Homework2C: Chess Ver2.0

Name	Student ID	Email	Distribution
Lin Liao	0133575	liaolin@cs.washington.edu	Implement the move generator, quiescence search, transposition table using hash scheme.
Tian Sang	0133587	sang@cs.washington.edu	Design and implement the evaluation functions for difference stages.
Ke Zheng	0133614	kzheng@cs.washington.edu	Implement the X-board Protocol with time controller, the whole control of the program, and opening book.

We use C++ language to implement the project. We develop and debug our program on the Visual C++ IDE and use Visual SourceSafe to manage source code and coordinate cooperation. Since we only use standard C/C++ library in our program, except some minor places, no source code was changed when we migrated the project from Windows to Linux. G++ is used to produce executable file on Linux.

This time, we have made remarkable progress in our chess program. Most of the bugs in our Chess ver1.0 have been fixed. It obeys all the chess rules and is able to make a so-called wise decision by thinking 5 or 6 or even more plies (using quiecience search). Our chess program incorporates opening book and different evaluation functions for different stages as opening, middle and ending. Transportation table is also implemented and its efficiency is discussed.

We referred to some web sites as listed in reference. But we borrowed only idea, not souce code.

Signature

_____ _____
_____

Description in Depth

- Quiescence Search

To overcome the effect of horizon, we implemented the quiescence search in our program. In each node, we store the evaluation value of

its parent node as well as its own value. If a node is a leaf, we tell if the difference between the evaluation values of its parent and its own is large enough. If the difference exceeds some threshold, quiescence search will be activated. The quiescence search is also a minimax search with αβ pruning. It differs from a normal search at two points: first it uses a much simpler evaluating function, mainly based on materials, to speed up; the other is it will not expand a node if this node is in quiescence state. The quiescence search has its own ply limit and children number upperbound, so we can adjust them seperately.

➢ Transposition Table

(We referred to [1] for the idea)

A transposition table is simply an array of information about recently visited positions in the game. Whenever you search a node in the game tree, you start by looking to see whether the corresponding position is already in the TT, and, if so, whether the information held there is useful.

We implement our TT using hash table, as most programs do. We use a 64*13 random long integer array to produce hash keys. Here 64 is the square number of a board and 13 is the number of different pieces (including blank). For a board status, the 64 random numbers are connected using XOR operator and thus produce the hash key. We also use another 64*13 random array to make checking number. Using this method, the probability of collision is nearly zero. In each entry of hash table, besides checking number, we also store the evaluation value, ply depth and a flag to indicate if this evaluation value is the exact value, or is only an upperbound or lowerbound (the reason is αβ pruning). These values make its combination with αβ pruning very easy.

➢ Opening Book

As thousands of strong players have been studying the initial position and have written about or played their findings. We don't want our program to waste the time rediscovering these moves, so there is usually a database of 'approved' openings.

We do consider incorporating a large database of both opening and ending which requires large data seeking and reading in specific file format. But due to limited time, we decided to encapsulate the most useful opening books into our program. And use the most simple search technique to find the opening book that can be applied.

➢ Evaluation Function

The evaluation function is the brain of our chess program, and it is also a key problem to solve. We try to take all the factors that

lead to win/loss into consideration by including material value, pawn position, king safety, board control, piece-attack, etc.

We have applied 3 different evaluators, respectively for opening, mid-game, and end-game. In all there stages, the material value is the most important factor, but for each stage, some criterion of evaluation may change. In the opening, we apply some basic rules by giving high priorities to moving out knights/bishops, castling, occupy or control the center of the board, and restrict moving the queen. In the mid-game, we stress the attacking value and king-safety, if some piece of the opponent is being attacked by us, there will be a bonus according to the opponent's and our piece's values. For king-safety, we restrict moving pawns beside the king by computing the pawn-structure, and encourage placing more pieces before our king and threatening the opponent's king. We clearly know if an exchange of pieces is worth. In the end-game, we stress the king's moving out to protect or attack pawns, instead of the king safety, and we give a high priority to promotion-hopeful pawn positions. The board control value of all stages, counted by each square, is computed but has a different weight to each stage, for example in the end-game it is not very important. Our program is also very reasonable in that if we guarantee our loss, we will resign.

The performance of our program is pretty good! Usually it can last about 40 moves when playing against GNU chess, and once it lasted 72 moves. Incredibly, it beat some human players when testing it. It is great that our program does have some intelligence!

- Bit Board

The efficiency of move generator is essential. We use the popular bit board method to facilitate our move generator. We referred to the basic description of bit board algorithm on [2] and some other web sites, but the concrete implementation is designed by ourselves, although it's probably not the optimal. We expend the use of bit board to compute the control matrix. Control matrix records the control status of each squrare and is well used in our evaluation function. We finished most of bit board algorithms in homework2B, and we added promotion, en passant and castling in this phase.

Discussion:

1. When we use bit board to compute the moves of bishop and queen, two rotation matrixs are needed to rotate bit board left 45 degree or right 45 degree [2]. After rotation, we need to rotate them back to original places. I first thought we could use Rotate45Left matrix to undo the rotation of Rotate45Right, and vice versa. But it was totally wrong. Finally we had to design two other matrixs call Reverse45Left and Reverse45Right.
2. When we implement the quiescience search, first we used a different move generator to produce only capture moves. This

method is proved not right at all. The reason is that when you use minimax tree in quiescience search, if you have only capture moves, you will often miss those good nodes and make your search meaningless.

3. After we implement the hash table, we did some experiments to test its efficiency. We found hash table is good at two situations. First is big ply limit number (at least 6, we think). The other is in the endgame phase. In our program, the possible ply limit is only 4 or 5, so no obvious improvement is observed. So we turn off the hash function when we turned in our program. A better solution may be changing ply limits dynamicly. Since in the endgame phase, the number of possibe moves is much smaller than middle game, we can both increate ply limit and use hash function. But we have no time to implement the idea. That's a little pity.

Reference:

1. http://www1.ics.uci.edu/~eppstein/180a/970424.html
2. http://www1.ics.uci.edu/~eppstein/180a/970408.html