Table of Contents

Home * Engines * CPW-Engine * **Root**

Header

```
/* this structure is meant to include data determining performance
   of the search routine, such as timing method, available time
   or depth, as well as boolean flags for additional heuristics
*/

struct sSearchDriver {
       int DEPTH;
};
extern sSearchDriver d;

/* this structure contains some statistical data about current search
*/

struct sSearchStat {
       int nodes;
       int qnodes;
}
extern sSearchStat stat;




#include "stdafx.h"



#define MAX_DEPTH 100
#define LEAVES_IN_TT 1
```

```
/* symbols used to enhance readability */
#define DO_NULL    1
#define NO_NULL    0
#define IS_PV      1
#define NO_PV      0



sSearchDriver sd;



int contempt = 0;



bool time_over = 0;




enum ettflag {
 TT_EXACT,
 TT_ALPHA,
 TT_BETA
};



enum eproto {
 PROTO_NOTHING,
 PROTO_XBOARD,
 PROTO_UCI
} extern mode;




int search() {
 int in_check;

    smove m[256];
 int bestmove = -1;
 int val = 0;

 ageHistoryTable();
```

```
if ( mode == PROTO_NOTHING ) printSearchHeader();


for (sd.depth=1; sd.depth<=MAX_DEPTH; sd.depth++) {


 if (sd.depth > 1) {
  if (time_over) break;
  if (time_stop_root()) break;
 }


/* Check  extension is done also at  the  root */


in_check = isAttacked( !b.stm, p.KingLoc[b.stm] );
if ( in_check ) ++sd.depth;


 int mcount = movegen(m, bestmove);

 int alpha = -INFINITY;

 for ( int i = 0; i < mcount; i++ ) {


  movegen_sort(mcount,m,i);


  if ( m[i].piece_cap == PIECE_KING ) {
   alpha = INFINITY;
   bestmove = m[i].id;
  }


  //info_currmove(m[i],i);
```

```
  move_make(m[i]);

  /* the "if" line introduces PVS at root */

  if ( i == 0 || -AlphaBeta( sd.depth-1, -alpha-1, -alpha, 1, 0 ) >
alpha )
     val = -AlphaBeta(sd.depth-1, -INFINITY, -alpha, 1, 1);

  move_unmake(m[i]);

  if (time_over) break;

  if ( val > alpha ) {
   alpha = val;
   bestmove = m[i].id;

   tt_save( sd.depth, alpha, TT_ALPHA, m[i].id );

   info_pv( val );
  }
 }

 tt_save( sd.depth, alpha, TT_EXACT, bestmove );

}
```

```
 int mcount = movegen(m, -1);
 for ( int i=0; i<mcount; i++ ) {
  if ( m[i].id == bestmove )
   com_sendmove( m[bestmove] );
 }

    return 0;
}



int AlphaBeta( int depth, int alpha, int beta, int
 can_null, int is_pv ) {

 int  val;
 char bestmove;
 char tt_move;
 int  tt_flag = TT_ALPHA;
 int  in_check;
 int  legal_move = 0;
 int  raised_alpha = 0;



 /* Check for timeout */

 if ( !time_over && !(sd.nodes & 0x3FF) )
  time_over = time_stop();



 /*************************************************************
 *  Are we in check? If so, extend. It also means that program  *
 *  will never enter quiescence search while in check.          *
 *************************************************************/



 in_check = ( isAttacked( !b.stm, p.KingLoc[b.stm] ) );
 if ( in_check ) ++depth;



 /*************************************************************
 *  At leaf nodes we do quiescence search (captures only)       *
```

```
*  to make sure that only relatively quiet positions           *
*  with no hanging pieces will be evaluated.                   *
*************************************************************/



   if ( depth == 0 ) {
 val = Quiesce(alpha, beta);



 if ( LEAVES_IN_TT )
  tt_saveLeaf( alpha, beta, val );



 return val;
   }



sd.nodes ++;



if ( isRepetition() ) return contempt;



if ( ( val = tt_probe(depth, alpha, beta, &tt_move) ) != INVALID )
 return val;

/*************************************************************
*  Here  we introduce  null move pruning. It  means  allowing  *
