How REBEL Plays Chess[1]

by Ed Schröder

In this document I will try to explain some of the secrets of REBEL, I have retired from the competition so I see no need any longer to hide my ideas on computer chess. It's my hope that some of my fellow programmers find something useful on this page and that it might contribute to increase the elo rating of their chess engine.

Also, if you have any question about the below stuff, **BETTER NOT** contact me by email, the response might be disappointing due to my lack of time, rather post your question on the CCC forum, it's the place where chess programmers around the world gather and discuss computer chess related topics. You need to be a member, if you aren't then signup here.

Move Ordering

Contrary to most chess programs, when going one ply deeper in the chess tree REBEL generates **all moves** for the given (and new) ply. During move generation REBEL quickly evaluates each move generated by using the **piece-square** technique, this evaluation is only used for move ordering later, it is certainly not for evaluating the position.

The advantage of this time consuming process is that it pays off later when move ordering becomes an issue, this provided you have created 12 reasonable organized **piece-square** tables for the move generator. REBEL's data structure for move generation looks as follows:

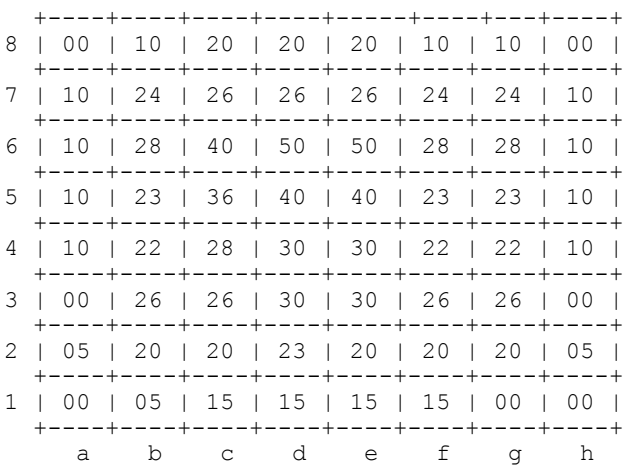
```
static char move_from  [5000];
static char move_to    [5000];
static char move_value [5000];
```

So when adding a new generated move to "move_from" and "move_to" REBEL also will fill "move_value" via the corresponding piece-square table using the formula:

```
move_value = table[move_to] - table[move_from];
```

1 This document is a PDF conversion of the webpage at http://members.home.nl/matador/chess840.htm. Look there for up to date information.

An example for the White Knight:



	a	b	c	d	e	f	g	h
8	00	10	20	20	20	10	10	00
7	10	24	26	26	26	24	24	10
6	10	28	40	50	50	28	28	10
5	10	23	36	40	40	23	23	10
4	10	22	28	30	30	22	22	10
3	00	26	26	30	30	26	26	00
2	05	20	20	23	20	20	20	05
1	00	05	15	15	15	15	00	00

Ng1-f3 will give 26 - 00 = +26
Nc3-d5 will give 40 - 26 = +14
Nf3-h4 will give 10 - 26 = -16
Nf3-e1 will give 15 - 26 = -11

Ng1-f3 usually is a good move, so is Nc3-d5, both get a high value. Nf3-h4 and Nf3-e1 are usually not so good moves and both get a low value. The system works like a charm for move ordering, all generated moves get a value and every time the search needs a next move simply the next highest one from the list is taken.

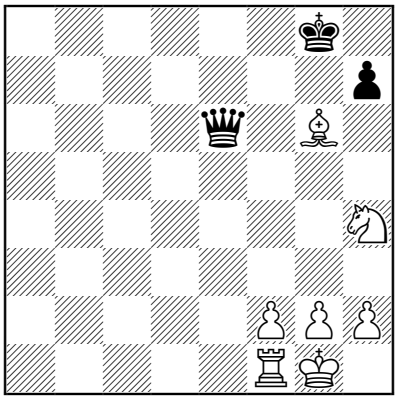
Last time I removed the code REBEL ran about a factor of 2 slower. It's of course very important the 12 piece-square tables are in harmony with each other.

Now that REBEL has generated all moves (with values) it moves to the second step, it will update the values of moves which are more important then others, thus further sort the moves to improve move ordering. Its final ordering looks as follows:

- The move from the Hash Table, it will get the highest value +127 to ensure the move is searched first.

- The **Mate-Killer-Move**, when present it will get the value +126. A **Mate-Killer-Move** is a normal **Killer-Move** coming from the 2 slots REBEL uses for killer moves, more below. However when a Killer-Move has produced a mate-value in the tree it is promoted to Mate-Killer-Move and thus searched as second move. It works very well in positions where mate threats are an issue.

- The **Winning Capture** moves. Moves that are winning material are searched third. REBEL maintains a sorted list of the so called **hanging pieces** with a maximum of 3 squares, the 3 squares are sorted on the expected material gain. Each move in the move list that captures on a hanging-sqaure is rewarded with 124, 123 or 122 depending on the status of the expected material gain. If you don't have such stuff in your program yet make sure you do, it is good for a tremendous speed-up of your program. The technique most programs use for this is called: SEE (Static Exchange Evaluator).



HINT: when it is possible that 2 or more moves can capture a "hanging-piece" make sure you search the move first that captures the piece with the lower piece-type, have a look at the diagram.

Black can capture the hanging-piece on square G6 with 2 pieces, the pawn and the queen, it's obviously better to search the lower piece-type capture (pawn) first then the higher piece-type capture (queen) when the issue is move ordering. By head I remember it speed-ups by some 5% doing so.

- Queen promotions with capture come next, value 121, given in the move generator. Most of them become automatically part of the **Winning Material** moves, see above.

- Normal Queen promotions (no capture) are next, value 120, given in the move generator. During Phase II REBEL makes sure they end up in the 125-122 area.

- Next are the so called **Good Captures**, these are the following captures: QxQ (119), RxR (118), NxB (117), BxB (116), BxN (115) and pxp (114). This is all done in the Move Generator part.

- Next are the so called **Killer Moves** maintained in 2 slots during Search. Killer Moves are the best moves found on a given depth in the Search. The idea behind Killer Moves is that if a given move brings the score above ALPHA it is usually a good thing to try the same move again when the Search reaches that ply again.

REBEL uses 2 slots, Killer-One and Killer-Two, both are depth based, Killer Moves get the following values:

```
Killer-One [current ply]      110
Killer-One [current ply-2]    108
Killer-Two [current ply]      106
Killer-Two [current ply-2]    104
```

HINT: most important while maintaining your killer moves during Search, never ever allow the following moves in your Killer Slots:

1. Moves that win material (see hanging pieces), they are polluting your valuable killer slots.
2. Good captures (see above), they are polluting your valuable killer slots.
3. Promotions, same reason.

REMARK: the above is true for REBEL, it might be different in programs who use a

different move ordering scheme, nevertheless it can't hurt trying.

LAST: needless to say, but.... make sure you don't have any double moves in your 2 killer slots.

- Next in move ordeing is the **Historic Mate Killer**, note it gets the value 109 so right after the first Killer Move, see above. You can safely skip this, it gives only a small speed-up of 3-4%. To make it work you will need to create 2 x 32-bit two-dimensional tables, one for white and one for black, for example:

```
static int Historic_Mate_Killer_White [12][64];  // [piece-type][square]
static int Historic_Mate_Killer_Black [12][64];
```

During Search (not in Q-Search) whenever you find a mate-score you increment the table with the **remaining depth** (horizon depth - current depth) and maintain the highest number as **best historic-mate-killer** move. Note the table(s) are piece-type-square based, so not square-square based. The advantage of this formula is that you get more hits then when using the square-square approach.

- We are still not finished, next in move ordering are the two castling moves: 0-0 (103) and 0-0-0 (102), all done in the Move Generator.

- Next are all the minor promotions, they get the value 101 in the Move Generator.

In principle that's it, REBEL's move ordering. However 2 fine correction principles are added, for all moves below the value of 70 a correction of 30 points might be added or subtracted, the principle works as follows:

1. Subtract 30 points if the move gives away material, for instance if the queen moves to a square that is covered by the opponents pawn, bishop etc. The data for that is directly available because of REBEL's concept, if you don't have this data available in your program it will be most probably too costly to try this move ordering feature.
2. Add 30 points if a piece that is hanging moves, it is usually good.

Let me try to put all of this in a nutshell before going to the next part, whenever REBEL goes one ply deeper in the chess tree it does the following:

- Generate_all_Moves();
- Update_all_Moves();
- Get_highest_Move();
- Do_something_till_all_moves_are_done();

Another move-ordering mechanism handles the so called large unsorted trees. An unsorted tree occurs when "Update_all_Moves" (see above) does not report:

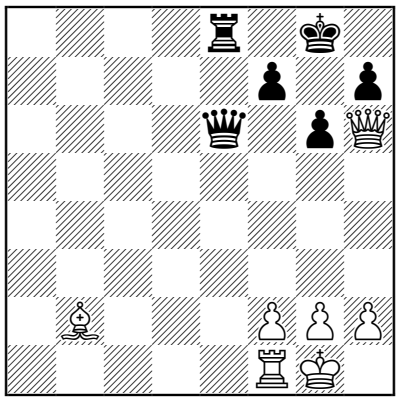
- A move from the Hash Table, or
- A **Winning Capture** move, see above.

REBEL then will lower the remaining depth to search with a factor of 2 and search the subtree first with **this depth** before searching the unsorted tree at **full depth**. The larger the unsorted tree the more powerful the algorithm performs. The speed-up in general is about 13-14% but at longer time controls will certainly increase.

