CS 297 Report

Improving Chess Program Encoding Schemes

Supriya Basani

(sbasani@yahoo.com)

Advisor: Dr. Chris Pollett

Department of Computer Science

San Jose State University

December 2006

Table of Contents:

Introduction

Chess-playing programs have been one of the most studied areas for Artificial Intelligence research. Many successful chess programs can beat chess experts, yet their style of play is incomparable to chess grandmasters. Alpha-Beta pruning algorithm written in 1963 is one of the most popular search algorithms on game trees. Many enhancements on top of this algorithm have been implemented to improve the search efficiency. Apparently this simplistic depth first, brute-force approach does not compile well with Artificial Intelligence techniques. Unlike the computer logic that examines every possible position for a fixed number of moves, the grandmasters get their right moves from constructing the whole board based upon few pieces of information on the board and from recollections of salient aspects of past games.

This report summarizes modifications to an existing computer chess program, GNU chess, so that it plays more like a human player. GNU Chess is a free chess-playing program developed as part of the GNU project of the Free Software Foundation (FSF). GNU Chess is intended to run under Unix or Unix-compatible systems. It is written in C and should be portable to other systems. My goal was to fully understand how GNU chess program worked and then experiment with my notion of human like encoding schemes. Instead of depending on the complicated search algorithm to find the next best move, my logic was to use the chess game database as much as possible. I was able to modify the Principal Variation Search algorithm used in GNU chess to lookup next best move from the chess game database more efficiently.

During my research and development work performed in CS 297 I presented four deliverables. This report summarizes all the work done in each of this deliverables. In the first deliverable I presented my findings on how to use external chess game database with GNU chess. In the second deliverable I presented detailed description of how Principal Variation Search (PVS) algorithm works and compared it with regular Alpha-Beta pruning. In the third deliverable I presented my modification to the PVS algorithm where it looks up the chess game database for the next best move during each search depth. This modification helped reduce the number of depths the PVS search algorithm had to search for the next best move. Along with this deliverable I also presented the auto play setup for chess programs where two GNU chess programs could play against each other. This setup is very useful to compare my chess program with the existing chess programs. In the fourth deliverable I was able to present the modification to chess game database lookup algorithm such that lookups can be significantly faster and more efficient. Finally I conclude this report with a detailed description of future work that will be done in CS 298.

Deliverable 1:

Chess Game Databases and GNU Chess Program

My project had started out with my research on how GNU chess worked. I researched and understood how external chess game databases can be used with GNU chess program. Chess game databases come in .PGN format. PGN format file is converted into binary (book.dat) format by running the GNU Chess command. The book.dat is a binary file written in network byte order. Once the database is converted into binary format, GNU Chess consults the book for next moves. If it finds an appropriate matching move,

it uses that move otherwise the program calculates the next best move using the PVS GNU Chess algorithm.

Figure 1: Integrating grandmaster Anand Vishwanathan's 2106 games:

```
>./gnuchess.exe
GNU Chess 5.07
Adjusting HashSize to 1024 slots
Transposition table:  Entries=1K Size=48K
Pawn hash table: Entries=0K Size=32K
White(1) : book add Anand.pgn
Created new book book.dat!
Got 107 hash collisions

Time = 4 seconds
Games compiled: 2106
Games per second: 526.500000
Positions scanned: 32504
Positions per second: 8126.000000
New & unique added: 12229 positions
Duplicates not added: 20275 positions
```

In Figure 1 I compiled grandmaster Anand Vishwanathan's 2106 games and converted it into book.dat format used by GNU Chess program.

Book.dat generation algorithm:

1. The book program uses lex and yacc to parse the Anand.pgn file.
2. Check if trusted player and decide to add to bookpos[]. bookpos[] array is almost 1MB size (1MB book moves can fit in it).
3. Call MakeMove (bookmove).
4. Calculate HashKey based on board position (collisions for two different board positions should be very rare with 64bit HashKey and a good hash algorithm).
5. Add the move to bookpos[] if its unique.

