Artificial Intelligence PLH311
Second Programming Assignment

Team LAB31146761
Diamantis Rafail Papadam
Alexandros Konstantinos Valtatzis

April 5, 2021

Minimax

The standard minimax algorithm is basically a brute-force search of all the possible chessboard states after a specific number of moves (depth of the search tree), or more generally after a cutoff function of the depth and the state returns *true*. Each node can represent either the white player or the black player.

Minimax assumes that the opponent always selects the seemingly best move, so working our way from the leafs to the root, we keep the minimax value of the child that minimizes it for a white node, whereas we keep the minimax value of the child that maximizes it for a black node. If the node is a leaf, meaning that the cutoff function mentioned above returns *true*, then we assign to that leaf the result of the evaluation function. The cutoff function we used is simply checking whether we have exceeded maxDepth which is set to 6. As for the evaluation function, we used the following:

$$evaluation(state) = wp + wpc \times 0.9 - bp - bpc \times 0.9$$

where:

$wp =$ total value of white pieces remaining.

$bp =$ total value of black pieces remaining.

$wpc =$ presents collected by white from root to this state.

$bpc =$ presents collected by black from root to this state.

The number of presents collected by both players is known by the recursive calls of minimax. It is important to mention that while searching states on the search tree by expanding it, we never added new presents since that's a part of the randomness of the game that we cannot predict in any way. Also, something technical worth mentioning is that we didn't actually count the number of times each player collected presents, but rather added 0.9 to what we called *bonus* whenever a white player collected a present at a path of the tree, and subtracted 0.9 whenever a black player did. A final detail about the evaluation function we used is the fact that we didn't take into consideration the "+1 *score*" that a pawn which reaches the last lair gets but neither did we remove that pawn from our simulated chessboard. In others words, we didn't lose 1 *point* by removing the pawn from the board but neither did we gain 1 *point* by reaching the last lair. We believe that what we did is practically equivalent to the other option.

Forward Pruning & Singular Extensions

In this part of our assignment we were required to enhance the minimax algorithm by implementing forward pruning and singular extensions. The reason that these two methods improve the efficiency of the algorithm is because they allow us to look deeper by intelligently pruning the tree. Forward pruning works in the following manner: The possible moves are sorted from the seemingly most efficient (the one that immediately gives us the most points) to the seemingly least efficient (usually a move that gives us 0 points in the short-term). Then according to how deep we are in the tree, we prune a percentage of the seemingly less good moves and only search the better ones. Singular extensions is basically the same thing, we could describe it as extreme forward pruning because the tree is only expanded on the most promising move.

In our assignment we didn't use forward pruning or singular extensions in minimax, but we used this enhancement in ab-pruning minimax because it is superior to minimax anyway.

What we did there, is define a *pruneDepth* (currently set to 4) as well as a *singExtDepth* (currently set to 9) which correspond to the depth where forward pruning starts and the depth where it becomes extreme and turns into singular extensions respectively. Our *abMaxDepth*, meaning the depth where ab-pruning stops the singular extensions, is currently set to 13. The formula that calculates how many moves to prune between *pruneDepth* & *singExtDepth* is the following:

$$\#branchesToPrune = \left\lfloor \#moves \times \frac{depth - pruneceDepth}{singExtDepth - pruneDepth} \right\rceil$$

Alpha–Beta Pruning

Ab-pruning, especially when the moves are considered from seemingly best to seemingly worst, is massively improving the efficiency of minimax as we know from the theory of this course. What ab-pruning does is getting the tree to "remember" the maximum value that we have found for the white player so far in the search (a), as well as the minimum value we have found so far for the black player (b). These 2 values allow us to prune any sub-tree when white is looking to play and his opponent has a move that leads to a minimax value higher than b or black is playing and his opponent has a move that leads to a minimax value lower than a. In an average case scenario this can enable us to look $\frac{4}{3}$ times deeper in the search tree. Also, if the moves are sorted perfectly from best to worse then we can look 2 times deeper than we would with default minimax in the same amount of time.

The maximum depth that our ab-pruning minimax algorithm is searching in is mentioned and explained in detail in the last paragraph of the previous section (Forward Pruning & Singular Extensions).

Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) relies on running a huge number of iterations to give accurate results. What it does on each iteration is to run a simulation of the game on the most promising leaf of the MCTSTree. The default way to run the simulation is by making players make random moves until one of them wins the game, then that changes values on the tree nodes of that path using back-propagation. If we reach a leaf whose simulation has already happened then his children (possible moves from that chess position) are generated and we run a simulation in any of those. The way we find the most promising leaf is by starting from the root and going deeper by choosing the node with the best UCB value at each step. We define:

$$UCB(node) = \frac{w}{n} + \sqrt{2} \times \sqrt{\frac{ln(N)}{n}}$$

where:

$N =$ number of node's parent simulations.

$n =$ number of node simulations.

$w =$ number of node wins.

Comparison between Minimax-MCTS

MCTS seemed to be really inconsistent at $6000ms$ cap, which is $25,000 - 40,000$ iterations on the PC that we used to run it. It seems to us that "raw" MCTS without any modifications is very time consuming and doesn't guarantee optimal moves in games with a huge amount of states like chess. On the other hand, modifications of MCTS, like replacing the simulation step with an evaluation of a smart neural network, should have much more potential to yield very solid moves in a reasonable amount of time.

