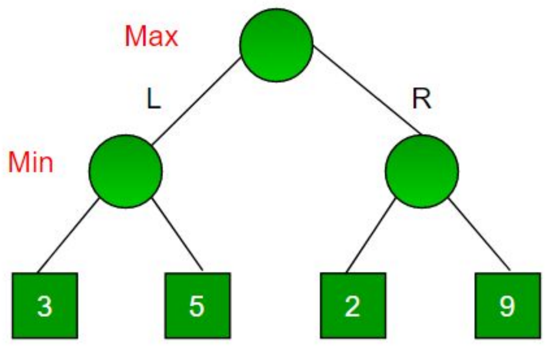
Parallel Minimax In Haskell and Its Applications

COMS4995_003 Final Project Proposal

Gabriel Clinger (gc2821) - Matthew Ottomano (mro2120)

Minimax

Minimax is an algorithm used in games to create intelligent AI's. The premise of the algorithm is the maximizing and minimizing nature of players versing each other. Minimax assumes that there are two players and gives the roles of maximizer and minimizer. The job of the maximizer is to make the decision which leads to the maximum score whereas the minimizer must do the opposite. This is visualized clearly using a tree:



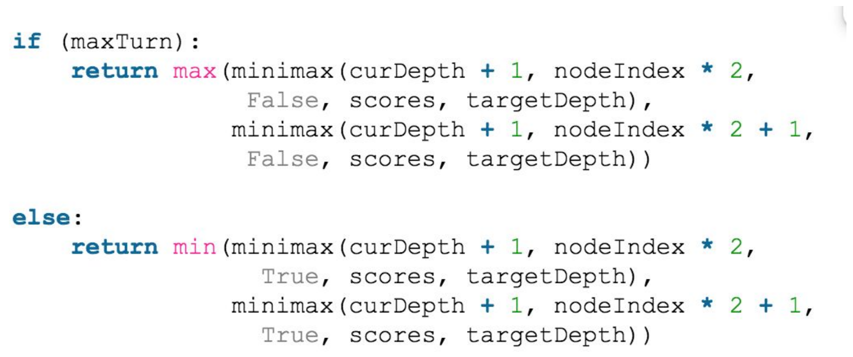
In the tree above, it may feel natural to the maximizer to try and get that score of 9 on the right-most leaf, however using the tree, the maximizer understands that the minimizer would choose the lesser of the two leaves. Therefore, instead of the desired 9, the minimizer would choose the 2. Using this same logic, if the maximizer chose L, then the minimizer would choose the lesser of the two leaves, 3 and 5, resulting in a 3. This is greater than the score of 2 from choosing R, therefore, the maximizer chooses L. This is a very simple example of how minimax works.

Alpha beta pruning

Minimax using Alpha beta pruning is a method of decreasing the number of nodes that are evaluated by the minimax algorithm in its search tree. Alpha beta pruning decreases the branching factor thus allowing for a faster and deeper search. We will use this optimization method to hopefully be able to search the full depth of each tree in a reasonable time.

Parallel Minimax

This project will focus on parallelizing the minimax algorithm in Haskell and analyzing metrics to understand the difference in performance from a sequential implementation to a parallel one. In order to understand the nature of minimax, one must understand its recursive nature. The code snippet of python below shows this clearly:



```
if (maxTurn):
    return max(minimax(curDepth + 1, nodeIndex * 2,
                False, scores, targetDepth),
               minimax(curDepth + 1, nodeIndex * 2 + 1,
                False, scores, targetDepth))

else:
    return min(minimax(curDepth + 1, nodeIndex * 2,
                 True, scores, targetDepth),
               minimax(curDepth + 1, nodeIndex * 2 + 1,
                 True, scores, targetDepth))
```

For both the turn of the maximizer and minimizer, minimax is called recursively until a target depth is reached. The recursive calls are embarrassingly parallel since using something like the par monad would allow this recursive function to be forked at every call, creating a thread for each new step within the tree. Other elements such as generating the moves of a player using heuristics exist and may be parallelizable, however this project will focus on parallelizing the tree traversal.

Applications of Minimax

The applications of minimax are found in games such as chess, 2048, connect 4 and even tic-tac-toe. Minimax can be used to create intelligent agents in these games. To limit the scope of this project, we will not be implementing the games listed above in Haskell. Instead, we will choose a subset of these games to test our minimax function on, using existing codebases from Hackage and GitHub.