1 A comparison of Rybka 1.0 Beta and Fruit 2.1

This document is designed as a reference guide to various technical elements in the discussions with Rybka 1.0 Beta and Fruit 2.1. In some sense, it is a prequel to the Rybka/IPPOLIT analysis. However, the issues considered are not really the same. The claims made about Rybka/IPPOLIT were of a sufficiently different nature to those about Rybka/Fruit that a different type of discussion seems necessary. [This document is the version of February 12, 2011].

1.1 Evidence, and standards therein

This document shall outline the evidence regarding Rybka 1.0 Beta and Fruit 2.1, and try (at times) to put it into context. In particular, one must make a choice of standard of comparison. Many have been suggested, such as "code" copying, or "copyright" considerations. It is my opinion, however, that the proper standard to use is that which is commonly used in the context of computer chess (or more generally, computer boardgames). This has, at least historically, been construed to mean that no code which has a demonstrative influence on the performance of the program in question may be borrowed from a competitor.[1] In some sense then, this report is directed at a hypothetical tournament arbitrator who has been asked to determine whether Rybka 1.0 Beta is sufficiently original to allow it to be in the same event as Fruit 2.1.[2] The previous decisions in this genre include that of the case between Berliner and Hsu, where the latter agreed to remove/rewrite approximately 0.3% of his codebase (some sort of simulation of the Cray Blitz evaluation function) due to the fact that it had been taken from the HITECH project at an earlier time.

Furthermore, this document is fundamentally incapable of anticipating even legitimate explanations of the evidence here enumerated, and as such, is more of a call to further conversation than the final word.

2 Outline of the evidence

There are various major points of evidence between Fruit 2.1 and Rybka 1.0 Beta, and a number of minor and/or more circumstantial ones. The major points of evidence include:

- the use of *exactly* the same evaluation features;
- the identical ordering of operations at the root node in the search;
- the same type of PST-scheme, re-using the identical File/Rank/Line weights.

[1] There do exist counterexamples, such as re-use of Nalimov tablebases and probing code. Another is Crafty (and Rybka 4) using Pradu Kannan's magic multiplication code from Buzz.

[2] Just to recapitulate, this means that I am largely going to avoid GPL or copyright infringment issues, and I also disregard any statement from a representative of either Fruit/Rybka concerning their opinion of the matter (after all, from the standpoint of a tournament director, two "competitors" could be colluding so as to gain multiple entries to an event — this was actually an issue 20+ years ago, but hopefully those days are past us!).

The lesser pieces of evidences include:

- the re-appearance of some similarities in data structures in hashing;
- the re-appearance of 10-30-60-100 scaling in some evaluation numerology;
- some commonality of UCI parsing code, including a spurious "0.0" float-based comparison in the integer-based time management code of Rybka.

Some of these can be considered "ideas" rather "code" for various purposes; while the standard I adopt here makes some *nuance* between the two, it is not so strict so as to demand any re-appearance of any specific code or numerology.

It must be said that it is not entirely clear what is "fair game" to re-use from an open source program. For instance, pursuant to the first major point of above, one of the more notable "ideas" of Fruit was its evaluation based mainly on mobility. Below I compare the components of this evaluation routine to those used in Rybka 1.0 Beta. While a large match is found, it is certainly possible to argue that the Fruit source code can be taken as a "manual" for chess programming (perhaps in the sense of a modern version of "How Computers Play Chess"), and if this paradigmatic view is taken, then the re-use of the same evaluation components should not be seen as too derelict.

3 Commonality of evaluation features

Most of the work here was first done by Zach Wegner. I have verified much of it (see annotated ASM dump here). The crux of the conclusion is that Rybka 1.0 Beta and Fruit 2.1 have *exactly* the same evaluation features. I will simply enumerate these here, and in almost all cases the functionality is the same.

3.1 Piece evaluation (omitting king safety for now)

3.1.1 Bishops

Both Rybka and Fruit consider only mobility as the primary evaluation component with bishops.[3] Both Rybka and Fruit have a trapped bishop penalty; the same definition of "trapped" is used in each (translated to bitboards with Rybka of course), though the penalty is halved in Fruit in one case. Both Rybka and Fruit have a blocked bishop penalty for (say) a bishop on `c1`, a friendly pawn on `d2`, and an enemy piece on `d3`. Both Rybka and Fruit halve the overall evaluation in an opposite-colour bishop endgame when the number of pawns for each side differ by no more than 2 (both have a flag for such a bishop endgame in a material table, and then separately check if the bishops are oppositely coloured).