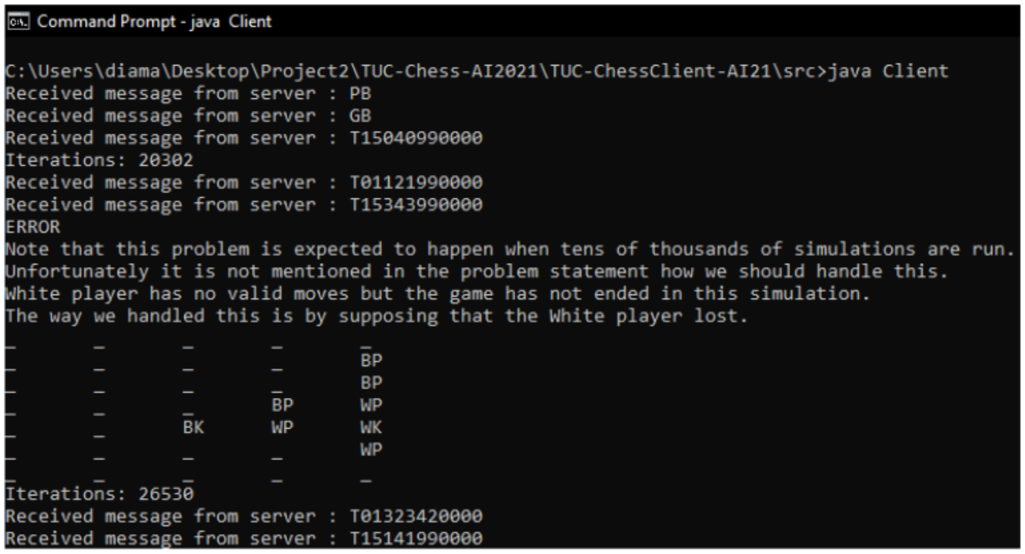
Having said the above, it is needless to say that even the default minimax with no forward pruning, ab pruning, or any other enhancement, completely obliterated MCTS by just looking 4 moves ahead. MCTS though, was able to beat the randomized agent given to us pretty easily and consistently; even by running a couple thousand iterations which takes less than a second on any modern laptop.

To conclude, the best algorithm we managed to create is $abMinimax()$ and it is called from the $Client.java$ file by choosing $world.selectMinimaxAction(true)$, where true refers to the ab-pruning.

Notes

Special cases that weren't specified in the assignment.

We only found one special case that wasn't mentioned in the problem statement. This case is known as stalemate on the real game of chess and leads to a draw. Here on the other hand, we supposed that the person that's unable to move should lose. The error message on the image below explains exactly what is happening on a node of the search tree but we turned those messages off because on an algorithm like MCTS the Client would get spammed.



```
Command Prompt - java  Client

C:\Users\diama\Desktop\Project2\TUC-Chess-AI2021\TUC-ChessClient-AI21\src>java Client
Received message from server : PB
Received message from server : GB
Received message from server : T15040990000
Iterations: 20302
Received message from server : T01121990000
Received message from server : T15343990000
ERROR
Note that this problem is expected to happen when tens of thousands of simulations are run.
Unfortunately it is not mentioned in the problem statement how we should handle this.
White player has no valid moves but the game has not ended in this simulation.
The way we handled this is by supposing that the White player lost.
_       _       _       _       _
_       _       _       _       BP
_       _       _       _       BP
_       _       _       BP      WP
_       _       BK      WP      WK
_       _       _       _       WP
_       _       _       _       _
Iterations: 26530
Received message from server : T01323420000
Received message from server : T15141990000
```

Extra things we did.

1) We sorted the available moves from most promising to least promising so that ab-pruning as well as forward pruning and singular extensions become more efficient.

2) We used a hashmap in *World.java* to store moves, using chess positions as keys. If a certain position re-occurs then we move immediately by reading the result from the hashmap instead of running the chosen algorithm again and wasting 6 second.

Things we would work on if given more time.

1) We would experiment further with evaluation functions and try to implement a good heuristic evaluation of a particular TUC-chess position. Having done that, we wouldn't simply count the pieces remaining to evaluate a certain position, but also give it a score based on the positional structure and how promising it looks. This would affect the evaluation by $x\%$ and our original evaluation function would affect the improved evaluation by $(100 - x)\%$. The value of $x \in \{1, 2, ...99\}$ would then be chosen experimentally by running minimax algorithms with different evaluation functions against each other.

2) Another thing we would definitely proceed to doing is creating what is called a transposition table. That table would have a memory limit and it would work like a cache, keeping in memory the most recent chess positions visited by an execution of minimax. If at another point on the same execution of minimax, the same chessboard state is visited, obviously through a different sequence of moves, then we would simply read the evaluation of that position from the transposition table and avoid doing the work twice or even multiple times.

Sources

- Lecture Slides & AI a modern approach by Suart Russell & Peter Norvig.
- Minimax.
- Ab-pruning.
- Understanding Minimax & ab-pruning deeper through an example.
- Tree data structure used for MCTS.
- Information about MCTS.
- Understanding MCTS deeper through an example.