There is room for improvement here, there could be more powerful formula's such as going through the unsorted tree in steps of 2 plies (or one) till the actual depth is reached, one should try.

All in all it makes REBEL's move ordering to look like this:

- Generate_all_Moves();
- Update_all_Moves();
- Reduce_depth_in_case_of_an_unsorted_tree();
- Get_highest_Move();
- Do_something_till_all_moves_are_done();



There is another move ordering trick worth to mention. REBEL has special code to recognize mate-threats in its evaluation part (EVAL), have a look at the diagram. It's obvious with white to move white can give a checkmate with Qg7# on the next ply.

When REBEL recognizes such a pattern in EVAL it doesn't even bother to search further, EVAL just returns the mate value avoiding to go deeper into the tree, it gives a nice speed-up depending on the number patterns the program knows.

But there is more to gain, when it is black to move it still can defend itself against the Qg7 mate threat, as in this case black can defend itself perfectly with 1..f6 (or even 1..Qe5 and Qf6) but all other moves lead to checkmate with Qg7#. The trick is to search the Qg7 move right after the Hash Table move in move ordering, a high chance it will lead to mate.

This ends the description of REBEL's move-ordering mechanism, if there is more to mention (I am sure I am have forgotten a few things) this page will be updated, let's move now to the next chapter, Search Techniques.

Search Techniques: Alpha-Beta

PREAMBLE, to understand this issue it is assumed you have a working chess program with a working searh algorithm. This topic is not meant to be an introduction to Search, for that I refer to other excellent web-pages on the Internet, to name a few:

- The home page of Bruce Moreland, excellent for starters.
- The work of Aske Plaat, more advanced techniques.

REBEL uses the standard **Aspiration Alpha-Beta Search** technique with 0.50 as basic value, no null-window tricks or whatsoever, just the bald algorithm. There is very little to say about this issue, there are only two small modifications worth to mention, both give a nice speed-up.

- Whenever REBEL gets a **fail-low** or **fail-high** at the root position (we are back at ply one), REBEL doesn't immediately opens the entire alpha or beta window, instead of that it tries a new window of 2.00 and only then when REBEL is faced again with a **fail** it opens the complete (corresponding) window. The 2.00 window will do for most of the positions when a **fail** occurs, that's the tought behind.

- REBEL keeps track of the best score of the previous ply, using that value we can do a little trick. Normally Aspiration-Search when it finds a new main-variation (best move) the BETA-WINDOW is either closed (null-window) or left untouched (standard). REBEL instead does the following,

It measures the difference in score from the previous iteration and when there is not too much decline in score BETA is narrowed, not null-windowed. Using the initial position as an example,

At the end of iteration 5 the best move is 1.e4 score 0.20, we save the 0.20 score, we go to iteration 6, ALPHA is set to -0.30, BETA is set to +0.70 and we search 1.e4 again this time with a depth of 6, we get a new main-varition and new score for 1.e4 say 0.15

Since there is only a small decline in score (0.20 to 0.15) or in case the score has gone up it is assumed safe to narrow BETA, REBEL's new window will look as follows: ALPHA=0.15 and BETA=0.25 (ALPHA + 0.10). The advantage is that possible new best moves do not automatically will fail-high, the second advantage is that BETA is narrowed with 0.45 which is quite a lot.

In the case the decline in score is bigger (say 0.20 and up) it is still unlikely a new best move will be higher than 0.20 so REBEL will narrow BETA anyway, only less, with 0.25, so BETA becomes 0.45 (previous score + 0.25).

Search Techniques: Lazy Eval

Definition, **LAZY EVAL** (abbreviated to LE) is a surrogate routine for EVAL that with a few instructions tries to estimate the value of EVAL.

Its goal, why do a time consuming EVAL (of hundreds, maybe thousands of instructions) if you can guess the score of EVAL with only a few instructions, thus speed-up the program.

Where to use LE, only at horizon depth and Quiescent-Search (QS) plies.

Is LE safe? No, in fact it is a **most dangerous** piece of software, don't ever use it in its original form, use either REBEL's modification or find solutions for the drawback of LE yourself, I will try to point out the danger of LE later, let's first explain how LE works in practice.

When you are at the horizon depth or in QS, try LE first before you call EVAL, it might save you from doing an expensive EVAL. Actually in REBEL LE is called **BEFORE** "Make_Move" trying to win even more processor time, the pseudo code:

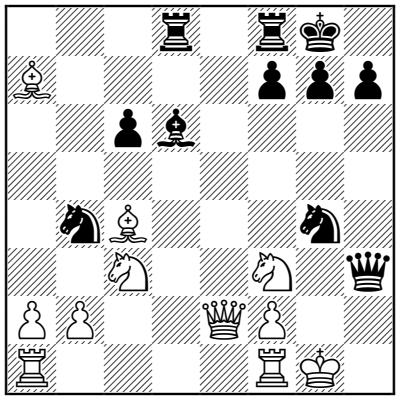
```
if (own_king_is_in_check)              -> don't try LE, too dangerous.
if (move_does_check_the_opponent_king) -> don't try LE, too dangerous.
if (special_case_position)             -> don't try LE, description below.
Lazy_Eval();                           -> calculate (guess) score, more below.
if (ALPHA < SCORE)                     -> don't reward LE, do normal EVAL.
else: reward the LE SCORE as the real score from EVAL and skip EVAL.
```

The routine "Lazy_Eval" calculates the all the material on the board plus its piece-square values in **I_SCORE**, actually REBEL has I_SCORE always at hand since it is **incremental maintained** in "Make_Move" and "Undo_Move" and always is up-to-date.

HINT, if you don't have such a variable in your program yet it's advised to make it, as I_SCORE is handy to use in many other places in your program, for instance in EVAL as it saves you a lot of processor time, but this aside.

The next step is to update I_SCORE with the move, exactly like you do in "Make_Move". Then you add-up a MARGIN to I_SCORE of say 0.50 and return I_SCORE as SCORE and then make the compare to ALPHA. MARGIN is an estimated value of all the positional aspects of EVAL and it is assumed that EVAL stays in that boundary, +0.50 in our example.

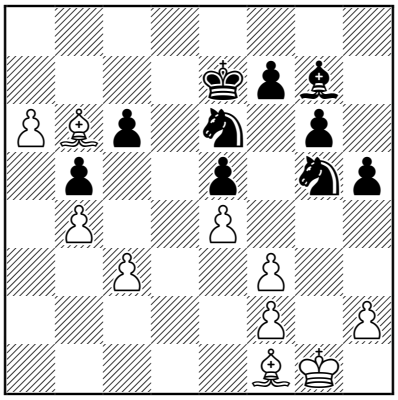
The idea of LE, if I_SCORE + MARGIN can't make it to ALPHA why bother to do an expensive EVAL, the move isn't any good, just use the score from LE and skip EVAL. The system works like a charm, however we are now ready to point out its main drawback, that is the value of **MARGIN**.



Consider the left diagram, if you have any good "king safety" in your program its value will be certainly over 1.00 or more rewarding the black attack on the white king thus, EVAL will result in a score where the **material balance** - the **positional aspects** exceed the value of 0.50, the value of MARGIN.

Houston we have a problem, LE returns a wrong value, nevertheless LE is rewarded, as practice has learned me the results are disastrous when king safety (with scores above 0.50) is dominant in the search, actually when MARGIN is exceeded too many times it will ruin all your chess knowledge and the program will start playing bad moves.

Second example, see diagram at the right, same story. If you have any reasonable **passed-pawn evaluation** it will consider the white pawn at A6 as extremely dangerous and accordingly evaluate that giving it a big score. Again the LE algorithm is going to fail tremendously, resulting in bad moves.



Solutions are many, simply increase the value of MARGIN to 2.00 or even 5.00, but doing that you will lose much of the power of LE as you force the program to do more full EVAL's. Here is the compromise I found for the problem, it works very well for REBEL, but you must have the score of the depth-1 EVAL at your disposal.

The default value of MARGIN in REBEL is 0.50, however the difference of **I_SCORE - SCORE** of the **depth-1 evaluation** is added to MARGIN. I_SCORE - SCORE of one ply back in the tree in most cases will correctly represent what's going on on the board, its value is highly reliable and thus MARGIN is updated with this value.

This is just another example how useful it can be to do EVAL on every ply in the main search, in this case it solves the main drawback of LE while keeping MARGIN very low.

```
if (special_case_position) -> don't try LE, description below.
```

This still needs to be explained, these are a few excpetion cases where the I_SCORE - SCORE of the depth-1 evaluation trick doesn't work or those cases I don't want to rely on LE because I find the situations too dangerous, here is the list:

- REBEL doesn't use LE when on the evaluation of depth-1 the quadrant-rule is used, the quadrant-rule is practiced in "pawn-endings", in case the king can't stop an opponent pawn from promoting the pawn is evaluated close to the value of a queen. If this happens, no LE is allowed one ply deeper.

- Other typical special end-game stuff, such as KPK endings where REBEL also knows when the pawn will make it to promotion.

- In case of the presence of a Mate Threat Killer.

Some more thoughts on lazy eval:

- It's highly questionable if LE is of any use if you have a cheap evaluation with only few chess knowledge, I wouldn't know for sure because REBEL never had a cheap and fast evaluation. The power of LE lies in a big fat evaluation routine, for the latest REBEL the speed-up factor of LE is 3.2.