3.1.2 Knights

Both Rybka and Fruit have mobility as the only knight evaluation component.

[3]Both also use the same "vanilla" notion of mobility throughout, as opposed to variants such as "safe-square" or "forward" mobility. For instance, Larry Kaufman used these more complicated mobility measures when he wrote the evaluation function for Rybka 3.

3.1.3 Rooks

Both Rybka and Fruit consider mobility, open files, semi-open files, whether the opposing king is on a semi-open file, and a 7th rank bonus. With the 7th rank bonus, both Rybka and Fruit require the opponent to have: either at least one pawn on the seventh rank, or the king on the eighth rank. Both Rybka and Fruit have a penalty for a blocked rook, and as with bishops, the definition is the same.

Overall, the only real difference for rooks is in the computation of an "open file" – when a rook is in front of a friendly pawn (wRa3/wPa2 for instance), Fruit does not consider this to be "open", but Rybka does.

3.1.4 Queens

Both Rybka and Fruit consider mobility for queens, and a 7th rank bonus. With the 7th rank bonus, both Rybka and Fruit again require the opponent to have either: at least one pawn on the seventh rank, or the king on the eighth rank.

3.2 King Safety

Both Rybka and Fruit compute king safety by computing whether a piece (ignoring kings) attacks a square adjacent to the opponent's king. For each such piece, a counter is incremented, and a score is added based on what the piece is. For instance, in Fruit the `KingAttackUnit` is 0 for a pawn, 1 for minors, 2 for a rook, and 4 for a queen. In Rybka these are 0, 941, 418, 666, and 532, and Rybka also only computes "yes/no" as to whether any pawns attack a square around the enemy king, while Fruit counts the number of such pawns. Both only penalise for king danger when the opponent has at least two pieces, one being a queen.

3.2.1 King Shelter/Storm

Both Rybka and Fruit compute pawn shelter/storm for a king based upon three adjacent files (I don't think this idea is that new in Fruit, so I will not discuss the details much). Rybka uses a look-up table of patterns, while Fruit does bitscanning. Fruit has a consideration with castling potential, while Rybka does not. There are likely as many differences as similarities here.

3.3 Pawn Evaluation

Both Rybka and Fruit consider doubled pawns, isolated pawns (on open/closed files), backward pawns (again on open/closed files), detection of passed pawns, and candidate passed pawns. This is all fairly standard, though I don't think this exact choice had appeared before Fruit. The definition of "backward" is slightly different in Rybka, as is a slight *nuance* with candidate pawns. The similarity of relative numerology in candidate/passed pawns is discussed below.

3.3.1 Passed Pawns

Both Rybka and Fruit start with a raw bonus for a passed pawn, depending on the game phase and the rank. There are then various bonuses for a passed pawn being dangerous. When the opponent has no pieces, an unstoppable passer is highly rewarded. When there are pieces, Fruit gives a bonus if all the following are true: the opponent does not (currently) have any pieces blocking it; we are not blocking it; and the pawn can advance safely (via SEE). Rybka splits up these bonuses in a piecemeal fashion, and considers not only the square directly in front of the pawn, but all those until the promotion square (and "attacks" bitboards are used instead of SEE).

Both Rybka and Fruit give a bonus depending upon the distances of both kings to the square in front of the pawn. The bonus in Fruit is solely based on the distance, while the rank of the pawn is additionally included in Rybka.

The similarity of relative numerology for passed pawns is discussed more below; the relative scaling for each is essentially 10-30-60-100, though in units of 256. I would say this is the deepest piece of evidence with passed pawns, as other components either have slight variations in Rybka or are not specific to Fruit.

3.4 Interpolation

Both Rybka and Fruit interpolate "opening" and "endgame" values to get a final evaluation. Fruit's is linear, while Rybka's is a bit more complicated. Again this component could be considered a reasonable "idea" to take from Fruit in any event, but it is just one piece of evidence among many.

