# Possible improvements Or Things We could Have Implemented

## *We could have implemented:

### 1)Null Move Heuristic

This simple heuristic improves the *beginning* of the *alpha-beta search*. Initially, there are no values for what the worst and best possible moves are, so they default to negative and positive infinity respectively. But using this heuristic the algorithm can *find an initial lower bound on the best moves*. It lets the opposing player play two moves in sequence (choosing them based on a small-ply min-max search), and computes the score after that. Certainly, any move made by the current player should beat a score obtainable by the opponent getting two chances to move.

### 2)Quiescence Searching

Since the depth of the min-max search is limited, problems can occur at the frontier. A move that may seem great may actually be a disaster because of something that could happen on the very next move. Looking at all these possibilities would mean increasing the depth of mini-max by 1, which is not the solution, as we would need to extend it to arbitrarily large depths. The goal is thus to search the tree until *"quiescent"* positions are found - i.e ones that don't affect the current positions too much (most maneuvers in chess result in only slight

advantages or disadvantages to each player, not big ones at once). Hence, *looking at higher depths is important only for significant moves - such as captures*. Consider for example a move in which you capture the opponent's knight with your queen. If that is the limit of your min-max search, it seems to be a great move - you receive points for capturing the opponent's knight. But suppose that in the very next move your opponent can capture your queen. Then the move is clearly seen as bad, as trading a queen for a knight is to your disadvantage. Quiescence searching will be able to detect that by looking at the next move. Again, it doesn't need to do this for every move - just for ones that affect the score a lot (like captures). One important caveat in the quiescence searching algorithm is that it should only look at moves that became available because of the current move being made. Consider the following situation. Your bishop is threatened by an opponent's pawn, and you have the ability to capture the opponent's knight with a different pawn. Suppose the algorithm is looking only 1 level ahead, and is examining some non-capturing move. It won't notice that the bishop can be captured in the next turn. But what happens when it's examining the knight-capturing move with quiescence. It will see that the opponent can take your bishop, which will even out the piece possession, making the move not seem as good.
So it's highly likely that the algorithm would pick a move other than capturing the knight, thus needlessly losing the

bishop in the next turn.
To prevent this, the algorithm must look at ***ONLY*** those moves available because of its own move. Since the opponent's "pawn captures bishop" was available regardless of whether you capture the knight or not, it should be ignored.

**3)Iterative deepening**

Instead of searching to full depth right away, *iterative deepening increases* the search *depth* by one at a time. This counter-intuitive approach allows to store useful information from earlier iterations to be used in later iterations to increase efficiency and it helps to control the time spent for a move decision, since search times for a fixed search depth vary significantly for different problems. This is how practical implementations of alpha-beta address the issue of Stopping. This is by no means an optimal solution, since it only helps to control that a set time limit is not exceeded.

**4)Move ordering**

Alpha-beta is most efficient if the best successor of each node is searched first. Information stored from previous iterations helps to achieve nearly optimal move ordering. A good move ordering allows the search to address the question about which node to Expand Next.

**5)Transposition Tables**

Transposition tables aid in the above mentioned task of move ordering by storing information from iteration to iteration. Furthermore, they allow for the detection of transpositions (a

position in a search that can be reached by different move sequences) and thus eliminate duplicate search effort. Partition Search generalizes the concept of transposition tables. Instead of storing information for individual positions, information for sets of positions is stored, increasing the usefulness of transposition tables. Ginsberg could show the merits of Partition Search for bridge, but no successful attempt to apply this algorithm to chess has been reported.

### 6)Search Extensions

Domain-dependent (such as check extensions) and domain-independent (singular extensions) knowledge can be used to increase the search depth for certain lines of play. Search extensions also address the Bad Line problem.

### 7)Null-Window Searches

The narrower the search window, the more efficient alpha-beta is. Null-window searches (alpa + 1 = beta) combined with a good move ordering improve the efficiency of alpha-beta, effectively pruning or limiting the amount of search effort spent in “bad" lines and thus tackling the Bad Line problem.

# Possible improvements to the already existing code

### 1) Initial Fixed Moves

As we can easily observe and guess that in the initial board we have the same arrangement of the 16 pieces of each side

then we can really assume some initial moves to be quite guessable. With this assumption what I want to say is that I can really store some initial possible configurations and their corresponding scores as calculated by the evaluation function in a file and use it each time a new game is started. This will do occupy some storage but will help me to optimize my cpu running time by some small factor(The complexity still remains the same but the average cpu time will reduce by a constant).

**2) Evaluation Function**

The evaluation function as used in the AI written by us has no such modifications but just use the predefined things, that have already been done in the already existing world. We will here like to propose some changes to the evaluation function. One of the changes that can be proposed is to consider the expectation of the number of wins a particular move will lead to but we cannot actually evaluate all the moves possible due to our computational and hardware limits. But what we can do is we can actually keep the current evaluation for the last step of our mini-max algorithm. So basically the branch that will to the maximum average of the score at its base will be the selected branch. One disadvantage of this is that in this approach the actual best move that could have been taken by the user may not lie in the selected branch as we are now considering the average of the scores at the highest level of the depth of the mini-max algorithm. This method is inspired from the monte-carlo tree search. We can actually mix up all of the current evaluation functions available in the market and

use the best parts of each of them to come up with a new much better evaluation function.

## Some Improvements That are Already There

**1)sortMoves():** This is a function implemented in the AlphaBetaChess.java file, this function selects some moves from the already existing set of all possible moves and sort them according to the score they all will lead to, in this way we first sort all the moves and in the mini-max we go like choosing the best move from the now available sorted list of moves. Thus this improves my running speed probably again by a constant additive factor but in some cases the effect may be significant.