   Currently bookpos stores the following for each move.

```
bookpos[i].key = HashKey;
bookpos[i].wins;
bookpos[i].losses;
bookpos[i].draws;
```

6. Reset the board position to initial state. And repeat steps for next entry in the PGN file.

Figure 2: Example of next move lookup from Chess game database:

```
> My first move ➔ White (1) : d4
              BOARD:
              r n b q k b n r
              p p p p p p p p
              . . . . . . . .
              . . . . . . . .
              . . . P . . . .
              . . . . . . . .
              P P P . P P P P
              R N B Q K B N R

Computer ➔ Thinking...
Looking for opening book in book.dat...
Read opening book (book.dat)...
Loading book from book.dat.
3054 hash collisions...  Opening database: 12229 book positions.
 In this position, there are 5 book moves:
 Nf6(58/81/39/149)  d6(58/5/3/5)  e6(25/0/1/1)  d5(55/38/21/103)
 f5(50/1/1/1)

 Nf6(97)  d5(73)  d6(55)  f5(33)  e6(0)
 Computer's move ➔ Nf6
```

Figure 2 shows an example of how GNU chess calculates the next best move from the chess game database.

Database lookup algorithm:

The GNU Chess program has to do the following anytime it checks if there is a book move.

1. Based upon current board position generate all legal next moves.
2. For each legal move, calculate the HashKey for the current board.
3. Do a sequential digest search for the HashKey in bookpos[] array. If the HashKey is found, then there is a book move, if not continue looking for all other legal moves.
4. If none of the legal moves HashKey matches then the program runs the PVS search algorithm to generate the next best move.

5. If more than one match is found then the program picks the move based upon number of games won with that move. It is also configurable how it picks the move - best, worst, random and so on.

Deliverable 2:

GNU Chess program's PVS Algorithm

In this deliverable I presented by findings on how PVS algorithm works in GNU Chess program. In order to understand the algorithm I had implemented the Alpha-Beta cutoffs algorithm and Principal Variation Search algorithm and ran it on sample input values. I also added special print statements into GNU Chess PVS algorithm in order to demonstrate how PVS algorithm works. The results of the finding are explained via the program outputs.

Figure 3: Sample game tree showing Alpha-Beta cutoffs and PVS cutoffs.