4 Identical procedures in root search

The underlying factualities here are taken from Zach Wegner's analysis that is given at `http://talkchess.com/forum/viewtopic.php?t=23118`. Note that the Fruit 2.1 code is spread across a number of function calls; it is unclear from a disassembly whether this is the case in Rybka 1.0 Beta.

Table 1: Root search operations in Fruit and Rybka

Fruit 2.1	**Rybka 1.0 Beta**
generate legal moves	generate legal moves
limit depth to 4 if #moves is 1	limit depth to 4 if #moves is 1
setup setjmp	setup setjmp
list/board copy	
reset/start timer	start timer
increment date and date/depth table	increment date and date/depth table
reset killers then history (sort_init)	reset killers then history
copy some Code()/UCI params	
score/sort root list	score/sort root list

Here are the comparative operations/ordering in Phalanx XXII (for example): generate legal moves, init killers/history, increment Age, setup time limits, sort root moves, start timer, return move if forced (there are various bits about book/learning that I omit). Except for a few obvious constraints (move generation must precede scoring/sorting them), much of the above can be re-ordered.[4]

I have verified the above orderings myself, and in an appendix below I analyse the final "iterative deepening" part of this function. This Fruit/Rybka overlap would already likely meet a "plagiarism" standard, for instance as used in the detection of non-original work in academia and/or book publishing (note that plagiarism is generally an *ethical* standard and not a legal one). There is also the question of how important this item is from a chess-playing standpoint, perhaps again viewing Fruit as a "manual" in some sense.

5 Common structure of PST computations

Another major point is the Piece-Square-Table (PST) computations, also known as "static" values. This occurs directly in the Fruit 2.1 code, while in Rybka we only see the end result.[5] Furthermore, there is a rescaling (centipawns versus 3399th pawns), and Rybka also uses different weightings for some parameters.

However, the use of various specific arrays is apparent in both Fruit 2.1 and Rybka 1.0 Beta. It is not immediately obvious how to judge their re-occurence; for instance, the use of $[-2, -1, 0, +1, +1, 0, -1, -2]$ for a file weighting can hardly be considered abnormal. The impetus of the evidence is that: the identical arrays are used by both Fruit 2.1 and Rybka 1.0 Beta for *each* piece, giving a total of 8 or so matching arrays (some of the arrays are themselves re-used, but the occurrence of each with a specific piece is an exact match in Fruit 2.1 and Rybka 1.0 Beta). There are minor differences, such as bonuses for central pawns, and that the `KingRank` array is unimportant in Rybka 1.0 Beta (due to the weighting for it being 0).[6]

5.1 The example of the knights

I give one example in fuller detail. For various reasons, the (white) knights seem to be the best choice, as there are two components (file and rank), and there are is only one minor variation (the `a8/h8` square). Below are the raw PST knight values for Fruit 2.1 and Rybka 1.0 Beta in the opening, the latter on the right. The co-incidence of these can be seen when we make a formulaic representation (omitting the left-right symmetry).

[4] The "scoring" phases also contain the common element of a hash lookup to find a best move, and it seems that not many other engines do this before starting the iterative deepening.

[5] I might stress that the fact that Fruit 2.1 visibly computes these while Rybka 1.0 Beta just has an array is not really relevant for the discussion here. The content is of more import.

[6] Similarly one could note that Rybka 1.0 Beta has a score for pawns in the endgame and queens in the opening, while in Fruit these are all just zero (the second is explicitly 0 in the source code, while the first is not). The existence of these "zero weights" makes drawing schematic diagrams a bit tricky, and prone to possible reliance on non-existent similarities.

