive and negative values that are returned are a summation of 1 or -1 and the positive or negative value of the current depth, which makes the early termination states where max player wins the most desirable, and the early states where the max player loses least desirable.

Each state is either a max node or a min node, starting with a max node at the first state of the algorithm and then alternating between min node and max node. When at a max node the state will choose the child node which has the highest terminal value, and the min node will choose his child with the lowest terminal value. This is because we are expecting our opponent to play perfectly, as well. The state will update its' own value to the optimization of its' children this way, and when it has gotten a terminal value from all of its' children, the state will update its' own value and return this to his parental state. This way values are propagated upwards in the search tree.



In this figure two terminal state has been reached, and evaluated, its' parent node, being a min node, will the choose the minimum of these children and propagate it upwards, updating the search tree:



This is how the evaluation of nodes works and how they are returned up through the search tree until you get a final value at the start node/first max node.

Opponent piece:

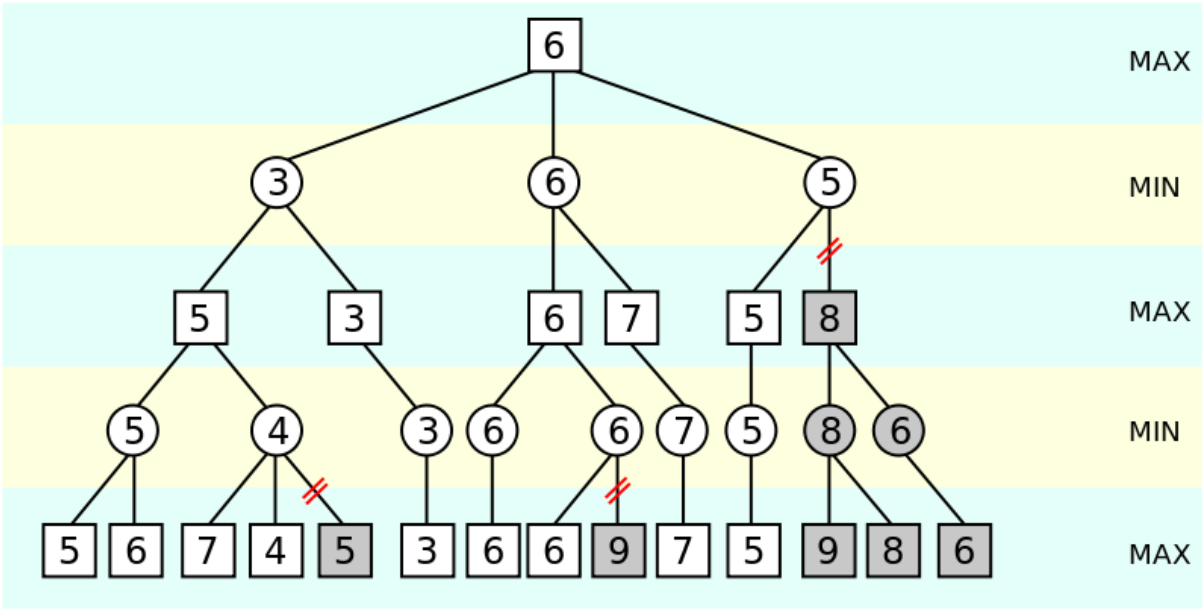
Since you need to end up with both a spot on the board where you want to place your given piece and to choose a piece for your opponent I decided to make a local class NextPiece within my

algorithm for storing x and y coordinates of current move, the heuristic evaluated value and the next opponent piece. This class is the return value of each node in my search tree and the opponent piece is updated for each node until it gets to the final max node, where the heuristic evaluation value is the same as originally evaluated at the bottom node, but the x and y coordinates of the given piece is updated, as well as the piece that you want to give to your opponent. The evaluated value is the one displayed in the figures, and at the final max node, these values are returned to the player class and used to make the optimal move for our max player.