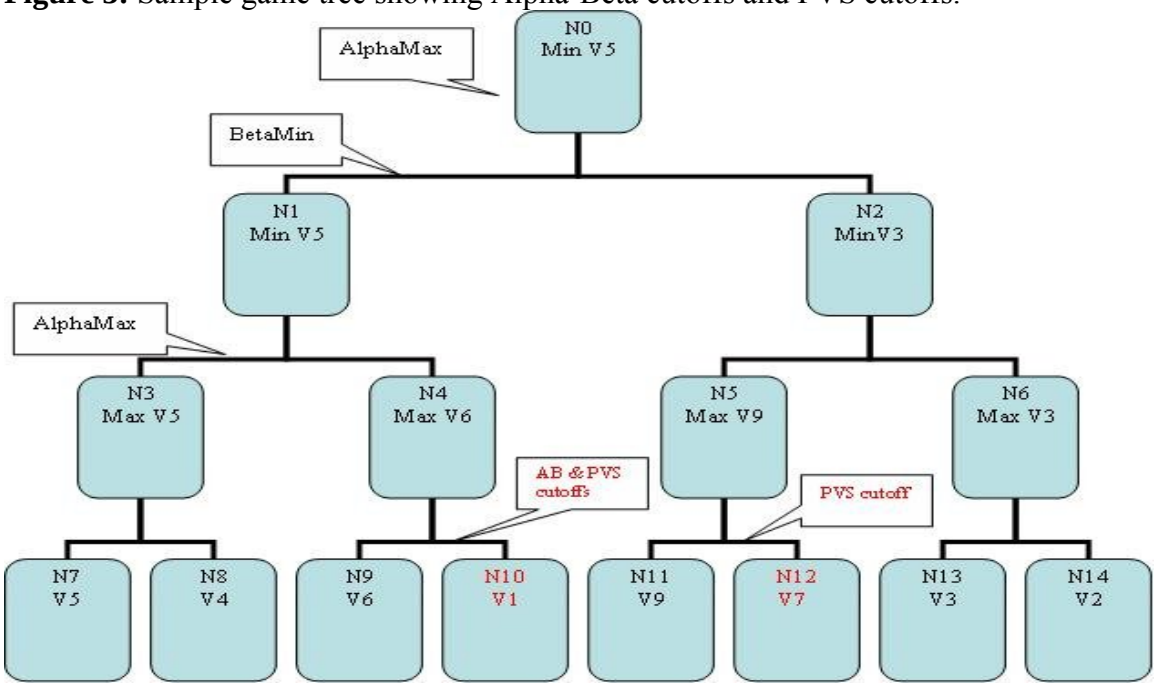


Figure 3 represents a sample game tree which was used to compare Minimax, Alpha-Beta cutoffs and PVS cutoffs. Game tree is a recursively defined data structure that consists of

the root node representing the current state and finite set of branches representing the legal moves. These branches point to the potential next states, each of which, in turn, is a smaller game tree. The distance of a node from the root is its depth.

PVS algorithm:

PVS algorithm maximizes the portion of game tree that can be cutoff by attempting to rapidly determine the best value of α and β. These α and β values define a window within which the Minimax value must lie; thus smaller the window the greater the cutoffs.

Once the algorithm finds a move (PV move/node) with score that is between the α and β value, the rest of the moves are searched with the goal of proving that they are all bad. If the algorithm finds out that it was wrong, and that one of the subsequent moves was better than the first PV move, it has to search again, in the normal Alpha-Beta manner.

Comparing Minimax, Alpha-Beta and PVS output on game tree represented in Figure 3:

Regular Minimax program visited all nodes in the game tree. Total number of nodes visited = 15.

With Alpha-Beta program, the number of nodes visited was 14. Cutoff occurred at node N4 because the alpha value (6) was greater than the beta value (5). Node N10 was cut off.

For the same game tree, the number of nodes visited using the PVS algorithm was 13. Cutoff occurred at N4 (N10 was cutoff) and also at N5 (N12 was cutoff).
N10 was cutoff due to basic alpha beta pruning.

N12 was cutoff due to the PVS algorithm. This cutoff occurred since at N5 alpha 9 >= beta 6. PVNode with a value of 9 was assumed at N5. Since N13 and N14 were also less than 9, this assumption proved to be right.

Figure 4: PVS output

```
./pvs.exe
minimax at node N7, alpha = -2147483648, beta = 2147483647 score = 5
Found PVNode. val = 5, alpha = -2147483648, beta = 2147483647
Calling2 minimaxAB with Minimal Window 5 6
minimax at node N8, alpha = 5, beta = 2147483647 score = 4
minimax at node N3, alpha = -2147483648, beta = 2147483647 score = 5
minimax at node N9, alpha = -2147483648, beta = 5 score = 6
Found PVNode. val = 6, alpha = -2147483648, beta = 5
CUTOFF For N4 because alpha 6 >= beta 5
Nodes cutoff: N10
minimax at node N4, alpha = -2147483648, beta = 5 score = 6
minimax at node N1, alpha = -2147483648, beta = 2147483647 score = 5
Found PVNode. val = 5, alpha = -2147483648, beta = 2147483647
Calling2 minimaxAB with Minimal Window 5 6
minimax at node N11, alpha = 5, beta = 6 score = 9
Found PVNode. val = 9, alpha = -2147483648, beta = 6
CUTOFF For N5 because alpha 9 >= beta 6
Nodes cutoff: N12
minimax at node N5, alpha = 5, beta = 6 score = 9
minimax at node N13, alpha = 5, beta = 6 score = 3
Found PVNode. val = 3, alpha = -2147483648, beta = 6
Calling2 minimaxAB with Minimal Window 5 6
minimax at node N14, alpha = 5, beta = 6 score = 2
minimax at node N6, alpha = 5, beta = 6 score = 3
minimax at node N2, alpha = 5, beta = 2147483647 score = 3
PVS Alpha Beta Minimax at root position N0 is: 5
```

However, if the following values were put into the tree:

N12 = 10 N13 = 11 and N14 = 11

In this case, N12 = 10, would be the final PVS value at N0. N14 would be cutoff since at N6 alpha 11 >= beta 10.

