Computer Chess: A Simple Implementation

by Rob Upcraft

Introduction

During the spring of 2011, I took an introductory course in artificial intelligence. The class itself was divided into three separate units: neural networks, automated theorem proving, and last of all, computer chess. This third section contained the smallest portion of time devoted to it and did not focus as much upon implementation as did the others. We had programmed our own neural networks and used an automated theorem proving program (Otter) to prove small statements, but the chess unit was largely theory.

A portion of our homework consisted of playing the game of chess with other students in order to develop a better sense of the game. I was (and remain) a mediocre chess player, however, and was repeatedly defeated by another student (and roommate) of mine. Thus, my motivation for creating my own chess AI stems from both an interest in implementing such programs and a personal desire to, at the very least, program a machine with the capacity to best this individual. Herein, I intend to briefly describe this implementation and the results thereof.

Program Description

Language and Environment

This chess program was written in C/C++ due to its portability and executable speed. It was initially written to run on a PC with a single CPU, but was later adapted with MPI to operate on Bethel University's supercomputing cluster in a parallel environment. As a result, the latest version [4] is only known to work on Linux, but earlier iterations have been compiled and run on both Windows and Macintosh platforms as well.

Program Design and Implementation Details

Board Representation

This implementation uses a small set of data structures and utility functions for board and move representation. The chess board itself is represented by 64 element array of type `char`. The setPieceAt() and getPieceAt() functions can be used to set and get, respectively, the values of particular grid squares. The contents of a grid square can be assigned a value such as EMPTY, PAWN, etc, as defined in the file "boardutil.h" to specify its value. Pieces on the "white" team are the defined piece value (PAWN, KNIGHT, etc) and those on the "black" team are defined as their piece value plus the value of BLACK (a black pawn, for instance, would have the value PAWN + BLACK).

Board Evaluation

Chess board structures are passed into the evaluateBoard() function to determine their value. This value is represented by a single integer; high positive numbers indicate a 'strong position for white while negative values with a large magnitude indicate a good position for black. A value of zero indicates a tie. In this implementation, different evaluation functions can be chosen by setting the "functionId" parameter in evaluateBoard().

Move Representation

Moves are represented using the "Move" structure as defined in "moves.h". This structure stores the type of piece (pawn, rook, etc), its starting row and column, ending row and column, and the score of the board after the move is executed. The board score is an integer that describes the state of the board in terms of its benefit to a player (see "Board Evaluation"). For castling, the move cannot be represented with a single set of start/ending rows and in this case, the constants CASTLE, KINGSIDE, and QUEENSIDE, are used in place of rows and columns to represent this action.

Move Selection Algorithm

This chess program uses a variation of the Minimax algorithm to select an optimal move. That is, it recursively searches a game tree (see Figure 1 for an example) out to the depth specified by the SEARCH_DEPTH constant in order to find the move that will most benefit itself while simultaneously taking into account an optimal opponent's likely moves. If two or more terminal nodes in the game tree have close values (as defined by the RAND_RANGE constant in "moves.h"), the program will randomly select one to return as the "best" move. In this program, this process is largely implemented in the bestMoveHelper() function in the "moves.c" file.

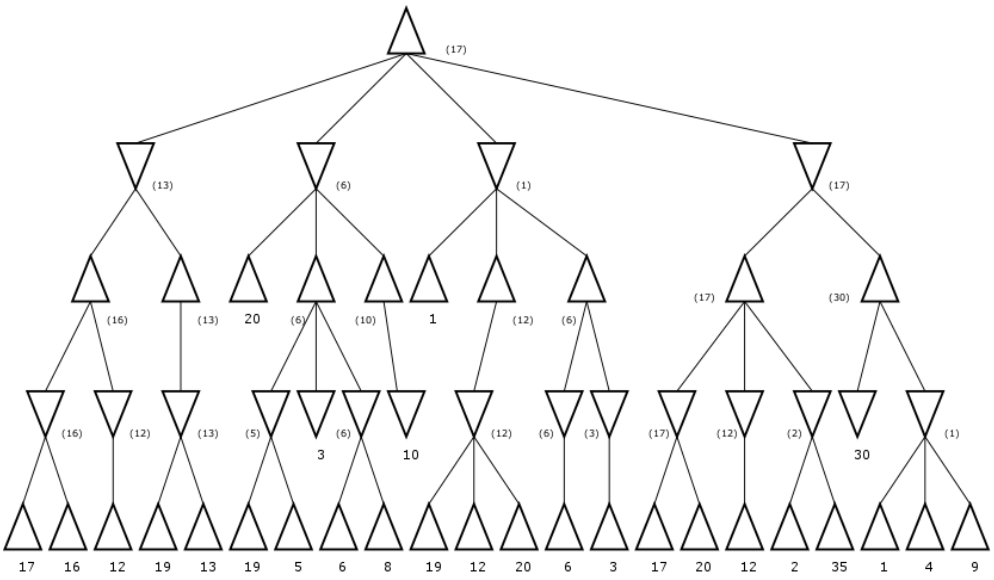

Figure 1: An example game tree [3]. The player at the root node attempts to maximize its own score while taking into account an optimal opponent minimizing its score at alternating levels of the tree. Here, the node with value 17 is chosen because it is the maximum at level 1, the minimum at level 2, etc.

