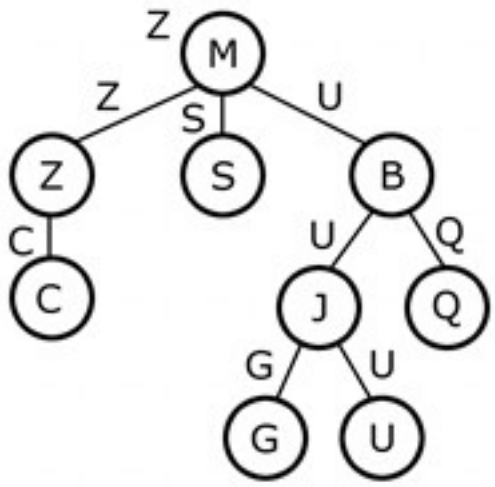
CSCI4110: High Performance Computing

Implement TicTacToe MiniMax Search in Parallel using Java 8. Binary Tree/General Tree

Consider a general binary tree (an unorganized binary tree, not a binary search tree). You are familiar with the postorder traversal of a general binary tree where both of the children (if the node has both a left and a right child) of a particular node are visited before the node itself is visited. Suppose you wanted to find the largest item stored in the tree. For a given subtree in the binary tree, you make a recursive call on the left child to get the largest item in the left subtree and a recursive call on the right child to get the largest item in the right subtree. The larger of these two items is compared to the item stored in the parent of the subtree to find the overall largest item in the subtree. The largest item from that subtree is then returned to its parent for similar tests, and so forth until the largest item is returned from the root node.

Now consider that each node can have any number of children, not just two or fewer as in a binary tree. This is a general tree, but finding the largest item in this tree is still a postorder traversal as discussed above. Theonly difference is that a parent will need tostore the pointers to its children in a list or asimilar data structure. Recursive calls aremade on all the children, identifying thelargest item amongst them all.

Now suppose the general tree only stores items in the leaves, rather than at each node. This simplifies the postorder traversal for the largest item as now the traversal simply returns the largest item amongst the children rather than comparing this item to the item stored at the parent node as well.



MiniMax Tree

Consider a very simple two player game where "Max" can make one of three possible moves (A, B, or C). If Max selects move A, its opponent, "Min", can respond with

three moves of its own (X, Y, Z). After Min takes its turn, the game is over, and the score of the game can be determined. Max is trying to maximize the score while Min is attempting to minimize the score. Suppose the results for Min's three possible moves when Max has selected move A are (9, 5, 11). Thus, if Max selects move A, Min should select move Y to minimize

the opponent's possible score. This isthe optimal move for Min to take in thissituation. For Max to find its optimalmove, it needs to figure out how Minwill respond to each of its possiblemoves (A, B, or C) and

select the movegenerating the maximum possible result.This analysis works when it is Min's turn as well, except that Min will be selecting the smallest value from its possible moves.

Of course, in selecting its move, Max is assuming that Min will play optimally, which may not be the case. Thus, in games that have many more turns for each player, Max must recompute its optimal move every time it is Max's turn to be sure that it is always maximizing its possible score for the current state of the game which can include suboptimal moves by Min.

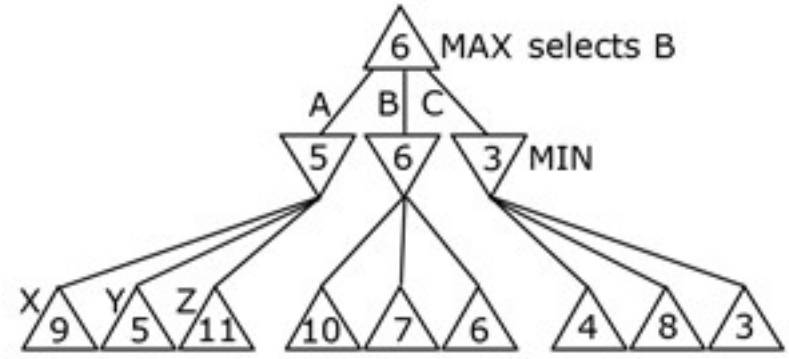
A MiniMax Tree is a tree designed to find the optimal move in a two player game such as Tic-Tac-Toe, Othello, Checkers, etc. Each node in the tree represents a move by Max or Min, the two players in the game.