So, when the PVS algorithm finds that minimax at node N6 = 11, it knows that the previous assumption of cutting off N12 = 10 was incorrect. So, it has to now search N12 with alpha = 9, beta = 2147483647 score = 10.

The above example illustrates two important things:

1. If move ordering is good, PVS usually does better than Alpha-Beta. More nodes can be cutoff. (In the example tree, one extra node was cutoff).
2. If the move ordering is not so good, PVS might have to re-search some of the subtrees incurring performance penalty. In any case, PVS will not search any more nodes than alpha beta, but it might have to re-search some of the subtrees like the example with the changed values mentioned above.

Deliverable 3:

Extension to PVS algorithm and Auto play setup

1. Extension to PVS algorithm

In GNU chess when the current board position is not found in the game database (Book) then the next move is calculated using the PVS algorithm. As PVS algorithm tries to refine α (or β) by searching several ply along the game tree, it is possible to reach a board position with some future moves which can be found in the game database. That is, two games can reach the same board positions even though they do not have the same sequence of moves.

Assume two games:

1. a b 2. c d 3. e f

2. c b 2. a d 3. e f

At move 3, the board position is same even though the initial sequence of steps is different. Thus their HashKey derived from current board position should be same.

My extension to PVS algorithm was to compare current board's HashKey at each search depth during PVS calculation with board HashKeys in the game database. If a matching board position was found then the move leading to that board position was returned and the PV search was terminated.

I implemented a special option called '-b' in GNU chess program which ran the program with the above extension. In order to test this extension the chess program was run against a dummy game database that had only one simple game as follows:

dumb.pgn

---

1. e4 Nc6 2. Ke2 d5 3. Ke1 Nf6 4. exd5 Qxd5 5. Ke2 Qe4#

{computer wins as black} 0-1

During the actual play if the user makes the following moves:

1. e4 Nc6 2. Ke2 d5 3. exd5

Now computer's next move with the new extension will be:

Next move lookup from game database fails because current board position does not exists in the game database.

PVS algorithm is run:

```
Root = 54, Phase = 1
Time = 5.00, Max = 20.00, Left = 0.00, Moves = 0
Ply  Time   Eval     Nodes Principal-Variation
 1+  0.00   1238     Qxd1      Qxd5
 1.  0.00  29238     NB32      Qxd5
 2-  0.00  29/20     N504
 2&  0.00   1134     Q548      Qxd5 Ke1
 2.  0.00  29134     N577      Qxd5 Ke1
 3&  0.00   1139     Q948      Qxd5 Ke1 Qe5+ Be2
 3.  0.02  29139     7929      Qxd5 Ke1 Qe5+
 4&  0.03   1115     9980      Qxd5 Ke1 Nf6 Nc3
 4.  0.03    115     9980      Qxd5 Ke1 Nf6 Nc3

Time = 0.0 Rate=388917 Nodes=[9708/272/9980] GenCnt=24847
Eval=[899/3189] RptCnt=0 NullCut=3 FutlCut=7588
Ext: Chk=1115 Recap=11 Pawn=364 OneRep=36 Horz=2 Mate=0 KThrt=68
Material=[3600/3600 : 4300/4300] Lazy=[150/249] MaxPosnScore=[150/385]
Hash: Success=15% Collision=64% Pawn=72%
```

Explanation:

On Ply 4& above, PVS generates "Qxd5 Ke1 Nf6 Nc3". After Nf6 move, the board position is same as

Qxd5 Ke1 Nf6 (HashKey Match found) Nc3

Game Database has the following sequence:

1. e4 Nc6 2. Ke2 d5 3. Ke1 Nf6 4. exd5 Qxd5

Current board could follow the following sequence after computer makes move Qxd5:

1. e4 Nc6 2. Ke2 d5 3. exd5 Qxd5 4. Ke1 Nf6

Thus PVS stopped at Ply 4& and used the PVNode at that point instead of going ahead. Without the extension the PVS code would have searched until Ply7 and in this case would have still returned move Qxd5.