Table 2: Fruit and Rybka White Knight Opening PST values

-135	-25	-15	-10	-10	-15	-25	-135
-20	-10	0	5	5	0	-10	-20
-5	5	15	20	20	15	5	-5
-5	5	15	20	20	15	5	-5
-10	0	10	15	15	10	0	-10
-20	-10	0	5	5	0	-10	-20
-35	-25	-15	-5	-5	-15	-25	-35
-50	-40	-30	-25	-25	-30	-40	-50

-5618	-1724	-1030	-683	-683	-1030	-1724	-5618
-1366	-672	22	369	369	22	-672	-1366
-314	380	1074	1421	1421	1074	380	-314
-325	369	1063	1410	1410	1063	369	-325
-683	11	705	1052	1052	705	11	-683
-1388	-694	0	347	347	0	-694	-1388
-2440	-1746	-1052	-705	-705	-1052	-1746	-2440
-3492	-2798	-2104	-1757	-1757	-2104	-2798	-3492

Table 3: Common PST schematic for white knights in the opening

$-4x-4x+y-z$	$-4x-2x+y$	$-4x+0+y$	$-4x+x+y$	$\cdots$
$-2x-4x+2y$	$-2x-2x+2y$	$-2x+0+2y$	$-2x+x+2y$	$\cdots$
$0-4x+3y$	$0-2x+3y$	$0+0+3y$	$0+x+3y$	$\cdots$
$x-4x+2y$	$x-2x+2y$	$x+0+2y$	$x+x+2y$	$\cdots$
$x-4x+y$	$x-2x+y$	$x+0+y$	$x+x+y$	$\cdots$
$0-4x+0$	$0-2x+0$	$0+0+0$	$0+x+0$	$\cdots$
$-2x-4x-y$	$-2x-2x-y$	$-2x+0-y$	$-2x+x-y$	$\cdots$
$-4x-4x-2y$	$-4x-2x-2y$	$-4x+0-2y$	$-4x+x-2y$	$\cdots$

By using $(x, y, z) = (5, 5, 100)$ we obtain the values for Fruit 2.1, and with $(x, y, z) = (347, 358, 3200)$, we obtain those for Rybka 1.0 Beta. All the numbers (as opposed to letters) in the schematic appear in the Fruit 2.1 source code.

```
static const int KnightLine[8] = { -4, -2, +0, +1, +1, +0, -2, -4 };
static const int KnightRank[8] = { -2, -1, +0, +1, +2, +3, +2, +1 };
```

Other than the left-right symmetry in the Line array, there is no particular reason for these numbers to be used. For instance, we could write α_f and β_r for the file and rank numbers (where f ranges over files and r over ranks), and the above array then looks like:

Table 4: PST schematic with Line/Rank arrays as parameters

$\alpha_{18}x+\alpha_{ah}x+\beta_8y-z$	$\alpha_{18}x+\alpha_{bg}x+\beta_8y$	$\alpha_{18}x+\alpha_{cf}x+\beta_8y$	$\alpha_{18}x+\alpha_{de}x+\beta_8y$	$\cdots$
$\alpha_{27}x+\alpha_{ah}x+\beta_7y$	$\alpha_{27}x+\alpha_{bg}x+\beta_7y$	$\alpha_{27}x+\alpha_{cf}x+\beta_7y$	$\alpha_{27}x+\alpha_{de}x+\beta_7y$	$\cdots$
$\alpha_{36}x+\alpha_{ah}x+\beta_6y$	$\alpha_{36}x+\alpha_{bg}x+\beta_6y$	$\alpha_{36}x+\alpha_{cf}x+\beta_6y$	$\alpha_{36}x+\alpha_{de}x+\beta_6y$	$\cdots$
$\alpha_{45}x+\alpha_{ah}x+\beta_5y$	$\alpha_{45}x+\alpha_{bg}x+\beta_5y$	$\alpha_{45}x+\alpha_{cf}x+\beta_5y$	$\alpha_{45}x+\alpha_{de}x+\beta_5y$	$\cdots$
$\alpha_{45}x+\alpha_{ah}x+\beta_4y$	$\alpha_{45}x+\alpha_{bg}x+\beta_4y$	$\alpha_{45}x+\alpha_{cf}x+\beta_4y$	$\alpha_{45}x+\alpha_{de}x+\beta_4y$	$\cdots$
$\alpha_{36}x+\alpha_{ah}x+\beta_3y$	$\alpha_{36}x+\alpha_{bg}x+\beta_3y$	$\alpha_{36}x+\alpha_{cf}x+\beta_3y$	$\alpha_{36}x+\alpha_{de}x+\beta_3y$	$\cdots$
$\alpha_{27}x+\alpha_{ah}x+\beta_2y$	$\alpha_{27}x+\alpha_{bg}x+\beta_2y$	$\alpha_{27}x+\alpha_{cf}x+\beta_2y$	$\alpha_{27}x+\alpha_{de}x+\beta_2y$	$\cdots$
$\alpha_{18}x+\alpha_{ah}x+\beta_1y$	$\alpha_{18}x+\alpha_{bg}x+\beta_1y$	$\alpha_{18}x+\alpha_{cf}x+\beta_1y$	$\alpha_{18}x+\alpha_{de}x+\beta_1y$	$\cdots$

