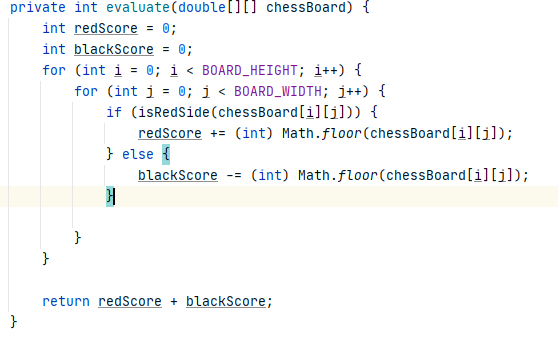
Việc đánh giá một thế cờ là rất quan trọng. Việc này quyết định xem người chơi là cao thủ hay chỉ là gà mờ. Việc đánh giá thế cờ rất phức tạp và khó khăn khi cài đặt. Việc đánh giá thế cờ này chỉ dựa vào kinh nghiệm của người chơi. Nếu chơi nhiều thì sẽ đánh giá được thế cờ một cách chuẩn xác. Để có được kinh nghiệm này thì trên thế giới đã áp dụng được reinforcement learning vào lý thuyết trò chơi. Ở trong chương trình này thì việc đánh giá quân cờ rất đơn giản. Ta sẽ giữ nguyên tất cả giá trị của quân cờ trên bàn cờ Ta lấy tổng giá trị quân cờ của một bên trừ đi cho bên còn lại thì sẽ ra được một giá trị lượng giá tĩnh cho thế cờ. Việc đánh giá chỉ dựa trên giá trị quân cờ sẽ làm cho ta thấy máy chỉ rình rình vồ được con cờ có giá trị cao nhưng đây là cách cài đặt đơn giản và nhanh chóng để xây dựng lên hàm minimax và alpha beta

- Cài đặt hàm đánh giá:



- Như ta thấy thì chỉ việc lấy tổng điểm số quân cờ bên đỏ trừ đi tổng điểm số quân cờ bên đen.

Thuật toán minimax được sử dụng trong chương trình để tìm ra một nước đi là “tối ưu” hay “tốt nhất” cho một thế cờ của máy. Việc tính điểm cho node lá của thuật toán minimax sẽ do hàm lượng giá đảm nhiệm. Như vậy ví dụ thế cờ ban đầu sinh được ra 40 nước đi từ hàm generateMovement(double[][] chessBoard, Movement[] possibleMovement, int level, int[] numberOfBranch, boolean isRed). Ta sẽ thử từ 1 đến 40 nước vừa được sinh từ hàm sinh nước đi. Nếu nước thử cho điểm số cao thì ta sẽ nhận giá trị đó ( ứng với lời gọi hàm là maximum player hay minimum player). Việc đi thử nước rồi rút lại cũng gần giống như cách làm của thuật toán backtracking. Ta kết hợp với minimax và alpha pruning để đi tìm nước đi tối ưu cho một thế cờ.

**Hàm minimax (giải thích bằng comment):**

public int minimax(int depth, double[][] temp, int level, boolean isMaximizingPlayer, int alpha, int beta,  
 boolean isRed) {  
 double[][] chessBoard = new double[BOARD\_HEIGHT][BOARD\_WIDTH];  
 for (int i = 0; i < chessBoard.length; i++) {  
 for (int j = 0; j < chessBoard[i].length; j++) {  
 chessBoard[i][j] = temp[i][j];  
 }  
 }  
 int[] numberOfBranch = new int[50];  
 Movement[] possibleMovement = new Movement[200];  
 if (depth == 0) {  
 return evaluate(chessBoard);  
 }  
 generateMovement(chessBoard, possibleMovement, level, numberOfBranch, isRed);  
 if (isMaximizingPlayer) {  
 int bestValue = -Integer.*MAX\_VALUE*;  
 *// Với các nhánh được tạo từ một thế cờ thì ta đi thử các nước* for (int k = 0; k < numberOfBranch[level]; k++) {  
 *// try* double enemyChessValue = 0;  
 int fromRow = possibleMovement[k].getFromRow();  
 int fromCol = possibleMovement[k].getFromCol();  
 int destRow = possibleMovement[k].getDestRow();  
 int destCol = possibleMovement[k].getDestCol();  
 if (hasEnemy(chessBoard, destRow, destCol, isRed)) {  
 enemyChessValue = chessBoard[destRow][destCol];  
 }  
 chessBoard[destRow][destCol] = chessBoard[fromRow][fromCol];  
 chessBoard[fromRow][fromCol] = EMPTY;  
 *// kiểm tra nước đi thử này nếu tốt qua sông thì gán lại giá trị cho quân tốt* if (possibleMovement[k].isSoliderMovement()) {  
 chessBoard[destRow][destCol] = *SOLIDER\_AFTER\_CROSSING\_RIVER\_WHITE*;  
 }  
  
 int value = 0;  
 *// cắt nhánh nếu ăn được tướng* if (enemyChessValue == *GENERAL\_BLACK*) {  
 value = 999999999 - level;  
 } else {  
 value = minimax(depth - 1, chessBoard, level + 1, false, alpha, beta, !isRed);  
 }  
 *// Cắt tỉa cây dựa vào giá trị alpha, beta* if (value > bestValue) {  
 bestValue = Math.*max*(bestValue, value);  
 *// ở độ sâu ban đầu thì lấy ra nhánh có điểm số tối ưu* if (level == 0) {  
 finalMovement.setFrom(fromRow, fromCol);  
 finalMovement.setDest(destRow, destCol);  
 if (value == 999999999 - level) {  
 finalMovement.setGameEnd(true);  
 } else {  
 finalMovement.setGameEnd(false);  
 }  
 }  
 }  
  
 alpha = Math.*max*(alpha, bestValue);  
  