- It's a strange mechanism, the bigger the evaluation routine, the more the speed-up factor of LE will increase. In this way it is not so bad to have an expensive evaluation routine, adding new chess knowledge is relative cheap because of LE.

- The drawbacks of LE are only few, at least if you practice it in the way REBEL does, this includes that you must have the evaluation of depth-1, if you don't have it and have a big fat evaluation routine yourself, consider it as useful to do full evaluations starting at horizon-1 depth and onwards, it might give your program a real push, I sincerely hope so.

- Last, if you have one of the latest REBEL versions you can experiment with LE, in the Personalities is a parameter called [ChessKnowledge = 100]. When you increase this value to 200 it will set the default MARGIN of 0.50 to 1.00, when you use [ChessKnowledge = 400] MARGIN is set to 2.00, when you use [ChessKnowledge = 500] MARGIN is set to infinite, meaning LE is not used at all. Using [ChessKnowledge = 500] you can check how well LE works in REBEL and that the side-effects are about to neglect.

Search Techniques: Futility Pruning

Futility Pruning is the orginal idea of **Ernst Heinz** practiced in his chess playing program DARK THOUGHT, it's well documented on his own pages.

The idea has been adapted in REBEL and improved using the **LAZY EVAL** technique as a base, see description above.

Futility Pruning is done at **horizon-1 depths** and **horizon-2 depths**, it basically comes down to LAZY EVAL only that MARGIN is increased. So when SCORE + MARGIN can't make it to ALPHA the subtree is pruned, its pseudo code:

```
if (current_depth == horizon_depth-1) then {
  if (own_king_is_in_check)          -> don't try FUTILITY, too dangerous.
  if (move_checks_the_opponent_king) -> don't try FUTILITY, too dangerous.
  if (move_is_a_capture)             -> don't try FUTILITY, too dangerous.
  if (special_case_position)         -> don't try FUTILITY, see Lazy Eval.
  Futility_One();                    -> calculate (guess) score, more below.
  if (ALPHA < SCORE + MARGIN)        -> don't reward FUTILITY, just proceed.
  else: prune tree (don't go deeper) using SCORE as the real score.
}
```

The routine "Futility_One" does exactly the same as the rountine "Lazy_Eval" (see above). MARGIN however instead of 0.50 is set to a higher value, basically 3.00 (3 pawn units) but is increased to 5.00 in the end-game. Another difference in comparison with LAZY EVAL is that SCORE isn't updated with MARGIN.

About exactly the same is done at **horizon-2 depths**, only its MARGIN is further increased to avoid errors, its pseudo code:

```
if (current_depth == horizon_depth-2) then {
  if (own_king_is_in_check)          -> don't try FUTILITY, too dangerous.
  if (move_checks_the_opponent_king) -> don't try FUTILITY, too dangerous.
  if (move_is_a_capture)             -> don't try FUTILITY, too dangerous.
  if (special_case_position)         -> don't try FUTILITY, see Lazy Eval.
  Futility_Two();                    -> calculate (guess) score, more below.
  if (ALPHA < SCORE + MARGIN)        -> don't reward FUTILITY, just proceed.
  else: prune tree (don't go deeper) using SCORE as the real score.
}
```

The routine "Futility_Two" does exactly the same as the rountine "Futility_One" (see above). MARGIN is set to 5.00 (5 pawn units) as standard value.

Futility Pruning was good for a speed-up of 22% in REBEL with only very few side effects, hardly any.

Search Techniques: Horizon

At **HORIZON-DEPTH** after evaluation of the position it is decided to go to Quiescent-Search (QS) or not. There are a couple of tricks that will avoid REBEL doing unnecessary expensive Q-searches, but let's examine the base code first before going into detail.

```
if (Lazy_Eval_is_true)               -> return score, no eval, no QS
Evaluate_Position();
```

```
if (current_depth == maximum_depth)    -> return score, no QS
if (Trick_one_is_true)                 -> return score, no QS, more below
if (move_does_check_the_opponent_king) -> Do QS
if (ALPHA >= SCORE)                    -> return score, no QS
if (Trick_two_is_true)                 -> return score, no QS, more below
Quiescent_Search();                    // QS coding
```

Trick_One gives about a 10% speed-up, its thought behind: The goal of QS **mainly** is to see if a piece is hanging (en-prise), based on that thought it is likely the score will not go down further than the value of the hanging piece itself.

So it must be able to calculate a margin for calling QS or else just return the score and thus not call QS at all. The formula in pseudo code:

```
MARGIN = 3.00                              // 3 pawns safe-guard value
MARGIN + highest hanging piece value       // Queen=900, Rook=500 etc.
MARGIN + 9.00 when own_king_was_in_check_before_make_move
MARGIN + 6.00 when the opponent can promote the next ply
if (SCORE-MARGIN > BETA)                   -> return TRUE
else return FALSE
```

Trick_Two gave about a 5-8% speed-up, I don't quite remember exactly, its thought behind: if the score is already 9.00 above BETA don't bother to do QS, it won't matter, the score is way too good, its pseudo code:

```
if (own_king_was_in_check_before_make_move) -> return FALSE (too risky)
if (opponent_can_promote_the_next_ply)      -> return FALSE (too risky)
if (SCORE-900 > BETA)                       -> return TRUE
else return FALSE
```

IMPORTANT: Naturally the 2 tricks can be practiced in QS itself too!

Search Techniques: Reductions

REDUCTIONS are the opposite of extensions. Extensions extend the search with one ply, reductions do the opposite, reduce the search with one ply. Reductions are very powerful, they speed-up the search tremendously. However the reductions-business is a wasps' nest, when you reduce too much or use wrong formula's your engine goes down in strength rapidly, so be extremely careful using reductions.

I would like to introduce some of the reduction techniques I use in REBEL which I consider safe, at least in the REBEL concept, realize they might work counter productive in your engine.

REDUCTION-1: In the main-search (not Q-search) after REBEL has called "Make_Move();" it investigates if the move is candidate for a reduction. REBEL uses the following formula:

```
if (remaining_depth > 2
    && own_king_was_not_in_check_before_make_move
```

```
    && move_is_no_capture
    && move_does_not_check_the_opponent_king) then    // such as Bf1-b5+
     { if (ALPHA > SCORE + MARGIN) reduce depth with 1 }

SCORE  = The score of the position (material value +
         piece-square value's)

MARGIN = TABLE [remaining_depth];

static int TABLE[] = {   0,    0,    0,  500,  500,  700,  700,  900,
                       900, 1500, 1500, 1500, 1500, 1500, 1500, 1500,
                      1500, 1500  ............................. };
```

Furthermore REBEL makes sure this type of reduction is only used once. The basic thought behind the idea is to reduce the search with one ply if the side to move is already behind a rook or worse depending on **remaining depth**. The effect is that for positions near the root REBEL will use a MARGIN of 15.00 (pretty safe) and deeper in the tree a MARGIN of 5.00 (more risk).

The reduction in practice is pretty safe, sometimes a tactical shot is seen one ply too late but it outweighs the advantage of the nice speed-up of 15% I got.

REMARK: for clearness sake, *remaining_depth = horizon_depth - current_depth*, thus the number of plies to go to the horizon.

OTHER REDUCTIONS: REBEL does several other reductions, to understand them you will need to know a little bit more about the quite different approach of REBEL's Search Algorithm in comparison to other chess programs, I therefore highly question the relevance of this topic as I assume that it can't be used in other programs without drastic changes, nevertheless here goes...

When we (for example) are at iteration 8, all the positions in the tree till the horizon are evaluated by REBEL's normal EVAL routine where the complete chess knowledge is. This is a costly operation, but (for REBEL) it pays off because it allows all kind of **static evaluation tricks**.

Mind you, if you have the complete evaluation at your disposal (the score, hanging pieces for both sides, king safety for both sides etc. etc.) and also have most of this information available for [current_depth-1] and [current_depth-2] all the way to the root position, stored on the stack, there are many static tricks you can do.

Actually the system has been (and still is) the sole base for REBEL's Selective Search approach, it allows reductions, prune complete subtrees, more accurate extensions, avoids needless null-move searches. More later, for now let's stick to the topic reductions.

REDUCTION-2a: When we are in the main-search (not Q-search) and have evaluated a position it is checked for being candidate for a reduction, to clarify where we are in the tree some free-style pseudo code:

```
Make_Move();          // update board position
Evaluate_Position();  // do a complete evaluation
Reduce_Depth();       // reduce one ply (Y/N)

Continue.....
```

"Reduce_Depth" first checks for a list of exceptions, it checks for situations a move never ever is considered being candidate for a reduction, the list:

```
the_move_is_already_reduced
own_king_was_in_check_before_make_move
move_does_check_the_opponent_king         // such as Bf1-b5+
move_wins_material                        // winning_capture
move_increases_pressure_on_opponents_king // see remark below
```

The coding for **move_increases_pressure_on_opponents_king** is tricky tricky, you must have some kind of code that recognizes mate-threats, that meassures if a move makes progress attacking the opponents king in a dangerous way, you can't afford to reduce moves like that. For REBEL it's not so time consuming, most of the data is directly available from the evaluation, issue King_Safety.

"Reduce_Depth" does check for several cases, reduction-2a goes as follows, when the score (from EVAL!) is already below ALPHA the move (or position) is candidate to be not so good, but then it shows up that the position has a direct threat, for instance it attacks the opponents queen (value 900) which would bring the score above ALPHA, then maybe the move is not so bad after all.