*  opponent  to  execute two moves in a row,  i.e.  capturing  *
*  something  and escaping a recapture. If this cannot  wreck  *
*  our  position,  then it is so good that there's  no  point  *
*  in searching further. The flag "can_null" ensures we don't  *
*  do  two null moves in a row. Null move is not used in  the  *
*  endgame because of the risk of zugzwang.                    *
*************************************************************/



if ( ( depth > 2) &&
  ( can_null ) &&
```

```
   ( eval(alpha, beta) > beta ) &&
   ( p.PieceMaterial[b.stm] > 1300 ) &&
   ( !in_check )  )
    {



  char ep_old = b.ep;
  move_makeNull();

  val = -AlphaBeta( depth - 3, -beta, -beta+1, NO_NULL, NO_PV );



  move_unmakeNull(ep_old);



  if ( time_over ) return 0;
  if (val >= beta) return beta;
 }
 /* end of null move code */




    smove m[256];
    int mcount = movegen( m, tt_move );

#ifdef USE_KILLERS
 /* reorder killer moves */
 for ( int j=1; j<mcount; j++ ) {



  if ( ( m[j].from == sd.killers[depth][1].from ) &&
    ( m[j].to   == sd.killers[depth][1].to   ) &&
    ( m[j].score < SORT_KILL-1 )  // don't lower the move value
   )
   m[j].score = SORT_KILL-1;

  if ( ( m[j].from == sd.killers[depth][0].from ) &&
    ( m[j].to   == sd.killers[depth][0].to   ) &&
    ( m[j].score < SORT_KILL )
   )
   m[j].score = SORT_KILL;
```

```
 }
#endif


 bestmove = m[0].id;


    /************************************************
 *  Now it's time to loop through the move list    *
 ************************************************/

 for (int i = 0; i < mcount; i++) {


  movegen_sort( mcount, m, i );


  if ( m[i].piece_cap == PIECE_KING ) return INFINITY;


  // problem: we do this test several times, even though
  // king capture is sorted fist

  move_make( m[i] );


/*****************************************************************
  *  The code below introduces principle variation search. It  means *
  *  that once we are in a PV-node (indicated by IS_PV flag) and  we *
  *  have  found a move that raises alpha, we assume that  the  rest *
  *  of moves ought to be refuted. This is done  relatively  cheaply *
  *  by using  a null-window search centered around alpha.  Only  if *
  *  this search fails high, we are forced to do a real one. This is *
  *  not done when the remaining depth is less than 2 plies.         *
  *****************************************************************/


  if (!is_pv || depth < 3) // non-pv node or low depth - don't use pvs
      val = -AlphaBeta( depth-1, -beta, -alpha, DO_NULL, NO_PV );
        else {
```

```
           if (!raised_alpha)
// alpha isn't raised - full width search
               val = -AlphaBeta( depth-1, -beta, -
alpha, DO_NULL, IS_PV );
            else {

// first try to refute a move - if this fails, do a real search
               if ( -AlphaBeta( depth-1, -alpha-1, -
alpha, DO_NULL, NO_PV ) > alpha )
                  val = -AlphaBeta( depth-1, -beta, -
alpha, DO_NULL, IS_PV );
            }
        }

        move_unmake(m[i]);



/*****************************************************************
  *  If  the  move doesn't return -INFINITY, it means that the King
*
  *  couldn't be captured immediately. So the move was legal. In this
*
  *  case we increase the legal_move counter, to look afterwards,
*
  *  whether there were any legal moves on the board at all.
*                                          *
  *****************************************************************
/



  legal_move += ( val != -INFINITY );



  if ( time_over ) return 0;


/*****************************************************************