Alpha-beta pruning:

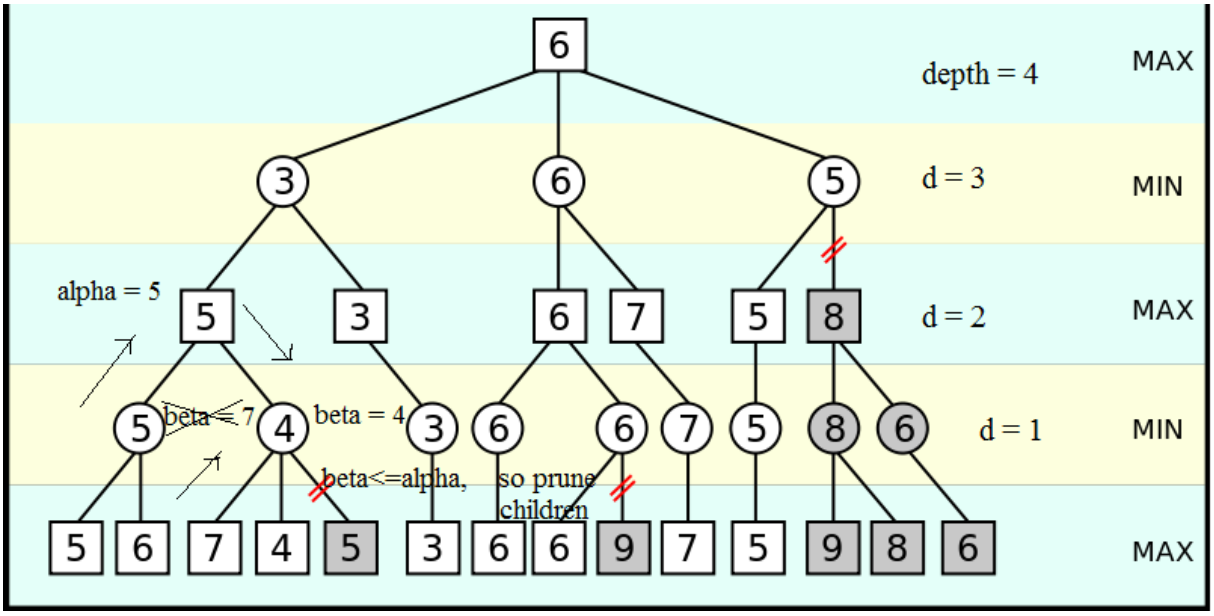
When searching through the tree, given the tree from our previous figure the depth=2 node will evaluate to -1, if we imagine that this max node has another child, a min node at depth = 1, and that min node child has another child which evaluates to -2. Now let us imagine that the min node at depth = 1 has another child, this one would not have to be evaluated, because if the node evaluates to greater than -2 it will not be chosen by the min node, and if it evaluates to a number less than -2, it will not be chosen by the max node at depth = 2, because this node will choose the child node from our figure, which evaluates to -1, because it is greater than -2. This way we can prune all other childs from the imagined min node at depth = 1.

In this figure all the pruned branches are greyed out, to illustrate how we can save time and workload for our algorithm:



In my algorithm this is solved by having each call to the recursive state evaluating function pass an alpha and beta integer value. For the final max node, these values start at alpha as minimum integer value and beta as maximum integer value, and is updated when it reaches the first terminal node, and then propagated upwards through the algorithm, updating each min node with the beta value and each max node with the alpha value. For instance let's use the figure to explain how it would work in my algorithm, using the first greyed out node in our figure. The max node at depth = 2 has been evaluated to alpha = 5, this value is passed to the min node at depth = 1. Beta is at this moment +infinity, and is updated to 7 when we evaluate the first child at depth = 0 and send this value to the

min node at depth = 1. A check to see if we can prune the rest of the children is then done by checking if beta is equal or less than alpha. Since 7 is not less than or equal to 5, we have to check the next child of the min node. Since this evaluates to 4, the beta value at our min node is updated to 4. When we check whether beta is less than or equal to alpha, we get that this is true, and we do not have to check any of the other children of our min node.



100 random vs. novice:

Player 1 is a random player and player 2 is the novice player.

Player 1 (random) total wins:	2
Player 2 (novice) total wins:	98
Total ties:	0

20 runs novice vs. minimax-3 agent:

Player 1 is novice and player 2 is minimax-3:

Player 1 (novice) total wins:	1
Player 2 (minimax-3) total wins:	18
Total ties:	1

20 runs minimax-3 vs. minimax-4:

Player 1 is minimax-3 and player 2 is minimax-4:

Player 1 (minimax-3) total wins:	0
Player 2 (minimax-4) total wins:	20

Total ties:	0