It is then checked if we divide the threat by 4 (making it 2.25 in this case) would again bring the score above ALPHA, if not the move is candidate for a reduction, the assumption is made the move will never make to ALPHA, the depth is reduced by one.

For reduction-2a there is another safe-guard (exception) rule which I practise on more occassions in REBEL. The idea is to check the hash table and gain some extra information on the position. The scores in the Hash Table are of course not so reliable because of the ALPHA/BETA algorithm but it may give a clue after all and just for an extra safe-guard check it can't hurt. The pseudo code:

```
(1) if (current_move_is_the_move_from_the_hash_table) - do not reduce
    if (current_position_is_not_in_the_hash_table)    - reduce
(2) if (ALPHA < hash_table_score)                     - do not reduce
(3) if (hash_table_score - evaluation_score > 0.50)   - do not reduce
    else                                              - reduce
```

The idea behind the code is another check for the credibility of the move, if it turns out to be the best move from the Hash Table it might be risky after all to reduce here, see (1), secondly since the position is known in the Hash Table and its score strangely enough is higher than ALPHA, better not reduce, see (2), thirdly the hash table score is half a pawn higher than the evaluation score, so the subtree probably has made progress in score in the tree, see (3).

While it is true that the hash table score in (2) and (3) is probably maltreated by the behaviour of the ALPHA/BETA algorithm it is not so bad to pay attention in this case, thus the move is not reduced.

REMARK: if memory serves me well the reduction gave a 18% speed-up, there are of course the usual drawbacks (every reduction has) but it in general it was a clear improvement.

REDUCTION-2b: we are still in "Reduce_Depth", going to the next formula detecting possible reductions. Reduction 2b is only done in the last "x" plies till the horizon (remaining depth). "x" varies in REBEL, although the definition of "x" is more sophisticated in REBEL it in general comes down to:

```
middle game     : maximum is 8
early end game  : maximum is 6
end game        : maximum is 4
late end game   : maximum is 3 // rook endings, B/N endings
pawn ending     : maximum is 2
```

"x" is defined again after each iteration, table driven, its formula for the **middle-game**:

```
x = table_for_mid_game [iteration_depth];

static char table_for_mid_game[] = {0,1,1,1,2,2,3,3,3,4,4,5,5,6,6,7,
                                    8,8,8,8,8,8,8,8,8,8.... };
```

You get the picture for the other tables, the idea is not only to limit "x" to 8, but also to excuse the early iterations from the reductions, a safety guard. So when we are in the last "x" plies of the search we try reduction-2b,

```
if (remaining_depth<=x && remaining_depth>1) then {
   if (ALPHA > SCORE + THREAT &&
       ALPHA < SCORE + THREAT + MARGIN) -> reduce depth with one ply.
}

SCORE  : score of EVAL
THREAT : Queen=900, Rook=500, Bishop=300, Knight=300, Pawn=100
MARGIN : TABLE [remaining_depth];

static int TABLE[]= { 00,00,10,15,20,25,25,25,25,25,25,25,25,25,
                      25,25,25,25,25,25,25 ........... };
```

The idea is, if SCORE+THREAD are not going to make it to ALPHA, but with an extra small MARGIN it will then reduce the depth. I can't remember the speed-up this reduction gave.

REDUCTION-2c: is very similar to 2b, here goes:

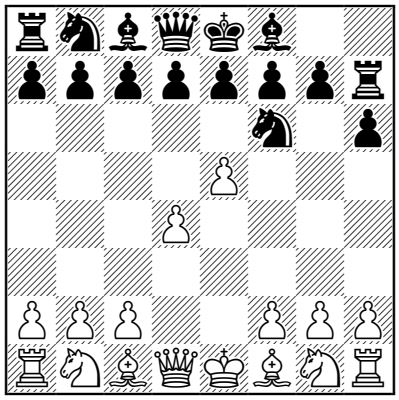
```
if (remaining_depth<=x && remaining_depth>2) then {
   if (ALPHA > SCORE + THREAT &&
       ALPHA < SCORE + THREAT + MARGIN) -> reduce depth with 2 plies.
}
```

```
SCORE  : score of EVAL
THREAT : Queen=900, Rook=500, Bishop=300, Knight=300, Pawn=100
MARGIN : TABLE [remaining_depth];

static int TABLE[]= { 00,00,20,30,40,50,50,50,50,50,50,50,50,50,
                      50,50,50,50,50,50,50 ........... };
```

The only differences are, a double margin, the remaining_depth must be greater than 2 and that the depth is reduced by 2 plies.

REDUCTION-2d: consider the diagram position, white to move. Obviously 1.exf6 is the best move, all other moves make little sense.



1.exf6 is a **winning capture**, reduction-2d is about reducing the depth for moves who **do not capture on F6** the easy material gain, moves such as 1.a3 1.a4 1.b3 and so on are reduced.

The idea looks great at first glance but is full of stings and some special safe-guard coding is needed before rewarding the reduction.

First it is measured if progress has been made in attacking the opponents pieces, for 1.a3 this is not true, however for 1.Bd3 this is true as it attacks the rook on H7, so 1.Bd3 is not reduced.

This measuring for progress is easy, REBEL has all the relevant information stored on its depth driven stack because it evaluates every position in the tree, see elsewhere on this page.

The pseudo code for reduction 2d, note it includes the hash table safe-guard check coding, see elsewhere on this page.

```
if (winning_capture_is_present ...BUT... move_does_not_capture) then {
   if (threat_progress_is_made)                       -> do not reduce
   if (remaining_depth <= 2)                          -> do not reduce
   if (current_move_is_the_move_from_the_hash_table) -> do not reduce
   if (current_position_is_not_in_the_hash_table)    -> reduce
   if (ALPHA < hash_table_score)                      -> do not reduce
   if (hash_table_score - evaluation_score > 0.50)   -> do not reduce
   else                                               -> reduce
}
```

REMARK: REBEL counts every reduction it makes, it also has a flexible paramater in which you from the interface can define the **maximum_number_of_reductions**. Before making a reduction this maximum is checked first.

The default value of the maximum is set to 99, meaning unlimited reductions. I think

there is some room for improvement here, for instance make the maximum number iteration driven, see example pseudo code:

```
maximum_number_of_reductions = table_for_reductions [iteration_depth];

static char table_for_reductions [] = { 0,1,1,1,1,2,2,2,2,3,3,3,3,4,4,
                                        4,4,5,5,5,5.... };
```

This ends the description of REBEL's reduction mechanisms, let's move now to the next chapter, Extensions.

Extension Techniques: Checks

INTRODUCTION: Extensions are very powerful, yet if you use too many extensions it will blow up the Search and produce a very bad branch-factor.

In the early days of computer chess when you only had a slow 5 Mhz processor at your disposal and you only could hit 5-7 plies at tournament time control, extensions were dominant, you needed them, and a lot of them too, to produce reasonable moves at such low depths.

Nowadays, having fast PC's, the need to use a lot of extensions has declined considerable, it is **in my opinion** better to focus on a **low branch factor** then to have a lot of extensions.

Mind you, if you are able to produce a chess program with an effective branch-factor of 2.0 then every doubling in processor speed will give you an extra iteration, in most cases good for an increase of +30-50 elo points in the **computer-computer competition**.

However, you still can't do without extensions, the **check-extension** is obliged to have, without the check-extension the elo of REBEL would drop 100-150 elo points, I am pretty sure of that. The **re-capture extension** as practiced in REBEL is good for +30-40 elo points, the remaining ones REBEL has are debatable, I use them but their elo strength is hardly measurable.

The **check extension** in REBEL is rewarded in MakeMove(); when it is recognized that the side to move **moves out of check**, the depth is extended with 1 ply, the common procedure in most chess programs.

An exception is made for the **first check**, it is not extended, also the **fourth check** is extended with 2 plies being in sync again with common procedure to extend every out_of_check situation with one ply. The idea behind: when there is only one check in the tree, it is probably not so important, thus skip it. However when you have 4 checks in the Search, the chance is big that checks play an important role, better get in sync, thus extend 2 plies.

Skipping the "first check" is very time sensitive, it speed-up REBEL with 25%, however its elo gain is very small in comparison with the common procedure to extend every out_of_check situation, for REBEL the gain is about +5 elo, you must find out yourself if the idea works in your own engine.

Through the years many experiments have been tried to improve the formula, about every year I have tried 3-5 new formula's, it was all in vain, there is a proverb that says, "Genius is the ability to reduce the complicated to the simple", this seems to be very true for extending checking moves, just extend with one play in every out_of_check situation.

Extension Techniques: Recaptures

The sense of recaptures is an old discussion among chess programmers, are they useful or not? For REBEL it certainly pays off, I have used them since the early days and still use the recapture extension. Without the recapture extension REBEL will run a factor of 2.2 to 2.5 faster, so it is quite an investment, still it is good for an elo of +30-40, let me try to present how it works.