Here we should have $\alpha_{ah} = \alpha_{18}$, etc., but I rewrote the subscripts to indicate which was a rank element, and which was a file element. Admittedly, this formulation tends to stress the identical nature of the α and β choices made by Fruit 2.1 and Rybka 1.0 Beta, but indeed, that is the whole point.

5.1.1 Magnitude of this evidence

The magnitude of this evidence can be weighed is various ways. It must be first be noted that, while the use of these two File/Line arrays is not too strange, the identical arrays appear for every piece, and so mere coincidence is unlikely. A second question is whether the arrays really matter, when the x and y values could be said to have as much influence on the PST values.[7] My answer to that would be that there is no reason to keep the Rank/Line scaling, and in a fully independent implementation of the Fruit "idea" of PST, I would definitely expect them to differ at some points.[8] Finally, there is the issue of whether these arrays could re-appear for "harmless" reasons, but I really can't say much more than I have already.

5.2 Diagrams for other pieces

For reasons of completeness, I give the schematic PST pictures for the other cases, noting the values chosen by Rybka 1.0 Beta and Fruit 2.1. For all of these, the array choice is the same; the exception is the `KingRank` array in Rybka, as the value is chosen as zero, so the contents of the array are meaningless.

5.2.1 Pawns PST

Table 5: Common PST schematic for white pawns

$-3x$	$-x$	0	x	$\cdots$
$-3x$	$-x$	0	x	$\cdots$
$-3x$	$-x$	0	x	$\cdots$
$-3x$	$-x$	0	$x^\star$	$\cdots$
$-3x$	$-x$	0	$x^\star$	$\cdots$
$-3x$	$-x$	0	$x^\star$	$\cdots$
$-3x$	$-x$	0	x	$\cdots$
$-3x$	$-x$	0	x	$\cdots$

Fruit 2.1 takes $x = 5$ in the opening, and $x = 0$ in the endgame. Rybka 1.0 Beta takes $x = 181$ in the opening, and $x = -97$ in the endgame. Fruit adds 10 to `d3/e3/d5/e5` and 20 to `d4/e4`, while Rybka adds 74 to `d5/e5`.

```
static const int PawnFile[8] = { -3, -1, +0, +1, +1, +0, -1, -3 };
```

My personal impression is that if the above were the totality of the evidence, it would be dismissible (the grid does not look that odd), but when in the context of everything else, it becomes more pressing. It can also be noted that many (if not most) other chess programs have a rank-dependence in Pawn PST.

[7] A silly counterpoint to this could be that the arrays contain 12 numbers (though not all are really "independent", as one fully expects the numbers to be higher at the centre than at the edge), which is a lot more than the 3 values x, y, z.

[8] I might say that this is especially true given the re-scaling done by Rybka 1.0 Beta to use 3399ths of a pawn rather than centipawns – why should the Rank/Line array values stay small (single digits), and the x, y values grow? But this is perhaps trying to read minds...

In can be noted that the White pawn values are loaded into the 256 bytes starting at location `0x64bdf0` in the 64-bit version, with the first 2 bytes for the opening value of `a1`, then 2 for the endgame value, then the same for `b1`, `c1`, etc. After this comes Black pawns, then White knights, etc.