Parallelization

Since the minimax search process can be modeled by a tree structure, the number of total nodes (board configurations) that must be analyzed per turn can be approximated by $n = \frac{m^{h+1}-1}{m-1}$ [1] where m is the number of legal moves available at each ply (about 30-40 for chess on average) and h is the depth of search. Thus, it can be seen that this algorithm has $O(m^h)$ running time and each additional level added to the search tree will add roughly 30^h extra nodes. For this reason, the single-processor version of the program was parallelized to make this increased depth of search viable. This was accomplished via an implementation of the following algorithm:

```
// Runs on the processor designated as the ‘‘head node’’.
function findBestMoveHeadNode()
    n = total number of available nodes excluding the head node
    legalMoves = array of legal moves available to the current player

    if size(legalMoves) > n
        partition legalMoves into n subsets of size, floor(size(legalMoves) / n)
            the n-th set will have an additional mod(size(legalMoves), n) moves
    else
        partition legalMoves into size(legalMoves) subsets of size 1
```

```
    for each subset created in the previous partitioning step
        send subset and the current board state to a non-head node for processing

    for each processor that received a move in the previous step
         wait for and receive optimal move and associated value

    find the maximum of the returned optimal moves, return it


// Runs on non-‘‘head node’’ processors
function findBestMoveSlaveNode()
    while program is running
        potentialMoves = receive set of moves and board state from head node
        bestMove = minimax(potentialMoves)
        send bestMove to head node
```

The implementation of this algorithm can be found in the bestMove() function within the source code (moves.c). It should be noted that in my implementation, the move partitioning and sending steps are accomplished simultaneously. MoveSet structures are created from the master legal moves list and sent to cluster nodes for processing.

Communication overhead that resulted from the cluster nodes sending and receiving data was minor as this took place only once during each player's turn and consumed very little time overall. Although this time was not directly measured, an approximation can be computed in the following fashion:

$$time = \frac{network\ overhead \times data\ (bits)}{network\ speed\ (bits/sec)}$$

We assume a 10% network overhead for the MathCS cluster. The total amount of data sent and received (in bytes) for each turn can be approximated by $85 + 17n + 8p$ (see source code) where n is the number of legal moves that need to be distributed and p is the number of cluster nodes being used. Since we assume an average of 30 moves per ply (and 30 available cluster nodes), the average communication overhead is estimated to be $\frac{1.1 \times 8 \times (85+17(30)+8(30))}{100{,}000{,}000} \approx 7.348 \times 10^{-5}$ seconds per turn.

Performance and Results

Gameplay Observations

Over the course of the project, the program's gameplay was continuously improved. The most noticeable advancement took place when the game tree search depth was increased by an extra level following parallelization. The human players who had competed with the

program throughout its development noted an increase in difficulty after this capability was added and it was at this point that the computer began to win games against them. To verify this increase in performance, I configured the program to compete against itself with one player looking ahead one ply further than the other. In both scenarios (white/black search advantage), the player with the ability to search further in the game tree won.

Another facet of the program's gameplay that should be noted was its tendency to play well defensively and capitalize on its opponents mistakes. Its biggest gains usually occurred during the middle game when a human opponent failed to safely execute an offensive strategy. However, because the program does not have a built-in opening or closing database, the opening and especially the end game tended to progress very slowly. Thus, while the software defended itself well and removed its opponent's material value in the process, it had a difficult time achieving a checkmate.

Evaluation Function Comparison

Although not central to the overall scope of the project, I spent some time developing and comparing different board evaluation functions. Alan Turing, for example, proposed evaluating the state of the board purely by material value [2]. Additional features such as mobility, pawn structure, and piece safety would be examined in the event of a tie, but the point values associated with them were much smaller than those attached to material value.

Alan Turing's Weights:
Pawn = 100
Knight = 300
Bishop = 350
Rook = 500
Queen = 1000

As evidenced by Turing's function, material value is generally of highest importance in computer chess and programs that search further in the game tree will usually have an advantage over those that searches to a shallower depth. However, changes in these weights and extra considerations like mobility can have an effect on the overall behavior of a program.