REBEL has no limitation on recapture extensions. However there is a (flexible) WINDOW the recapture should fit in, which at the root is set to +5.00 / -5.00 and narrowed each ply the Search goes deeper, this to a minimum value of +2.00 / -2.00, the names of the 2 variables: **HIGH** and **LOW**, its pseudo code:

```
static int LOW,HIGH;

I_SCORE = Sum_Material_plus_Piece-Square_Values // see elsewhere
HIGH = I_SCORE + 5.00                          // pre-fill HIGH and LOW
LOW  = I_SCORE - 5.00                          // before the search.

HIGH = HIGH - TABLE [current_depth]            // narrow HIGH and LOW when
LOW  = LOW  + TABLE [current_depth]            // going one ply deeper.

HIGH = HIGH + TABLE [current_depth]            // restore HIGH and LOW when
LOW  = LOW  - TABLE [current_depth]            // climbing back one ply.

static int TABLE  00,00,50,50,50,50,25,25,     // narrow HIGH and LOW till
                  25,25,00,00... };            // 2.00 at ply=9 and onwards.
```

The idea is of course to extend only those recaptures which are close to the material balance of the root position. REBEL deliberately uses 2 new variables (HIGH and LOW) since it is not wise to rely on ALPHA or BETA here, this because of possible **fail-high** and **fail-low** cases at the root that might occur and suddenly all kind of unwanted recaptures are extended because of the low value(s) of ALPHA and/or BETA.

The recapture extension is rewarded in "Make_Move", its pseudo code:

```
if (move_already_extended)                    -> do not extend
if (previous_ply_was_no_capture)              -> do not extend
```

```
if (move_does_not_capture_on_the_same_square) -> do not extend
if (move_is_not_a_winning_capture)            -> do not extend, see elsewhere
if (I_SCORE < LOW)                            -> do not extend
if (I_SCORE > HIGH + piece value depth-1)     -> do not extend
else: extend tree with one ply.
```

piece value depth-1 is the value of the captured piece of the previous ply (current_depth-1) to widen HIGH. Also here through the years many experiments have been tried to improve the recapture extension, this seems to work best.

Extension Techniques: Pawns

White Pawns that reach the 7th rank and Black Pawns that reach the 2th rank are extended with one ply.

Extension Techniques: Endgame

In the endgame 2 extensions are recognized and rewarded:

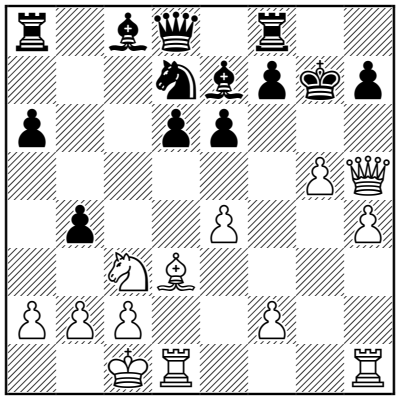
- When (during Search) a transition occurs to the **simple ending** the search is extended with one ply. A simple ending is defined as a rook and/or bishop/knight ending, it is useful to entend then.

- The Search is extended with **3 plies** when the search transits to a pawn-ending. An extra window check of 3 pawns is done on I_SCORE before the extension is rewarded, its pseudo code:

```
if (no_capture)                  -> do not extend
if (no_pawn_ending)              -> do not extend
if (I_SCORE > +3.00)             -> do not extend
if (I_SCORE < -3.00)             -> do not extend
else: extend tree with 3 plies.
```

The extension is very powerful, it will often avoid REBEL entering a lost ending and vice versa.

Extension Techniques: King Safety



An extension is done when a move seriously increases its chances to attack the opponent king, you will need to have some sophisticated code to recognize such occasions, consider the diagram.

The moves 1.e5 and 1.g6 are extended with one ply since it seriously increases the attack on the black king, the other moves are not extended as they do not increase the pressure on the black king, also checks (such as 1.Qh6+) are excluded as an exception condition.

The extension in practise isn't very powerful, say +5-10 elo, its costs is pretty cheap, a 5% slowdown of REBEL's search.

Some Final Remarks on Extensions:

- When a move is extended make sure the same move is not extended another time, it's good in general not to extend moves twice.

- It's good to have some kind of formula that will control the maximum_number_of_extensions, REBEL does this based on its iteration.

- REBEL uses the concept of fractional extensions which gives you as a programmer more freedom to define extensions more precise. The fractional extension technique as used in REBEL divides a ply into 4 parts. To extend the tree with a full ply the value "4" is added to a counter which is used later to calculate the real depth.

Having such as system you can do all sort of tricks, reward extensions on their importance by toying with values between 1-4, or even 6 (1½ ply) and so on.

This ends the description of REBEL's used extensions, let's move now to the next chapter, Selective Search, one of the most complicated parts to understand, especially when you are a null-mover and are not familiar with static evaluation pruning.

Selective Search Techniques: Introduction

REBEL since its early existence in 1980 has been a selective program, in those days you had 2 choices, either have a pure **brute force** program or enter the dangerous path of **selective search** by **static evaluation**, the latter was only done by a few, among them Richard Lang (Chess Genius) and myself which became the base of their success later.

In those days **Null-Move** (as we know it today) did not exist. I first heard of Null-Move in 1986 during the World Championship in Cologne, Germany. **Don Beal** was participating with his chess program that used a Null-Move technique in his Quiescent Search, the seed of a major breakthrough in computer chess was sowed.

During the tournament **Frans Morsch** (FRITZ) kept on talking about Null-Move to me, "Ed, there must be something real good in Null-Move, I am going to research this". I didn't pay attention and shrugged, Null-Move, no way.

But then in 1991/92 Frans Morsch implemented Null-Move is his Fritz in a **new way** and Null-Move became a big success as it was a very powerful and easy way of doing **selective search**, no more tricky static evaluation tricks, but the relative safe search based R=2 approach, easy, clean and powerful.

Then Frans leaked his Null-Move approach to **Chrilly Donninger** the author of NIMZO who wrote an article in the ICCA journal and Null-Move became public. Nowadays I can't mention a chess program that doesn't use Null-Move, the chess programmer community owes Frans Morsch a big thanks.

REBEL however kept loyal to its own system, that is, doing Selective Search by Static Evaluation and below you will find its main description. Later I added Null-Move to REBEL's Selective Search but it is used in a total different way namely: **to find the errors (exceptions) in the static evaluation concept**, more later.

Selective Search Techniques: Concepts

The first thing you will need to keep in mind is that REBEL's way of doing Selective Search **demands** doing a complete evaluation of each position in the Main-Search till horizon_depth-1.

You simply need to have decent information about the position before you can decide to prune a complete subtree, but let's start... **take a deep breath first**...

REBEL's search is split into 2 parts:

- The Null-Move Part (the first "x" plies of the iteration depth)
- The Static Evaluation Part (the remaining plies of the iteration depth)

For instance: we are at iteration 11, REBEL in the first 5 plies will practice (if needed) Null-Move, **for the remaining 6 plies REBEL will fully rely on his own Static Evaluation Concept, no Null-Move is used**.

The value of "x" is flexible, it depends on the stage of the game and is also iteration

based, we will call "x" **S_DEPTH** (Static Depth) from now on, its formula plus tables:

```
define stage of the game before calling Search:
middle game     : STAGE = 0
end game        : STAGE = 1
late end game   : STAGE = 2    // rook endings, B/N and pawn endings
```

S_DEPTH is defined after each iteration, table driven, its formula:

```
S_DEPTH = TABLE [STAGE] [iteration_depth];

char TABLE [0][] = { 0, 1, 1, 1, 2, 2, 3, 3, 3, 4, 4, 5, 5,  // middle game
                     6, 6, 7, 8, 9,10,11,12,13..... }

     TABLE [1][] = { 0, 1, 1, 1, 2, 2, 3, 3, 4, 5, 6, 7, 8,  // endgame
                     9,10,11,12,13,14,15,16,17..... }

     TABLE [2][] = { 0, 1, 1, 1, 2, 2, 3, 4, 5, 6, 7, 8, 9,  // late endgame
                    10,11,12,13,14,15,16,17,18..... };
```

Example: iteration = 11

```
S_DEPTH = middle game   -> 5 -> 5 plies Null-Move -> 6 plies Static Pruning.
S_DEPTH = end game      -> 7 -> 7 plies Null-Move -> 4 plies Static Pruning.
S_DEPTH = late end game -> 8 -> 8 plies Null-Move -> 3 plies Static Pruning.
```

REMARK: in reality S_DEPTH is more sophisticated, there are various between-forms, but that you can figure out yourself, this is only the main thought behind S_DEPTH.

Furthermore during Search **S_DEPTH** is incremental updated to be in sync with REBEL's extensions otherwise the Static Evaluation Part would become too long. In principle it comes down that every time REBEL does an extension it will increment S_DEPTH too.

REBEL's search in free style pseudo code looks as follows:

```
if (remaining_depth>1 && current_depth<=S_DEPTH) then do Null-Move Part
{
  Make_Move();                              // update board position
  Evaluate_Position();                      // do a complete evaluation
  if Selective_Search_PART_ONE_is_true(); -> go one ply deeper (no Null-Move)
  else: do Null-Move Search                 -> R=2 or R=3 (more later)
}

if (remaining_depth>1 && current_depth>S_DEPTH) then do Static Evaluation Part
{
  Make_Move();                              // update board position
  Evaluate_Position();                      // do a complete evaluation
  if Selective_Search_PART_TWO_is_true(); -> go one ply deeper
  else: prune (cut-off subtree)
}
```

Having explained the concept of S_DEPTH we now can move to the 2 parts of coding, the **Null-Move Part** and the **Static Evaluation Part**, let's do the Null-Move Part first.

Selective Search Techniques: Null-Move

We are in the very first plies of the tree defined by **S_DEPTH**, see above, the goal of this part is to decide if it is necessary to do an **expensive Null-Move-Search** or not, this based on **Static Evaluation**.