5.2.2 Knights PST endgame

Table 6: Common PST schematic for knights in the endgame

$-4x-4x$	$-4x-2x$	$-4x+0$	$-4x+x$	$\cdots$
$-2x-4x$	$-2x-2x$	$-2x+0$	$-2x+x$	$\cdots$
$0-4x$	$0-2x$	$0+0$	$0+x$	$\cdots$
$x-4x$	$x-2x$	$x+0$	$x+x$	$\cdots$
$x-4x$	$x-2x$	$x+0$	$x+x$	$\cdots$
$0-4x$	$0-2x$	$0+0$	$0+x$	$\cdots$
$-2x-4x$	$-2x-2x$	$-2x+0$	$-2x+x$	$\cdots$
$-4x-4x$	$-4x-2x$	$-4x+0$	$-4x+x$	$\cdots$

This is essentially the same as the Knights in the opening, except that the rank bonus (the y-variable of before) is absent, as is the `a8/h8` penalty. Fruit 2.1 takes $x = 5$ while Rybka 1.0 Beta takes $x = 56$. As before, the main content here is not the general "centralisation", but the exact weightings from

```
static const int KnightLine[8] = { -4, -2, +0, +1, +1, +0, -2, -4 };
```

5.2.3 Bishops PST

Table 7: Common PST schematic for white bishops

$-3x-3x+y$	$-3x-x$	$-3x+0$	$-3x+x$	$\cdots$
$-x-3x$	$-x-x+y$	$-x+0$	$-x+x$	$\cdots$
$0-3x$	$0-x$	$0+0+y$	$0+x$	$\cdots$
$x-3x$	$x-x$	$x+0$	$x+x+y$	$\cdots$
$x-3x$	$x-x$	$x+0$	$x+x+y$	$\cdots$
$0-3x$	$0-x$	$0+0+y$	$0+x$	$\cdots$
$-x-3x$	$-x-x+y$	$-x+0$	$-x+x$	$\cdots$
$-3x-3x+y-z$	$-3x-x-z$	$-3x+0-z$	$-3x+x-z$	$\cdots$

Fruit 2.1 has $(x, y, z) = (2, 4, 10)$ in the opening and $(x, y, z) = (3, 0, 0)$ in the endgame. Rybka 1.0 Beta has $(x, y, z) = (147, 378, 251)$ and $(x, y, z) = (49, 0, 0)$.

The principal x-weighting is the same as with `PawnFile/QueenLine/KingLine`.

```
static const int BishopLine[8] = { -3, -1, +0, +1, +1, +0, -1, -3 };
```

One might expect some of these to be re-used, but the fact that Rybka 1.0 Beta and Fruit 2.1 use the exact same arrays in the exact same places makes this of more import. Furthermore, Rybka 1.0 Beta has the same type of penalties (`BackRank/Diagonal`) as Fruit 2.1 in the opening (in fact, it might have been better to make two separate grids for opening/endgame, to show that both have $y = z = 0$ for the endgame values).

5.2.4 Rooks PST

Table 8: Common PST schematic for rooks (opening)

$-2x$	$-x$	0	x	$\cdots$
$-2x$	$-x$	0	x	$\cdots$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
$-2x$	$-x$	0	x	$\cdots$

This one is almost so mundane as to pass without comment. Fruit 2.1 has $x = 3$ and Rybka 1.0 Beta has $x = 104$, and both have $x = 0$ in the endgame.

```
static const int RookFile[8] = { -2, -1, +0, +1, +1, +0, -1, -2 };
```

Again the principal query would be as to why was *this* RookFile chosen in both, as opposed to (say) re-using the PawnFile array instead.

5.2.5 Queens PST

Table 9: Common PST schematic for white queens

$-3x - 3x$	$-3x - x$	$-3x + 0$	$-3x + x$	$\cdots$
$-x - 3x$	$-x - x$	$-x + 0$	$-x + x$	$\cdots$
$0 - 3x$	$0 - x$	$0 + 0$	$0 + x$	$\cdots$
$x - 3x$	$x - x$	$x + 0$	$x + x$	$\cdots$
$x - 3x$	$x - x$	$x + 0$	$x + x$	$\cdots$
$0 - 3x$	$0 - x$	$0 + 0$	$0 + x$	$\cdots$
$-x - 3x$	$-x - x$	$-x + 0$	$-x + x$	$\cdots$
$-3x - 3x - z$	$-3x - x - z$	$-3x + 0 - z$	$-3x + x - z$	$\cdots$