 *// backtrack* chessBoard[fromRow][fromCol] = chessBoard[destRow][destCol];  
 chessBoard[destRow][destCol] = enemyChessValue;  
 if (possibleMovement[k].isSoliderMovement()) {  
 chessBoard[destRow][destCol] = *SOLDIER\_BEFORE\_CROSSING\_RIVER\_WHITE*;  
 }  
 if (beta <= alpha) {  
 break;  
 }  
 }  
 return bestValue;  
 } else {  
 int worstValue = Integer.*MAX\_VALUE*;  
 *// Với các nhánh được tạo từ một thế cờ thì ta đi thử các nước* for (int k = 0; k < numberOfBranch[level]; k++) {  
 *// try* double enemyChessValue = 0;  
 int fromRow = possibleMovement[k].getFromRow();  
 int fromCol = possibleMovement[k].getFromCol();  
 int destRow = possibleMovement[k].getDestRow();  
 int destCol = possibleMovement[k].getDestCol();  
 if (hasEnemy(chessBoard, destRow, destCol, isRed)) {  
 enemyChessValue = chessBoard[destRow][destCol];  
 }  
 chessBoard[destRow][destCol] = chessBoard[fromRow][fromCol];  
 chessBoard[fromRow][fromCol] = EMPTY;  
 *// kiểm tra nước đi thử này nếu tốt qua sông thì gán lại giá trị cho quân tốt* if (possibleMovement[k].isSoliderMovement()) {  
 chessBoard[destRow][destCol] = *SOLIDER\_AFTER\_CROSSING\_RIVER\_BLACK*;  
 }  
 int value = 0;  
 *// cắt nhánh nếu ăn được tướng* if (enemyChessValue == *GENERAL\_RED*) {  
 value = -999999999 - level;  
 }else{  
 value = minimax(depth - 1, chessBoard, level + 1, true, alpha, beta, !isRed);  
 }  
 *// Cắt tỉa cây dựa vào giá trị alpha, beta* if (value < worstValue) {  
 worstValue = Math.*min*(value, worstValue);  
 }  
 beta = Math.*min*(worstValue, beta);  
  
 *// backtrack* chessBoard[fromRow][fromCol] = chessBoard[destRow][destCol];  
 chessBoard[destRow][destCol] = enemyChessValue;  
 if (possibleMovement[k].isSoliderMovement()) {  
 chessBoard[destRow][destCol] = *SOLDIER\_BEFORE\_CROSSING\_RIVER\_WHITE*;  
 }  
 if (alpha >= beta) {  
 break;  
 }  
 }  
 return worstValue;  
 }  
}

Giã mã của alpha-beta pruning:

**function** alphabeta(node, depth, α, β, maximizingPlayer) **is**

**if** depth = 0 **or** node is a terminal node **then**

**return** evaluate()

**if** maximizingPlayer **then**

value := −∞

**for each** child of node **do**

value := max(value, alphabeta(child, depth − 1, α, β, FALSE))

α := max(α, value)

**if** α ≥ β **then**

**break** *(\* β cutoff \*)*

**return** value

**else**

value := +∞

**for each** child of node **do**

value := min(value, alphabeta(child, depth − 1, α, β, TRUE))

β := min(β, value)

**if** β ≤ α **then**

Thuật toán AlphaBeta nói chung giúp chúng ta tiết kiệm nhiều thời gian so với Minimax mà vẫn đảm bảo kết quả tìm kiếm chính xác. Tuy nhiên lượng tiết kiệm này không ổn định - phụ thuộc vào số nút mà nó cắt bỏ. Trong trường hợp xấu nhất thuật toán không cắt được một nhánh nào và phải xét số nút đúng bằng Minimax. Ta cần đẩy mạnh việc cắt bỏ nhờ đẩy nhanh sự thu hẹp của cửa sổ tìm kiếm alpha - beta. Cửa sổ này được thu hẹp một bước khi gặp một giá trị mới tốt hơn giá trị cũ. Khi gặp giá trị tốt nhất thì cửa sổ này thu hẹp nhất. Do đó nếu càng sớm gặp giá trị tốt nhất thì cửa sổ càng chóng thu hẹp. Như vậy phải làm sao cho các nút ở lá được sắp xếp theo trật tự từ cao xuống thấp. Trật tự này càng tốt bao nhiêu thì thuật toán chạy càng nhanh bấy nhiêu (các công thức về số nút phải lượng giá trong điều kiện lí tưởng ở trên tính được với trật tự là tốt nhất).