For this project, I included material weight functions by Alan Turing and Hans Berliner (a noted computer chess expert) in addition to one of my own design. My function had material weights almost identical to Turing's, but included considerations for piece safety and pawn advancement. The piece safety measure was included to alert the AI player of a potential threat at its evaluation horizon (the maximum depth to which the computer is searching the game tree). The small bonus for advancing pawns was also added to reward the computer for moving them closer to a position at which promotion is possible. Since a promotion event is beyond the horizon for my implementation from the initial board configuration, this was included to encourage the AI player to move its pawns to a position from

which promotion could be predicted.

Rob's Evaluation Function:
Pawn = 100
Knight = 300
Bishop = 350
Rook = 500
Queen = 900

Piece Safety: Each piece on the board that cannot be captured by an opponent on the next ply increases the board value by an extra $\frac{1}{30}$ of its value.

Pawn Advancement: Pawns increase the total board value by 4 points for every additional rank they are advanced toward the opponent's side.

I conducted a series of tests in which one AI player using one evaluation function would compete against one with a different function. Although my evaluation function provided a small reward for pawn advancement, it did not make a noticeable difference during the early game as both the Turing and Berliner valuations resulted in comparable pawn advancement (see figure 2). One notable difference that did occur, however, was that players using these material-only functions (Turing and Berliner) tended to develop their Knights earlier in the game due to their lack of a pawn advancement bonus.

```
8|#R|#N|#B|  |#K|#B|#N|#R|        8|#R|  |#B|#Q|#K|#B|#N|#R|
7|#P|#P|#P|#P|  |#P|#P|#P|        7|#P|#P|#P|#P|  |#P|#P|#P|
6|  |  |  |  |#P|  |  |  |        6|  |  |#N|  |  |  |  |  |
5|  |  |  |  |  |  |  |  |        5|  |  |  |  |#P|  |  |  |
4|  |  |  |  |  |  |  |#Q|        4|  |  |  |  |@P|  |  |  |
3|@N|@P|  |  |  |  |  |  |        3|  |  |  |  |  |  |@Q|  |
2|@P|@B|@P|@P|@P|@P|@P|@P|        2|@P|@P|@P|@P|  |@P|@P|@P|
1|@R|  |  |@Q|@K|@B|@N|@R|        1|@R|@N|@B|  |@K|@B|@N|@R|
-|--|--|--|--|--|--|--|--|        -|--|--|--|--|--|--|--|--|
 | A| B| C| D| E| F| G| H|         | A| B| C| D| E| F| G| H|
```

Figure 2: Sample boards, 5 plies into each game. White and black pieces are denoted by the symbols '@' and '#', respectively. Left: Turing (white) v Rob (black). Right: Rob (white) v Berliner (black). Notice that Turing and Berliner functions result in quick Knight development. Rob's function tends to develop the Queen early.

Conclusion

This project began as fun experiment to see if I could develop a chess program that would make legal, albeit uninformed moves. After this requirement was met, the software evolved into my senior project and it became my goal to make the program reasonably competitive. That is, it needed to have the functionality to best, or at least put up a strong fight against me in a game of chess. Given that it surpassed my chess playing ability some time ago and in its post-parallelized state, was able to defeat human players who regularly beat me, I would argue that this goal has been met.

In addition the development of the core program, I experimented with some simple evaluation functions and the initial results seemed to confirm what had been covered in the introductory Artificial Intelligence course; while considerations like safety and mobility can affect machine choices, material value and search depth tend to hold the most weight in such decisions. Were I to re-write the program from scratch, some design changes would be made for efficiency and code readability, but I consider the current version to be a success for its scope. Perhaps the largest future extension I would propose is the addition of opening and end game databases in order to make it perform more competitively in these areas.

Acknowledgements

This project would not have been possible without help from a number of other individuals. Among them are the following:

Dr. Brian Turnquist - Project advisor.
Dr. Nathan Gossett - Project reader, supported the supercomputing cluster.
Dr. Andrew Fast - Project reader.
Brian Olson - Tested the program throughout its development.
Gabe Kleinschmidt - Tested the program throughout its development.

References

[1] Eric Gossett. Discrete Mathematics with Proof. Hoboken, NJ: John Wiley & Sons, Inc., 2009. Print.

[2] Monroe Newborn. Computer Chess. New York: Academic Press, Inc., 1975. Print.

[3] Ray Toal. Minimax Game Tree. N.d. Graphic. Loyola Marymount University. 8 Feb 2012. `http://www.cs.lmu.edu/~ray/images/minimax.png`.

[4] Source code available at: `http://www.robupcraft.com`