This one is a bit tricky, as Fruit has a zero value for `QueenCentreOpening`, though it is explicitly in the code. And again (see Bishops) there is a `BackRank` penalty only in the opening (in both). Fruit 2.1 has $(x, z) = (0, 5)$ in the opening and $(x, z) = (4, 0)$ in the endgame, while Rybka 1.0 Beta has $(x, z) = (98, 201)$ in the opening and $(x, z) = (108, 0)$ in the endgame. Again having a separate grid for the endgame might make the `BackRank` penalty more clear.

```
static const int QueenLine[8] = { -3, -1, +0, +1, +1, +0, -1, -3 };
```

5.2.6 Kings PST

Table 10: Common PST schematic for white kings (opening)

$3x - 7y$	$4x - 7y$	$2x - 7y$	$0 - 7y$	$\cdots$
$3x - 6y$	$4x - 6y$	$2x - 6y$	$0 - 6y$	$\cdots$
$3x - 5y$	$4x - 5y$	$2x - 5y$	$0 - 5y$	$\cdots$
$3x - 4y$	$4x - 4y$	$2x - 4y$	$0 - 4y$	$\cdots$
$3x - 3y$	$4x - 3y$	$2x - 3y$	$0 - 3y$	$\cdots$
$3x - 2y$	$4x - 2y$	$2x - 2y$	$0 - 2y$	$\cdots$
$3x + 0$	$4x + 0$	$2x + 0$	$0 + 0$	$\cdots$
$3x + y$	$4x + y$	$2x + y$	$0 + y$	$\cdots$

Again there is a somewhat of a stretch here in making a common schematic, as Rybka 1.0 Beta doesn't have the adjustment for `KingRankOpening`, so the y-variable of above only appears in Fruit 2.1. However, the file array does match.

```
static const int KingFile[8] = { +3, +4, +2, +0, +0, +2, +4, +3 };
```
Fruit 2.1 has $(x, y) = (10, 10)$ and Rybka 1.0 Beta has $(x, y) = (469, 0)$.

The schematic in the endgame, and the `KingLine` array used for it, are the same as with bishops and queens above (based on centralisation). Fruit 2.1 has $x = 12$ and Rybka 1.0 Beta has $x = 401$.

```
static const int KingLine[8] = { -3, -1, +0, +1, +1, +0, -1, -3 };
```

6 Things of lesser importance

6.1 Data structures with hashing

The first 64 bits of the hash structure in Rybka and Fruit are used in the same manner. I can find no other engines that use this – even Fruit 1.0 differs (having a 64-bit lock). The common parts are:

a 32-bit lock, 16 bits for the move, 8 bits for depth, and 8 bits for date.

To choose a random comparison, Faile orders these as [hash, depth, score, move] with differing bit widths.

6.2 Use of a quad()-like function for passed pawns

6.2.1 The quad() function in Fruit 2.1

This is the `quad()` function in Fruit 2.1, with `ASSERT`s stripped out, and the `Bonus` values above made explicit, with my comment about percentage of 256.

```
   Bonus[Rank4] = 26;  // 10.15625%
   Bonus[Rank5] = 77;  // 30.078125%
   Bonus[Rank6] = 154; // 60.15625%
   Bonus[Rank7] = 256; // 100%
int quad(int y_min, int y_max, int x)
{return y_min + ((y_max - y_min) * Bonus[x] + 128) / 256;}
```

As can be seen, this function uses approximately a 10-30-60-100 weighting, given instead by "hexapawns" as 26-77-154-256. Also, the function rounds (with the +128) to the nearest integer (rather than trunctating) in the division by 256. The only difference between the values that `quad()` returns and those of the arrays included in Rybka 1.0 Beta are that the latter seems to truncate rather than round.