To measure the performance of this routine REBEL keeps track of the percentage of Null-Move re-searches. It's percentage usually fluctuates between 5-7% which is excellent, it means that 93-95% of the total of Null-Move searches return OKAY meaning that no research at full depth is needed.

Mind you, if you have SCORE + THREAT (coming from the evaluation) already at your disposal, then why do an expensive Null-Move-Search by default? There is no need, REBEL tries the below first, its pseudo code:

```
if (ALPHA < SCORE + THREAT) return TRUE; // search node with full depth.
```

The condition works very well to limit expensive Null-Move searches because when SCORE+THREAT is already greater than ALPHA, then in most of the cases (>95%) Null-Move will return FALSE (thus research) and so you will have to search the tree at full depth anyway.

This is the exact goal of the routine, find the **likely** situations where Null-Move will produce FALSE and save yourself a lot of unnecessary Null-Move searches, thus save processor time. We are going to the next conditions...

Other cases where Null-Move is avoided:

```
if (own_king_was_in_check_before_make_move)
    return TRUE;       // search node with full depth
if (move_does_check_the_opponent_king)
    return TRUE;       // search node with full depth
else: return FALSE;    // do Null-Move
```

REMARKS:

- Despite of the above advantages to avoid needless Null-Move Searches there is a disdvantance worth to mention, that is that move-ordering is slighty better in case you don't practice an Avoid_Null_Move routine, it can be different for each program.

- It's very easy to make REBEL a full Null-Move program, all it needs to do is to make **S_DEPTH** equal to **iteration_depth**, actually REBEL has an Interface driven parameter for that: [Selective Search = 0] in the Personality Editor.

- When REBEL uses Null-Move it uses R=3 for the middle-game and R=2 for the end-game.

Selective Search Techniques: Static

We are in the last plies of the tree defined by **S_DEPTH**, see above, the goal of this part is to decide **if a subtree is worth to search to the full depth or prune it completely**, this by **Static Evaluation**.

We are of course on very thin ice here, mind you if we are at iteration 11 where S_DEPTH=5 then REBEL from ply=6 and onwards will allow the complete pruning of a subtree (5 plies in this case!), very powerful of course if all works well, but extremely dangerous when the error-margin becomes too high.

Nevertheless REBEL since its early existence works that way and the system in practice works very well. Actually the below described **pruning system** gave REBEL quite a lead till 1995/96 before Null-Move made its entrance and became dominant the years after.

How it is done, some easy pseudo code to start with:

```
if (own_king_was_in_check_before_make_move)
    return TRUE;      // search node with full depth
if (move_does_check_the_opponent_king)
    return TRUE;      // search node with full depth
if (move_is_winning_capture)
    return TRUE;      // search node with full depth
if (ALPHA < SCORE + THREAT + MARGIN)
    return TRUE;      // search node with full depth

SCORE  : score of EVAL
THREAT : Queen=900, Rook=500, Bishop=300, Knight=300, Pawn=100
MARGIN : TABLE [remaining_depth];

static int TABLE[]= { 00,00,10,15,20,25,25,25,25,25,25,25,25,25,
                      25,25,25,25,25,25,25 ........... };
```

After these check have all failed, the subtree in principle is now candidate for (complete) pruning, however there are 2 exception cases where REBEL will not prune, these are:

- In case the move (from Make_Move) is a white pawn that moves to the 7th or 6th rank.
- In case the move (from Make_Move) is a black pawn that moves to the 2th or 3th rank.
- When the move (from Make_Move) seriously increases the pressure on the opponents king, more below.

You must have some kind of code that recognizes mate-threats, that meassures if a move makes progress attacking the opponents king in a dangerous way, you can't afford to

prune moves like that. The issue is already described elsewhere.

If these conditions aren't met also **the whole subtree is pruned**!

The complete pseudo code:

```
if (own_king_was_in_check_before_make_move)
    return TRUE;      // search node with full depth
if (move_does_check_the_opponent_king)
    return TRUE;      // search node with full depth
if (move_is_winning_capture)
    return TRUE;      // search node with full depth
if (ALPHA < SCORE + THREAT + MARGIN)
    return TRUE;      // search node with full depth
if (white_pawn_moves_to_rank6_or_7)
    return TRUE;      // search node with full depth
if (black_pawn_moves_to_rank3_or_2)
    return TRUE;      // search node with full depth
if (move_threatens_opponent_king)
    return TRUE;      // search node with full depth
else: return FALSE;   // prune complete subtree
```

In reality the code is more sophisticated handling some more (minor) issues, but the above listed is REBEL's framework for **static pruning subtrees** and will do good in practice.

Quiescent Search

Quiescent Search (abbreviated to **QS**) in REBEL has 2 goals, in a nutshell:

- Check if the evaluation of the horizon depth (SCORE) is safe from tactical surprises such as captures, promotions, checks. SCORE tends to go down, a correction takes place.
- Check for possible (long) series of checking-moves, there could be some material gain, or even a mate. SCORE tends to go up.

For REBEL the focus of QS is entirely on getting the right score for the evaluation of the horizon position, no further special tricks.

Unlike other chess programs REBEL (since the early 80's) in QS **does not** investigate **all captures**, there is absolutely no need for that, it's pretty safe to search only the **winning captures**, **equal captures** (QxQ, RxR etc) and **Queen Promotions**. Minor promotions are also excluded from QS, it's a waste of valuable processor time.

Excluded from the above are of course the situations when the king is in check, all moves are generated and searched.

For a given ply in QS REBEL will first generate and search the **winning captures**,

secondly when those moves do not cause a BETA cut-off then search the **equal captures** and third and last do the **checking moves** (this limited to a predefined depth, more later), the rest of the moves is skipped.

Queen Promotions are part of the winning-capture concept. Naturally when the position has a best move from the Hash Table that move is searched first.

Its pseudo code, note that apart from 3-stage order mechanism QS is much of the same as the Main Search.

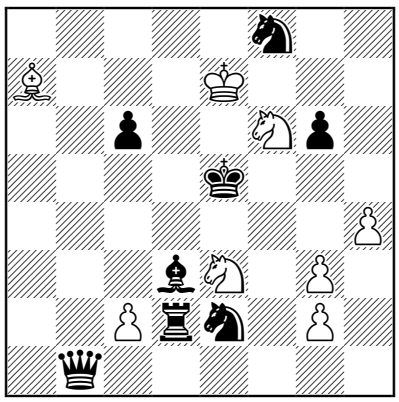
```
Get_a_Move_Until_No_More_Moves          -> following the move ordering
                                           as described above
if (Lazy_Eval_is_true)                  -> done, return score, see elsewhere
     Evaluate_Position();
if (Trick_one_is_true)                  -> done, return score, see elsewhere
if (move_does_check_the_opponent_king) -> go one ply deeper in QS
if (ALPHA >= SCORE)                     -> done, return score
if (Trick_two_is_true)                  -> done, return score, see elsewhere
else: go one ply deeper in QS
```

Long Checks : REBEL in QS will search all moves that check the opponent king, this till a given depth, called **MAX_CHECKS_DEPTH**. This variable is defined during each iteration as:

```
MAX_CHECKS_DEPTH = iteration_depth + 2
```

It means that QS by default is allowed to search all checking moves during the first 2 plies of QS. Futhermore MAX_CHECKS_DEPTH is increased during the Main Search and QS when the following is true:

```
if (king_is_in_check) then {
   if (only_one_legal_move)  MAX_CHECKS_DEPTH = MAX_CHECK_DEPTH + 2
   if (only_two_legal_moves) MAX_CHECKS_DEPTH = MAX_CHECK_DEPTH + 1
}
```



This mechanism will guarantee REBEL to find deep tactical shots such as deep mates, deep repetitions, deep material combinations when the position is dominant to checks, have a look at the diagram.

In this position REBEL is able to announce a **Mate in 30 Moves** at iteration 1 having searched only 2351 positions, all because of the above described update mechanism of MAX_CHECK_DEPTH.

I use this system since 1987/88, it was introduced in the Mephisto MMV, even on an

ancient 6502 processor running at only 5 Mhz the system worked well. Only in the latest version of REBEL I have changed the formula a bit, that is:

```
MAX_CHECKS_DEPTH = current_depth + 2 // at the very start of QS

if (king_is_in_check && in_QS) then {
   if (only_one_legal_move)  MAX_CHECKS_DEPTH = MAX_CHECK_DEPTH + 2
   if (only_two_legal_moves) MAX_CHECKS_DEPTH = MAX_CHECK_DEPTH + 1
}
```

It practical means that during the Main-Search MAX_CHECKS_DEPTH is no longer increased, the change gave a +14% speed-up.

REMARK: to use the REBEL concept of long-checks you will need to have some kind of code that is able to count the number of legal moves when the king is in check.

Evaluation: Introduction

REBEL has a large and very expensive evaluation function with hundreds evaluation characteristics, it is impossible to present them all, therefore only the most dominant evaluation stuff will be presented, also a bit about its data structure and programming techniques, let's start with the latter.

Piece-Type, REBEL has a most simple data structure for the 12 piece types as used on its internal chess board:

```
+----+----+----+----+----+----+----+----+----+----+----+----+----+  00 = empty square
| 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 |  01 = white pawn
+----+----+----+----+----+----+----+----+----+----+----+----+----|  02 = white knight
| .. | Wp | WN | WB | WR | WQ | WK | Bp | BN | BB | BR | BQ | BK |  .. = ............
+----+----+----+----+----+----+----+----+----+----+----+----+----+  12 = black king
```

The reason for this simple approach is two-sided:

- Easy access to tables for indexing, keep tables small and surveyable.
- Make use of the processor so called "indirect addressing" possibilities, for instance in the the case of the use of the popular C statement **switch-case** and/or use "indirect addressing" for calling routines.