Figure 5: Snippet of PVS extension code

```
/*
 * At every depth, if PVS extension is specified, consult the book and see if there is a
 * match board position. If a match found, use the RootPV as the next black move.  If not,
 * continue with the PVS as usual until the maximum depths possible.
 */
int SearchRoot (short depth, int alpha, int beta)
{
                    ...
                    ...
      if(flags & BOOKPVS) {
        found = BookPVSQuery(HashKey, SANmv);
        if (found == true) {
          // Book move found with new query.  Stop PVS now and use the PVNode immediately.
          SANMove (RootPV, 1);
          fprintf(gdbg_fp, "\nPVS extension: Stop PVS and use book move. depth = %d\n.", Idepth);
          SET (flags, TIMEOUT);
        }
    }
    ...
    ...
}
/*
 * Called from each depth of PVS search to see if there is a matching book entry
 * based upon the new board position at this pvs move
 */
int BookPVSQuery(HashType hkey, char *move)
{
          int j;
          if(bookloaded && !book_allocated) {
                    return BOOK_ENOBOOK;
          }
          for (DIGEST_START(j,hkey); !DIGEST_EMPTY(j); DIGEST_NEXT(j, hkey)) {
            if (DIGEST_MATCH(j, hkey)) {
                    if (bookpos[j].wins > 0) {
                              fprintf(gdbg_fp, "PVS extension: Match found HashKey = 0x%x ", hkey);
                              fprintf(gdbg_fp, " move(%d) = %s wins = %d ",
                                        GameCnt/2 + 1, move, bookpos[j].wins);
                              fflush(gdbg_fp);
                              return true;
                    }
            }
          }
          return false;
}
```

2. Autoplay Setup

With Autoplay setup I was able to make chess engines play against each other. This feature is very useful since enhancements made to the GNU chess engine and the original engine can play against each other and their performance can be compared.

Autoplay is an open source chess program that connects two xboard/winboard protocol compliant chess engines and lets them play against each other. GNU chess is an xboard compliant chess engine and it could be run in the engine mode using the –x option.

The engine displays the information in the Coordinate Notation that uses only the squares that the pieces were on to denote movements. (such as 1.e2e4 e7e6)

I was able to modify the Autoplay source code such that it stored all the moves made in a .PGN format which I could later run it on xboard to view the complete game.

Autoplay can be started like this:

./autoplay.exe -1 "./White_gnuchess.exe -x" -2 "./Black_gnuchess.exe -x"

How Autoplay works:

Autoplay creates two new processes for each GNU chess engine.

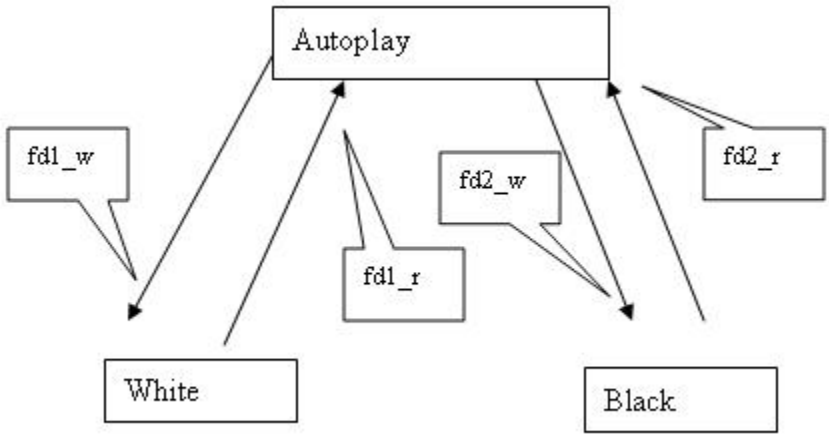
It does a fork/exec for White_gnuchess.exe and then fork/exec for Black_gnuchess.exe.

