Chess game

Main Classes:-

Board 🡺 contains all graphical functions

contains 8\*8 cells

Cell 🡺 a StackPane that contains a Squared Rectangle & an Image

* A cell may or may not have an image, but it must contain a Rectangle.

ChessGame class 🡺 is the launcher class

Ref 🡺 a class for static variables & constants

Pieces classes:-

Every piece class contains the movement functions of the it’s character whether it’s black or white except for BlackPawn & WhitePawn.

AI algorithm

The other side:

the other side moves would partially depend on my moves.

For instance-

* if “my move has **no threats** on him”

then “he would move a **random move** or **threaten me** “if he can””

* if “my move **threatens him**”

then “he must check for **4 solutions**”

* 1st solution is the easiest & that to move to another safe place “if there is”
* 2nd solution is to sacrifice himself in order to protect those who are more important on his side “if there is”
* 3rd solution is to sacrifice himself in order to kill me then I can kill him “if he can”
* 4th solution is to threaten me & eliminate the threat “if he can”
* If “both are **threatening each other**”

then “he must check for **2 solutions**”

* 1st solution if he can protect himself & still threaten me “if he can”
* 2nd is to sacrifice himself & eat me in order to eliminate the threat “if he can”.

So we would have a new class

Checker class 🡺 tests the state of the pieces in order to determine what sort of move he will take & to invoke that move we use a second class. It contains the methods of the other side player.

Main Algorithm

* Movement-Generation
* Board Evaluation
* MiniMax
* Alpha-Beta pruning.

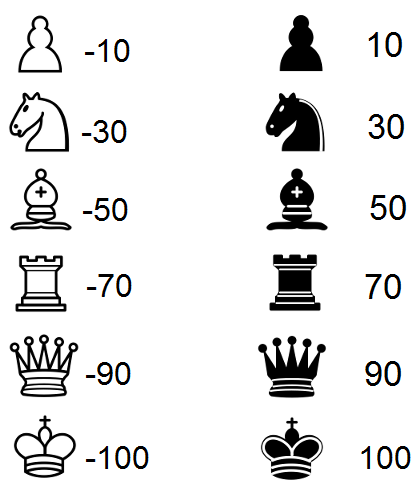
At each step, we’ll improve our algorithm with one of these time-tested chess-programming techniques.

**Step 1: Move generation and board visualization**

We start building controllers for visualizing the board. The move generation controllers basically implement all the rules of chess. Based on this, we can calculate all legal moves for a given board state.

Although this algorithm isn’t a very solid chess player, it’s a good starting point, as we can actually play against it.

**Step 2: Position evaluation**

Now let’s try to understand which side is stronger in a certain position. The simplest way to achieve this is to count the relative strength of the pieces on the board using the following table:

Now we’re able to create an algorithm that chooses the move that gives the highest evaluation

The only tangible improvement is that our algorithm will now capture a piece if it can.

**Step 3: Search tree using MiniMax**

Next we’re going to create a search tree from which the algorithm can choose the best move. This is done by using the MiniMax algorithm.

In this algorithm, the recursive tree of all possible moves is explored to a given depth, and the position is evaluated at the ending “leaves” of the tree.

After that, we return either the smallest or the largest value of the child to the parent node, depending on whether it’s a white or black to move. (That is, we try to either minimize or maximize the outcome at each level.)

With MiniMax in place, our algorithm is starting to understand some basic tactics of chess.

The effectiveness of the MiniMax algorithm is heavily based on the search depth we can achieve. This is something we’ll improve in the following step.

**Step 4: Alpha-beta pruning**

Alpha-beta pruning is an optimization method to the MiniMax algorithm that allows us to disregard some branches in the search tree. This helps us evaluate the MiniMax search tree much deeper, while using the same resources.

The alpha-beta pruning is based on the situation where we can stop evaluating a part of the search tree if we find a move that leads to a worse situation than a previously discovered move.

The alpha-beta pruning does not influence the outcome of the MiniMax algorithm — it only makes it faster.

The alpha-beta algorithm also is more efficient if we happen to visit **first**those paths that lead to good moves.

With alpha-beta, we get a significant boost to the MiniMax algorithm.

**Conclusion**

The strength of even a simple chess-playing algorithm is that it doesn’t make stupid mistakes. This said, it still lacks strategic understanding.

With these methods here, we’ve been able to program a chess-playing-algorithm that can play basic chess. The “AI-part” (move-generation excluded) of the final algorithm is just 200 lines of code, meaning the basic concepts are quite simple to implement.