A few examples to explain:
Example-1: Any good C compiler will produce perfect code if you use **switch-case** using an unbroken and continuous string of characters, consider the following code while (for instance) generating moves or scanning the internal chess board:

```
switch (piece_type) {
    case  0 : goto empty;         // empty square, get next square
    case  1 : goto white_pawn;    // evaluate white pawn
    case  2 : goto white_knight;  // evaluate white knight
```

```
            case  3 : goto white_bishop;
            case  4 : goto white_rook;
            case  5 : goto white_queen;
            case  6 : goto white_king;
            case  7 : goto black_pawn;    // evaluate black pawn
            case  8 : goto black_knight;
            case  9 : goto black_bishop;
            case 10 : goto black_rook;
            case 11 : goto black_queen;
            case 12 : goto black_king;
        }
```

Any good C compiler will translate the above C code into **one assembler instruction**, something like:

```
        jmp    TABLE [EAX]
```

Example-2: You can even do the same in just **one C instruction** for calling routines instead of using the **goto** instruction, here is how:

```
 (TABLE[piece_type-1])();         // call routine as defined in TABLE

  void (*TABLE[])() = { w_pawn,w_knignt,w_bishop,w_rook,w_queen,w_king,
                        b_pawn,b_knight,b_bishop,b_rook,b_queen,b_king };

  void w_pawn()   { code here }
  void w_knight() { code here }
  .... .........  ............
  void b_king()   { code here }
```

Indexing: REBEL in EVAL wherever it makes sense will use **square tables** above **fixed values**. For instance, when evaluating an isolated pawn it can be given a fixed value as penalty, such as 0.10, however a square table is more precise, more flexible to tune too.

```
  +----+----+----+----+----+----+---+----+
8 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
  +----+----+----+----+----+----+----+----+
7 | 10 | 12 | 16 | 20 | 20 | 16 | 12 | 10 |
  +----+----+----+----+----+----+----+----+
6 | 10 | 12 | 16 | 20 | 20 | 16 | 12 | 10 |
  +----+----+----+----+----+----+----+----+
5 | 10 | 12 | 16 | 20 | 20 | 16 | 12 | 10 |
  +----+----+----+----+----+----+----+----+
4 | 06 | 08 | 10 | 16 | 16 | 10 | 08 | 06 |
  +----+----+----+----+----+----+----+----+
3 | 04 | 06 | 08 | 10 | 10 | 08 | 06 | 04 |
  +----+----+----+----+----+----+----+----+
2 | 02 | 04 | 04 | 10 | 10 | 04 | 04 | 02 |
  +----+----+----+----+----+----+----+----+
1 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
  +----+----+----+----+----+----+----+----+
    a    b    c    d    e    f    g    h
```

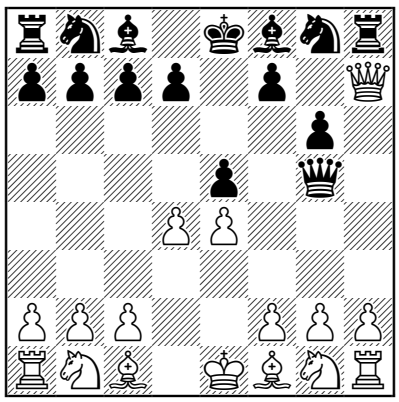
Penalties for white isolated pawns based on their position on the board.

Background: usually in the middle game an isolated pawn on A2 is not so bad as an isolated pawn in the center.

Advice: use such square tables wherever you can in EVAL.

Remark: this example is only valid for REBEL's middle game, use different values for the end game.

Evaluation: Hanging Pieces



REBEL in EVAL (for the side to move) will detect the so called **hanging pieces** with a maximum of 3 squares, the 3 squares are sorted on the expected material loss.

The same applies for the opponent side, it's only called different, **threatened pieces**, consider the left diagram.

Hanging Pieces are used elsewhere in other parts of the program (move ordering, Q-Search etc.), the same applies for **Threatened Pieces**, it is mainly used as THREAT value, see Selective Search, Reductions etc.

With "black-to-move-next" REBEL's list will look as follows:

```
Hanging Pieces:    Qh7 (value 9.00), Bc1 (value 3.00), pawn d4 (1.00)
Threatened Pieces: Qg5 (value 9.00), Rh8 (value 5.00)
```

When "white-to-move-next" the list is swapped:

```
Hanging Pieces:    Qg5 (value 9.00), Rh8 (value 5.00)
Threatened Pieces: Qh7 (value 9.00), Bc1 (value 3.00), pawn d4 (1.00)
```

How it is done.

REBEL uses 2 **board tables**, one for white (WB), one for black (BB) which are zeroed at the beginning of EVAL. While scanning the board REBEL will update WB and BB, let's take the **white knight on G1** from the diagram as an example: On G1 the knight can move to the squares: E2, F3 and H3, those squares are updated as follows:

```
WB[F3]=++WB[F3] | 16;    // square+1 , set bit 4
WB[H3]=++WB[H3] | 16;    // square+1 , set bit 4
WB[E2]=++WB[E2] | 16;    // square+1 , set bit 4
```

This process is done for all the white pieces, each square that is controlled by a white piece is incremented with 1 and a corresponding bit is set (using the OR operator), its data structure:

```
+------+------+------+------+------+------+------+------+
| BIT0 | BIT1 | BIT2 | BIT3 | BIT4 | BIT5 | BIT6 | BIT7 |
+------+------+------+------+------+------+------+------+
|      Number of     | PAWN |KNIGHT| ROOK | QUEEN| KING |
|      ATTACKERS     |      |BISHOP|      |      |      |
+------+------+------+------+------+------+------+------+
```

After all white pieces are done square F3 from WB looks as follows:

```
+------+------+------+------+------+------+------+------+ B0-B2: 2 attackers
| BIT0 | BIT1 | BIT2 | BIT3 | BIT4 | BIT5 | BIT6 | BIT7 |
+------+------+------+------+------+------+------+------+ B3   : white pawn
|      Number of     | PAWN |KNIGHT| ROOK | QUEEN| KING |
|      ATTACKERS     |      |BISHOP|      |      |      | B4   : white knight/bishop
+------+------+------+------+------+------+------+------+
|  0   |   1  |  0   |  1   |  1   |  0   |  0   |  0   |
+------+------+------+------+------+------+------+------+
```

The same is done for all the black pieces updating the table for black (BB). It's very powerful to have such data, for each sqaure on the board (empty or occupied) REBEL has the attackers and defenders, many interesting evaluation tricks can be tried for instance by using the value (bit-setting) of a square as an index to a 256 byte evaluation table, just think a bit of all the possibilities.

The data is used for evaluatig mobility, king safety, pawn structure, passed pawns, center control, outposts and more. Also it is used to generate the hanging pieces of the position which is the current topic. Having the data of **WB** and **BB** REBEL has enough information to decide if a piece is hanging, its pseudo code:

```
get_next_piece_on_the_board                          // scan the board
status = TABLE[piece_type][WB[square]][BB[square]];  // get status via the bits
if (status == 0) continue;                           // piece safe -> next square
else: piece hangs, status contains its value         // Q=9 R=5 B=3 N=3 P=1

char TABLE [12] [256] [256];                         // about 860 Kb
```

During program startup the 3-demensional TABLE is filled from hard disk with the predefined values of all combinations of possible bit settings for white and black by piece_type. The formula to create the contents of TABLE will be given later, but maybe it's more fun to figure it out yourself.

HINT: **WB** and **BB** are zeroed at the beginning of EVAL, this is a costly operation in C, here is a trick to speed it up using **redefinition**:

```
unsigned char WB[64],BB[64];
long *PWB = (long *) WB;             // redefine char (8-bit) to long (32-bit)
long *PBB = (long *) BB;

PWB[0]=PWB[1]=PWB[2] ... =PWB[15]=0; // 16 x 32-bit stores, clear WB
PBB[0]=PBB[1]=PBB[2] ... =PBB[15]=0; // 16 x 32-bit stores, clear BB
```

This is about 8-10 times faster then the usual:

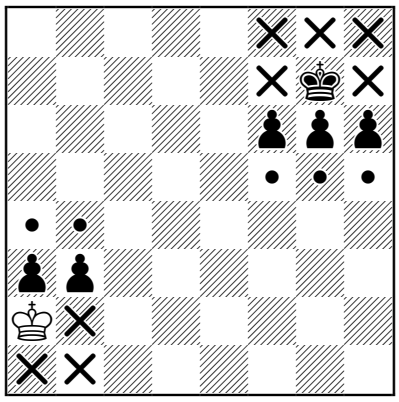
```
for (x=0; x<=63; x++) { WB[x]=0; BB[x]=0; }
```

Make sure that your compiler's **alignment** at least is set to 32 bit so that the generated memory addresses of WB and BB are divisible by 4. In most compilers the default setting is 32 or 64 bit which is okay.

King Safety

REBEL has a large piece of code to evaluate the pressure on its own king and on the opponent king, both routines are each others mirror, REBEL doesn't practice the so called asymmetric approach.