There are two pipes that are created per process:

White_gnuchess.exe: fd1_r and fd1_w

Black_gnuchess.exe: fd2_r and fd2_w

Figure 6: Autoplay Design



Autoplay - Reads from fd1_r (reads a move from white chess engine)

Writes to fd2_w (writes the move to black chess engine)

Reads from fd2_r (reads a move from black chess engine)

Writes to fd1_w (writes the move to white chess engine)

Autoplay process does a select(fd1_r, fd2_r) and reads the data on whichever filedescriptor the data has arrived on. Then, it writes the data to the other engine.

Deliverable 4:

Extension to Chess game database lookup logic

GNU chess program's game database (book) lookup logic was enhanced such that the lookups were made much faster and efficiently.

Anytime GNU chess program tries to lookup moves from the game database (book.dat) if has to do the following:

1. Based upon current played board position generate all legal next moves.
2. For each move, calculate the hashkey for the current board.

3. Do a sequential digest search for the hashkey in bookpos[] array. If the hashkey is found, then there is a book move, if not then continue looking.
4. If more than one book move is found then select the one with highest wins.

Currently bookpos stores the following:

```
bookpos[i].key = HashKey;
bookpos[i].wins;
bookpos[i].losses;
bookpos[i].draws;
Eg: (0x88bd7241, 2, 0, 1)
```

As an extension to this book lookup logic I modified the information that is stored in bookpos such that for each white move read from the game database I store all the winning next black moves with the total number of wins. GNU chess program when run with –B option runs the program with the following extension.

Assuming user is playing white and computer is playing black.

We can just store the following in new format:

```
static struct hashtype {
        uint16_t wins;
        uint16_t losses;
        uint16_t draws;
        HashType key;
        struct nmove {
                char    nmove[SANSZ];
                uint16_t nwins;
        } nextmoves[MAX_BLACK_NEXTMOVES];
} *bookpos;

Example: (0x88bd7241, 2, 0, 1, e5 4, Nc6, 8, a6, 2, Nf6, 9)
          (with MAX_BLACK_NEXTMOVES = 4)
```

With this extension when the game starts, the book lookups will be faster because now the program has to only generate current board's HashKey, find the matching HashKey from the book and then look at the next black winning moves and pick the one with highest number of moves. Information stored in book.dat will also be similar format.

Thus we can skip generating the legal moves and calculating Hash Key and doing a lookup each time.

Example 1: GNU Chess with original logic:

```
> ./gnuchess -e -p
GNU Chess 5.07
Adjusting Hash Size to 1024 slots
Transposition table:  Entries=1K Size=40K
Pawn hash table: Entries=0K Size=28K

White move (1): e4
 Computer Thinking...
BookQuery: Legal moves generated = 20, Number hash slots searched = 13
 Opening database: 996148 book positions. In this position, there are 10 book moves:
e5(600)  c5(479)  e6(206)  c6(132)  b6(101)  g6(96) d6(89) Nf6(72) Nc6(64)  d5(62)
 Computer move is: e5
...
White move (4): Ba4
Computer Thinking...
BookQuery: Legal moves generated = 32, Number hash slots searched = 18
Opening database: 996148 book positions. In this position, there are 8 book moves:
Nf6(241)  d6(78)  b5(34)  Nge7(33)  f5(33)  Bd6(0)  Be7(0)  g6(0)
Computer move is: Nf6
And so on ....
```

Example 2: GNU Chess with Book Extension:

```
> ./gnuchess -e -p -B
GNU Chess 5.07
Adjusting Hash Size to 1024 slots
Transposition table:  Entries=1K Size=40K
Pawn hash table: Entries=0K Size=28K

White move (1) : e4
Computer Thinking...
BookQuery: Following next black moves found:
Move = e6, Wins = 41
Move = e5, Wins = 162
Move = c5, Wins = 100
Move = c6, Wins = 23
NEXT BLACK MOVE CHOSEN. Move(1) = e5 RootPV= 3364
Computer move is : e5
...
White move (4) : Ba4
Computer Thinking...
BookQuery: Following next black moves found:
Move = d6, Wins = 4
Move = f5, Wins = 1
Move = Nf6, Wins = 49
NEXT BLACK MOVE CHOSEN. Move(4) = Nf6 RootPV = 4013
Computer move is: Nf6
And so on...
```

Without the extension the GNU chess program had to generate following number of moves, calculate HashKey for each of those moves and had to search the book for the number of slots mentioned below for just first five moves.

