# Interface Implement

In class ControlGame, there are four methods.

| | |
|---|---|
| playChess() | Call playChess() for initialize. |
| moveChess() | Call moveChess() when the player move a chess. |
| responseMoveChess() | Call responseMoveChess() when the AI move a chess. |
| printBoard() | Call printBoard() when a determined move needs update GUI. |

# Algorithms

## AlphaBeta Search(History Table Heuristic)

AlphaBeta search is very similarly to MaxMin search. However, it is an effective optimizing. The most different is Beta Pruning. Let's see how it works.

Now, it's my turn. Alpha is the best case for me, Beta is the worst case for enemy. For enemy's turn, -Beta is his best case, -Alpha is my worst case. By this way, we count enemy's value by calling alphaBeta(-Beta, -Alpha). And get its' opposite number, it's the value for me. So, if it is larger than Beta, it is to say this move is bad enough for enemy, so we return it as the best value for me. This is Beta Pruning. However, if it is less than Beta and larger than Alpha, it is to say this move is better for me, so we update Alpha. At last, if there didn't occur Beta Pruning, we return Alpha as the best value for me.

```
int AlphaBeta(int depth, int alpha, int beta) {
  if (depth == 0) {
    return Evaluate();
  }
  GenerateLegalMoves();
  while (MovesLeft()) {
    MakeNextMove();
    val = -AlphaBeta(depth - 1, -beta, -alpha);
    UnmakeMove();
    if (val >= beta) {
      return beta;
    }
    if (val > alpha) {
      alpha = val;
    }
  }
  return alpha;
}
```

However, the order for all moves influence the running time. If Beta Pruning occurs early, the search runs fast. So, I use History Table Heuristic. I take the experience in International and use the square of depth for heuristic value. Every time find a best move, I record it in the history table. If the value in the table is larger, it is to say it's a good enough move and we choose it before. So, we should try these good move first.

## Iterative Deepening DFS

ID-DFS is effective for control time. Sometimes, we need search a very deep depth to get the best move. Sometimes just one or two depth is enough. If we use a fixed depth, there will be some waste. Or the depth is not enough that we can't get the best move. So, I control the depth from 1 to max depth. When this controlled depth search is finish, if we have find the winner move, break. If the search time is too long, break and take the present best move. Because the shallower depth DFS record a lot of history table data, the deeper depth DFS will run fast. This is an optimizing for running time.

## Quiescent Search(MVV/LVA Heuristic)

History table heuristic is a dynamic heuristic, because its' heuristic value is relative to the past situation. However, there are some quiescent heuristic, the heuristic value is just relative to present situation. MVV/LVA heuristic is one of them. MVV is the Most Valuable Victim. LVA is the Least Valuable Attacker. The heuristic value is LVA-MVV. The larger value, the better eat move.

Quiescent search only generate eat moves, and sort moves by MVV/LVA heuristic. It can solve horizontal effect. When search at the max depth, if we use normal alphaBeta search, it will get a very normal move, sometimes looks fool. If we use quiescent search, the result will be radical. On average, this result will be better.

## Null-Move Forward Pruning

Sometimes, if I don't move and let enemy go continue two steps, my result is still good enough. So, when search, we test null-move, if the result is still good enough to occur Beta Pruning, we needn't search follow situation. The search will run faster. However, if null-move can't occur Beta Pruning, the null-move test is a waste of time. So, we should check some situation that is certainly no Beta Pruning. For example, if my boss is in danger, I must move. If the final phase, move is obviously better than null-move. So, I must move. And if I continue don't move two steps, Beta Pruning hardly occurs. So, I won't do two continue null-move.

## Zobrist Hashing

Zobrist hashing is use to judge repeat situation. Every kind of chess at different location has a random unique chess key number. In order to make sure the key number if random enough and unique, I use RC4 algorithm. This is not important. The importance is how to judge repeat situation. That is zobrist hashing. At First, initial the situation key. When add a piece A at location B, the situation key XOR the chess key of A at location B. Similarly, when delete a piece A at location B, the situation key XOR the chess key of A at location B. Because the key is random and unique enough, the situation key when all A[i] at B[i] can only product by XOR every chess key of A[i] at B[i]. The situation key is just a 32 bits integer. It can be stored in an array. So, we can judge the situation by the situation key.

# Reference Material

http://www.xqbase.com/ PS. I learn algorithms from it. But I write the code by myself.