It would go to far to list all REBEL's stuff regarding king-safety, presented below is its main frame. Excluded are issues like the "pawn shield", "pawn rams", "castling", handling "opposite kings".



Consider the diagram, the squares around the king will be evaluated using the data that is gathered in WB and BB, see elsewhere.

The squares around the king marked with × are measured different than those marked with ♝, same story for the squares marked with •, each type of squares has its own dynamics.

[Note: the ♝ symbol in this diagram refers to a kind of square, not to a pawn].

But before going into detail it is important to understand the following principle:

REBEL uses **progressive evaluation** for king safety, more REBEL will use progressive evaluation wherever it makes sense in EVAL, it's a quite different technique than **normal evaluation**. To clarify the terms:

- **Normal Evaluation**: evaluate square F8 regarding king safety, add the value to the collective evaluation variable (SCORE), evaluate the next square F7, F6, F5, G8 .... until all squares are evaluated.

  Disadvantage : it won't evaluate the coherence between the pieces that attack the king, here progressive evaluation comes in.

- **Progressive Evaluation**: the final evaluation is delayed till all squares are measured, instead of that a **counter** is initialized (COUNTER=0) and updated while evaluating all the squares, then this counter is used as an index to the final evaluation table for king saftey (TABLE) and TABLE[COUNTER] will added to SCORE.

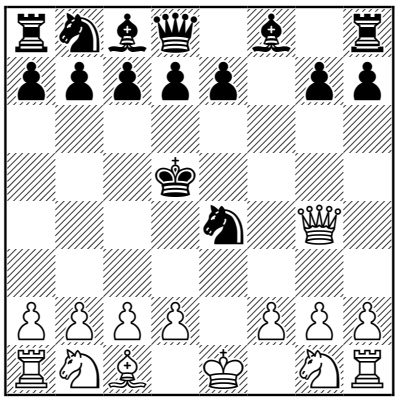
  Advantage: depending on the quality of the contents of TABLE and COUNTER it is possible to measure the coherence between the pieces that attack the king, that when for instance there is only a Queen that attacks the king it isn't rewarded too high, but

when a rook and a bishop participate in the attack the bonus jumps over a pawn, or more.

Let's have a look at REBEL's (progressive) king-safety evaluation table:

```
int TABLE [] = {  0,   2,   3,   6,  12,  18,  25,  37,  50,  75,
                100, 125, 150, 175, 200, 225, 250, 275, 300, 325,
                350, 375, 400, 425, 450, 475, 500, 525, 550, 575
                600, 600, 600, 600, 600  ......... };
```

The bonus for king safety is small with little pressure on the king (COUNTER is low), however the bonus goes up progressively when COUNTER increases, it goes up with 1/4 of a pawn each step.



Such a concept makes it easy to tune too, one only has to tune the values of TABLE. We are now ready to calculate the value of COUNTER.

Consider the diagram on the right, the (well known) position occurs after the moves: 1.e4 Nf6 2.Bc4 Nxe4?! 3.Bxf7+! Kxf7 4.Qh5+ Ke6? 5.Qg4+ Kd5?? Black wants to defend Nd5 by all means and will lose, black should have played 4..Kg8.

To avoid REBEL making such giant mistakes COUNTER is pre-filled with a value of a square-table, note: we are evaluating the black king.

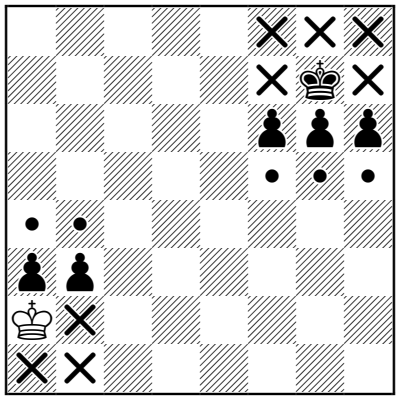
```
 +----+----+----+----+----+----+---+----+
8| 02 | 00 | 00 | 00 | 00 | 00 | 00 | 02 |
 +----+----+----+----+----+----+----+----+
7| 02 | 01 | 01 | 01 | 01 | 01 | 01 | 02 |
 +----+----+----+----+----+----+----+----+
6| 04 | 03 | 03 | 03 | 03 | 03 | 03 | 04 |
 +----+----+----+----+----+----+----+----+
5| 04 | 04 | 04 | 04 | 04 | 04 | 04 | 04 |
 +----+----+----+----+----+----+----+----+
4| 04 | 04 | 04 | 04 | 04 | 04 | 04 | 04 |
 +----+----+----+----+----+----+----+----+
3| 04 | 04 | 04 | 04 | 04 | 04 | 04 | 04 |
 +----+----+----+----+----+----+----+----+
2| 04 | 04 | 04 | 04 | 04 | 04 | 04 | 04 |
 +----+----+----+----+----+----+----+----+
1| 04 | 04 | 04 | 04 | 04 | 04 | 04 | 04 |
 +----+----+----+----+----+----+----+----+
    a    b    c    d    e    f    g    h
```

Initialize COUNTER:

```
COUNTER = TABLE [black_king];
```

This will avoid the black king enter the center, as soon as white has a little pressure the squares surrounding Kd5 the score will go over 1 or 2 pawns with ease.

Also, squares like H7, H8 are set a bit higher because there are less surrounding squares on the black king, this will avoid moves like Kg8-h8 whenthere is absolutely no need for that.



Based on the left diagram the black king being on G7 the value of COUNTER=1.

The **second step** is to measure the pressure on the • squares (F5, G5 and H5).

What is measured on those squares ONLY is the **attack pattern** using the bits (3-7) of WB[F5], WB [G5] and WB[H5] via the OR operator, it will represent the pressure it has on those squares.

The code is powerful in the middle game helping REBEL to set up a king attack, when for instance black's king is well guarded (BKg8, BRf8, BPf7,g7,h6), a white rook on H1 will measure the pressure it has on the H6 pawn and if the white Queen is part of the attack too COUNTER will be added with 2, more in the last step, its pseudo code.

```
char flag=0;

flag = flag | WB[F5];      // update attack pattern
flag = flag | WB[G5];      // update attack pattern
flag = flag | WB[H5];      // update attack pattern
```

The **third step**, measure the × squares (F7, F8, G8, H8, H7), update COUNTER and the attack pattern (flag), its formula and pseudo code:

```
if (WB[F7] != 0x00) {
   COUNTER++;                       // white has pressure on F7
   flag = flag | WB[F7];            // update attack pattern
   if (BB[F7] == 0x81) COUNTER++; // square F7 not protected by
                                    // any of the black pieces
}
```

Do the same for the squares F8, G8, H8 and H7.

The **fourth step**, measure the ♝ squares (F6, G6 and H6) update COUNTER and the attack pattern (flag), its code is only a bit different than step-3, it checks for the presence of an own piece too defending its king, the formula and pseudo code:

```
if (WB[F6] != 0x00) {
   COUNTER++;                                  // white has pressure on F6
   flag = flag | WB[F6];                       // update attack pattern
   if (BOARD[F6] != own_piece) COUNTER++;
   if (BB[F6] == 0x81) COUNTER++;              // square F6 not protected by
                                               // any of the black pieces
}
```

Do the same for the squares G6 and H6.

We are now ready for the **last step**, calculate the value for the **attack pattern**, add it to COUNTER, then calculate the final value for the black king regarding king safety, the pseudo code:

```
COUNTER = COUNTER + TABLE [flag << 3]; // reward the attack pattern with
                                       // 1, 2 or 3 depending on the bits
                                       // that were set.
SCORE = SCORE + EVAL [COUNTER];        // final evaluation.

char TABLE [] = {
//      . P N N R R R R Q Q Q Q Q Q Q Q K K K K K K K K K K K K K K K K
//            P   P N N   P N N R R R R   P N N R R R R Q Q Q Q Q Q Q Q
//                  P       P   N N N       P   P N N   P N N R R R R
        0,0,0,0,0,0,1,1,0,1,2,2,2,3,3,3,0,0,0,0,1,1,2,2,2,3,3,3,3,3,3,3 };

int EVAL [] = {   0,  2,  3,  6, 12, 18, 25, 37, 50, 75,
                100,125,150,175,200,225,250,275,300,325,
                350,375,400,425,450,475,500,525,550,575
                600,600,600,600,600 ......... };
```

REMARKS :

- King Safety in REBEL is not done when the opponent has no Queen, REBEL on such occasions fully relies on its Search, in practice it doesn't have any negative side effects. Actually I noticed that doing King Safety in the end-game seriously hurts REBEL's end-game knowledge. The issue is debatable of course.

- So when white has a Queen and black has no Queen, king safety for the black king is done, for the white king it is not, and vice versa.

- Furthermore when there is only a lone Queen left (no light pieces anymore) to attack the opponent king, king safety is skipped too. Debatable of course too, in parctice it works best of REBEL.

- Last, when there are too few light pieces left to attack the opponent king (Rooks, Bishops, Knights) COUNTER isn't pre-set to the king position on the board, as explained above, instead COUNTER is zeroed, in principle it means the king is free to walk across the board without getting a huge penalty for that. Its pseudo code:

```
COUNTER = TABLE [black_king];              // see above
if (white_has_not_at_least_2_light_pieces)
    COUNTER=0;
```

- Again, it would go to far to list all REBEL's stuff regarding king-safety, this would make this already too long page even more longer. Excluded are issues like the "pawn shield", "pawn rams", "castling", handling "opposite kings".