Consider a two player game as a general tree where all the possible game scores representing the different choices that Max and Min can make are stored in the leaves of the tree. If it is Max's turn to select a move, the optimal move that Max should make is a postorder traversal of the tree where selecting the maximum of the children or the minimum of the children alternates as you move through the levels of the tree.

Tic-Tac-Toe MiniMax

The goal is to have a computer playing Tic-Tac-Toe optimally as either X or O. Let's use the **MiniMax Tree** scheme to achieve this.

If Max is the Xs player and Min is the Os player, the MiniMax Tree is a general tree where the root node, an empty 3x3 grid, is Max's turn as Xs go first in Tic-Tac-Toe. The root node has 9 children, and Max must eventually examine all 9 to find its best move. First, Max places an "X" in the (1,1) location of the grid. Now Min ("O") has eight locations that it can move. This process of playing the game continues until a leaf is reached and a value is assigned to that leaf, called a terminal state.



Note that the "general tree" is represented by the 3x3 Tic-Tac-Toe grid being filled up as moves are taken by Max and Min.

With Max as Xs, then the game outcome at the terminal states of the MiniMax Tree is positive if Max has three Xs in a row, negative if Min has three Os in a row, and 0 if neither Xs nor Os has three in a row and all 9 spaces of the board are occupied. Further, if Max determines it can win by traversing the MiniMax Tree, Max should be encouraged to complete its three Xs in a row as quickly as possible. The value (10 - turn) as the game outcome where turn is the depth of the MiniMax Tree (that is, how many turns have

been taken by both players) will achieve this. For example, a win by Max in 5 turns is valued as +5 while a win by Max in 9 turns is +1.

As discussed as a postorder traversal, Nonterminal/nonleaf nodes are either selecting the largest values returned by their children or the smallest, depending on whether it was Xs turn or Os turn as that node. The selected value is returned to the parent once all children have been analyzed. Eventually, a result makes it to the root. Now Max ("X") knows that it can win with a move in the (1,1) location on the grid. However, X may be able to win more quickly in some other location, so X must still examine all possible places that it can move. Thus, X next tries the (1,2) location in the grid. This requires that moves be removed from the board once analysis of that move is completed so that the grid is once again empty for Xs next move analysis. Also, numerous copies of the board/grid are created so that the branches do not overwrite one another's selected moves.

Once X knows the best place for its initial move, it sends the move to the Tic-Tac-Toe game, and that move becomes permanent. Now O takes its turn at the root and finds its best choice, but X already occupies one spot on the board, so O only has to consider 8 moves. Thus, the computer player can find its optimal move whether it is playing Xs or Os (or both).

Parallel Tic-Tac-Toe MiniMax

Even a simple game like Tic-Tac-Toe can result in an extremely large MiniMax Tree. It is possible to take advantage of parallelism in a MiniMax Tree traversal to speed up the computation. A postorder traversal does not care the order in which the children are visited, it only matters that all children are visited so all of the results can be compared to one another before the traversal gives the result to the parent.

Thus, when X is deciding on its first move, we can start up several (up to 9) separate threads (**fork**) to potentially speed up the speed of the traversal by a factor of 9. These threads will each take a different amount of time to finish as the depth of the minimax tree depends on the location of the test move. Therefore, we must wait (**join**) for all of the threads to finish before returning the result.

The order that threads finish is output in the console. For the serial case, these values are in ascending order. For the parallel case, any thread can finish at any time. The sequence will rarely repeat. It does appear that certain threads generally take longer than others. Why?

Parallelize

With Java 8, it is straightforward to convert the above algorithm into one that automatically uses multiple processors if available. However, we must identify the part of the algorithm that can safely be run in parallel. For TicTacToe MiniMax, the order that we examine each spot that max (or min) can select on their turn does not matter. However, we must wait for the analysis of all of the possible moves to complete before reporting which one is best. When we allow each possible move to be processed simultaneously (if processors are available), we refer to this as a **fork**. When we wait for the subtasks to complete before reporting the optimal choice, we refer to this as **join**. Thus, only after all of the possible move analyses have begun (fork) should we start to wait for them to complete and compare the results to one another (join).