BookQuery: Legal moves generated = 20, Number hash slots searched = 13

BookQuery: Legal moves generated = 29, Number hash slots searched = 13
BookQuery: Legal moves generated = 30, Number hash slots searched = 16
BookQuery: Legal moves generated = 32, Number hash slots searched = 18
BookQuery: Legal moves generated = 30, Number hash slots searched = 16

All these were skipped in the extension where we stored the 4 possible next black winning moves. As the game progress, the number of legal moves will increase and so the book search will take more time without the above extension.

Book extension code:

```
/* Each entry in book.dat is also extended to contain
 * upto next 4 winning black moves.
 */
static unsigned char buf[2+2+2+8+((SANSZ+2)*MAX_BLACK_NEXTMOVES)];

/* Offsets */
typedef struct {
  int nmove_off;
  int nwins_off;
} noff_t;
static noff_t nextoffs[MAX_BLACK_NEXTMOVES] = {
  14, 22,
  24, 32,
  34, 42,
  44, 52
};
/* Generating the binary game database file.
 * Fill up buf to write to disk (book.dat)
 */
book_to_buf()
{
      ...
  for (n = 0; n < MAX_BLACK_NEXTMOVES; n++) {
      memcpy(&buf[nextoffs[n].nmove_off],
bookpos[index].nextmoves[n].nmove, SANSZ);
      for (k=0; k<2; k++) {
            buf[nextoffs[n].nwins_off + k] =
((bookpos[index].nextmoves[n].nwins)   >> ((1-k)*8)) & 0xff;
      }
  }
}
/* Read from disk (book.dat) and populate bookpos
 */
buf_to_book()
{
      ...
  for (n = 0; n < MAX_BLACK_NEXTMOVES; n++) {
      memcpy(bookpos[i].nextmoves[n].nmove,
&buf[nextoffs[n].nmove_off], SANSZ);
```

```
      bookpos[i].nextmoves[n].nwins += (buf[nextoffs[n].nwins_off] <<
8) | buf[nextoffs[n].nwins_off+1];
  }
}

/* Add the next black move in this white bookpos index
 */
void BookAddNextBlackMove(unsigned long index, char *move)
{
   ...
   // Check if the move is already there.
   for (n = 0; n < MAX_BLACK_NEXTMOVES; n++) {
      if (strcmp(bookpos[index].nextmoves[n].nmove, move) == 0) {
            wins = ++(bookpos[index].nextmoves[n].nwins);
            found = true;
            return;
      }
    }
    // No match.  Put in the first available slot.
     for (n = 0; n < MAX_BLACK_NEXTMOVES; n++) {
      if (strcmp(bookpos[index].nextmoves[n].nmove, "") == 0 ||
            bookpos[index].nextmoves[n].nmove[0] == '\0') {
            strcpy(bookpos[index].nextmoves[n].nmove, move);
            wins = ++(bookpos[index].nextmoves[n].nwins);
            return;
      }
     }
     return;
}

/*
 * For a given white hashkey, find the next black move with the
 * highest number of wins.
 */
int BookFindNextBlackMove(HashType hkey)
{
   ...
   for (DIGEST_START(j,hkey); !DIGEST_EMPTY(j); DIGEST_NEXT(j, hkey)) {
      if (DIGEST_MATCH(j, hkey)) {
          // If multiple winning moves are there, pick the
          // one with highest wins
          printf("\n\nFollowing next black moves found:\n\n");
          for (n = 0; n < MAX_BLACK_NEXTMOVES; n++) {
            if (bookpos[j].nextmoves[n].nwins > maxwins) {
                  maxwins = bookpos[j].nextmoves[n].nwins;
                  maxindex = n;
            }
            if (bookpos[j].nextmoves[n].nwins > 0) {
                  printf("Move = %s, Wins = %d\n",
                  bookpos[j].nextmoves[n].nmove,
                  bookpos[j].nextmoves[n].nwins);
            }
          }
          if (maxindex < 0 || maxindex >= MAX_BLACK_NEXTMOVES) {
            return false;
          }
          move = bookpos[j].nextmoves[maxindex].nmove;
```

```
            p = ValidateMove(move);
            RootPV = p->move;  //set the NEXT Black move chosen
            return true;
        }
     }
     return false;
}

{ //Code modification done in lexpgn.c
   ...
   /* MakeMove increments GameCnt */
   MakeMove(side, &p->move);
   if (addtobook[side]) {
      if (BookBuilder (result, side) == BOOK_EFULL) {
            printf("Book full - Failed to add move %s\n",
                   yytext);
            ShowBoard();
            return 1;
      }
      /*
       * If book extension is specifed, save the next winning
       * black move in same index as the current white
       * move bookpos index.
       */
      if (flags & BOOKEXT) {
            if (side == black && white_book_index != 0) {
                  // Put this black move into that index.
                  BookAddNextBlackMove(white_book_index, yytext);
                  white_book_index = 0;
            }
      }
     }
     ...
}
```