  *  Beta cutoff - the position is so good that the score will not
*
  *  be accepted  one ply below.
*
  *****************************************************************
```

```
/



        if ( val >= beta ) {
   bestmove = m[i].id; // bugfix 2008-07-18
   sd.history [m[i].from] [m[i].to] += depth*depth;

#ifdef USE_KILLERS

   /* if a move isn't a capture, save it as a killer move */
   if ( m[i].piece_cap == PIECE_EMPTY ) {



    /* make sure killer moves are different
       before saving secondary killer move */
    if ( m[i].from != sd.killers[depth][0].from ||
      m[i].to   != sd.killers[depth][0].to
       )
                   sd.killers[depth][1] = sd.killers[depth][0];



    /* save primary killer move */
       sd.killers[depth][0] = m[i];
   }
#endif



   tt_save(depth, beta, TT_BETA, bestmove);
            return beta;
        }

  /*****************************************************************
  *  We  can  improve  over alpha, so we change the  node  value  *
  *  together with the expected move. Also the raised_alpha flag, *
  *  used to decide about PVS, is set.                            *
  *****************************************************************/




if ( val > alpha ) {
   raised_alpha = 1;
```

```
   tt_flag = TT_EXACT;
             alpha = val;
   bestmove = m[i].id;
        }

    }   // end of looping through the moves



 /* Checkmate and stalemate detection */

 if ( !legal_move ) {
  bestmove = -1;



  if ( in_check )
   alpha = -INFINITY + sd.depth - depth;
  else
   alpha = contempt;
 }



 /* tt_save() does not save anything when the search is timed out */
    tt_save( depth, alpha, tt_flag, bestmove );



    return alpha;
}
```

```
[[code]]
[[code format="cpp"]]
int info_currmove(smove m, int nr) {



 switch(mode) {
 case PROTO_UCI:
```

```
  char buffer[64];
  char move[6];



  algebraic_writemove(m,move);
  sprintf(buffer,
"info depth %d currmove %s currmovenumber %d", sd.depth, move, nr);



  com_send(buffer);
 }
 return 0;
}



int info_pv( int val ) {
 char buffer[2048];
 char score[10];
 char pv[2048];



 if (abs( val ) < INFINITY - 2000) {
  sprintf( score,"cp %d", val );
 } else {
  //the mating value is returned in moves not plies
  if (val > 0)
   sprintf( score,"mate %d",(INFINITY - val+1)/2 );
  else
   sprintf( score,"mate %d",(-INFINITY - val+1)/2 );
 }



 unsigned int nodes = (unsigned int) sd.nodes;
 unsigned int time = gettime() - sd.starttime + 1;



 util_pv(pv);
```

```
    if ( mode == PROTO_NOTHING )
  sprintf(buffer, " %2d. %9d  %5d %5d %s", sd.depth, nodes, time/10
, val, pv);
 else
       sprintf(buffer,
"info depth %d s
core %s time %u nodes %u nps %u
pv %s", sd.depth, score, time, nodes, countNps(nodes, time), pv);



 com_send(buffer);



 return 0;
}




/* this function guards against overflow and allows to display
   correct nps for longer searches. */



unsigned int countNps(unsigned int nodes, unsigned int time) {
    if ( time > 20000 )
  return nodes / (time/1000);
 else
     return (nodes*1000) / time;
}



int isRepetition() {



 for (int i=0; i < b.rep_index; i++) {
  if (b.rep_stack[i] == b.hash)
   return 1;
 }
```

```
 return 0;
}
```

```
void clearHistoryTable() {
     for (int i = 0; i < 128; i++)
   for (int j = 0; j < 128; j++) {
             sd.history[i][j] = 0;
   }
}
```

```
void ageHistoryTable() {
     for (int i = 0; i < 128; i++)
   for (int j = 0; j < 128; j++) {
             sd.history[i][j] = sd.history[i][j] / 8;
   }
}
```

External calls:

- int movegen();
- int move_make(move);
- int move_unmake(move);

Up one Level