Future Work

My research will continue on comparing and converting Chess game board so that next best move can be calculated based upon games that have been seen before.

Board conversions:

My Goal will be to keep a collection of several board games. Possibly create cluster of game boards based upon how close they are to each other. At any point when the next move is being calculated, check if a matching entry can be found in the stored games.

If a matching entry cannot be found then generate some number of moves on the current board position such that it might lead to a stored board position. In some cases, just one move might lead to a few stored games. Pick the best move that had the maximum number of wins. In some case, multiple moves might lead to a stored game even though the current board position might not be present in the stored games list.

"Board Conversion" is required in this case. Following points will be researched and implemented in CS 298:

1. How many moves and what moves are required by both players to convert the current game into a stored game? (in general, given a board, what are moves required to convert it to another board).
2. If 2 moves might lead to a stored board game and 4 moves might lead to another stored board game, which would you choose? 2 moves has higher probability of getting to the stored game.. but 4 moves might lead to more games which have more wins?
3. W hat are all the cases when a board cannot be converted to another board? In all those cases, this conversion part can be skipped.
4. How much depth to search (how many plys) when converting so that we don't spend too much time in an inefficient manner?
5. Can you always decide that two board games are same? (HashKey)
6. If you have gone too far into the game without finding any matching entries from the stored game, how realistic is that you can find some matching entries later? When should you stop doing such a conversion so that time is not wasted?

Conclusion:

The GNU chess is the first open source software that I had to work so closely with. Since open source software does not come with proper documentation, at times it had been very challenging to understand the code. I also spent a lot of time learning new chess moves and about different grandmasters. The best part though was that I got to play chess a lot during this semester and apparently my family also has got hooked on chess.

Reference:

[1973] Mechanisms for comparing chess programs. T. A. Marsland. P. G. Rushton. ACM Press. 1973.

[1982] Parallel Search of Strongly Ordered Game Trees. T. A. Marsland. M. Campbell. ACM Press. 1982.

[1983] Computers, Chess, and Cognition. T. A. MArsland. J. Schaeffer. Springer-Verlag. 1983.

[1989] Control Strategies for Two-Player Games. B. Abramson. ACM Press. 1989.

[1996] New Advances in Alpha-Beta Searching. J. Schaeffer. A. Plaat. 1996.

[1996] Recent advances in computer chess. M. Newborn. T. Marsland. ACM Press. 1996.

[2004] Chess playing machines from natural towards artificial intelligence?. F. Paul. Technical University of Wroclaw. 2004.

[2006] The Expert Mind. P. Ross. Scientific American. 2006.

[2006] Learning long-term chess strategies from databases. Sadikov. Aleksander. Bratko. Ivan. Kluwer Academic Publishers. 2006.

http://www.chessgames.com/ Online historical chess games databases.

http://www.gnu.org/software/chess/ GNU chess program web site.