Fruit 2.1 uses these for a number of passed pawn eval components. In every case, when a pawn on a given rank has a specific attribute, the `quad()` returns the score to be applied in the evaluation.

```
static const int PassedOpeningMin = 10;
static const int PassedOpeningMax = 70;
static const int PassedEndgameMin = 20;
static const int PassedEndgameMax = 140;
static const int AttackerDistance = 5;
```

```
static const int DefenderDistance = 20;
static const int CandidateOpeningMin = 5;
static const int CandidateOpeningMax = 55;
static const int CandidateEndgameMin = 10;
static const int CandidateEndgameMax = 110;
```

There are also 2 other constant bonuses, the first of which is also a constant in Rybka 1.0 Beta, while the second is divided up into more cases and given the `quad()`-style weighting based on rank.

```
static const int UnstoppablePasser = 800; // always 800 in Fruit 2.1
static const int FreePasser = 60;         // always 60  in Fruit 2.1
```

Not all of these terms have exactly the same meaning in Rybka 1.0 Beta, and discussing any differences would diverge from my focus on the re-use of `quad()` function. Perhaps the main difference is with `FreePasser`, as to whether the pawn's path is met by a friendly or enemy piece, which uses SEE in Fruit and "attacks" bitboards in Rybka.

6.2.2 Passed pawn numerology in Rybka 1.0 Beta

As noted above, Fruit calls `quad()` every time, while one can note that the values are only dependent on the rank, and then use precomputation to get array values as in Rybka 1.0 Beta. The elements of the arrays in Rybka 1.0 Beta are **not** direct outputs of the `quad()` function in Fruit 2.1, but up to rounding (note the "+128" in the code above), this is indeed the case.

Here are the values in the arrays for Rybka 1.0 Beta, indexed by rank:

```
int PassedOpening[8] = { 0, 0, 0, 489, 1450, 2900, 4821, 4821 };
int PassedEndgame[8] = { 146, 146, 146, 336, 709, 1273, 2020, 2020 };
int PassedUnblockedOwn[8] = { 0, 0, 0, 26, 78, 157, 262, 262 };
int PassedUnblockedOpp[8] = { 0, 0, 0, 133, 394, 788, 1311, 1311 };
int PassedFree[8] = { 0, 0, 0, 101, 300, 601, 1000, 1000 };
int PassedAttDistance[8] = { 0, 0, 0, 66, 195, 391, 650, 650 };
int PassedDefDistance[8] = { 0, 0, 0, 131, 389, 779, 1295, 1295 };
int CandidateOpening[8] = { 0, 0, 0, 382, 1131, 2263, 3763, 3763 };
int CandidateEndgame[8] = { 18, 18, 18, 181, 501, 985, 1626, 1626 };
```

Perhaps the most obvious "sore-thumb" is the `PassedFree` array, which looks mighty close to 100-300-600-1000, but is off-by-one in two entries. Indeed, this is accounted for exactly by the "hexapawns" rescaling. Here are inputs to a `quad()`-like function (with different rounding) to produce the Rybka arrays.

```
PassedOpening:       Min = 0, Max = 4821
PassedEndgame:       Min = 146, Max = 2020
PassedUnblockedOwn:  Min = 0, Max = 262
PassedUnblockedOpp:  Min = 0, Max = 1311
PassedFree:          Min = 0, Max = 1000
```

```
PassedAttDistance:   Min = 0, Max = 650
PassedDefDistance:   Min = 0, Max = 1295
CandidateOpening:    Min = 0, Max = 3763
CandidateEndgame:    Min = 18, Max = 1626
```

6.2.3 Impact of this evidence

It is fairly clear (particularly with the `PassedFree` array) that the values in Rybka 1.0 Beta were generated automatically, and not by hand. The use of `quad()`-like function is not sufficiently generic for its output to be considered commonplace (in computer chess or otherwise).

The counterpoint to this is whether the use of `quad()` is really all that important. At one level, the numbers used in an evaluation function certainly do have an impact on playing strength of an engine. On the other hand, it can be said that Rybka is basically using the 10-30-60-100 scaling "idea" (not all that novel), with a choice of weights that is not too similar to that of Fruit.

Finally, it can be noted that the quirky 256-based scaling seems to be done for a reason in Fruit 2.1 (as the `quad()` function is called every time, perhaps general integer division would be too slow), while when the array is pre-computed as in Rybka, it is a bit inscrutable to me why the direct 10-30-60-100 scaling would not be preferable.

6.3 UCI parsing

Finally there is the subject of UCI parsing. Some of this is not precisely "chess-related", though there is overlap with time management issues, and some of this could also be useful for GPL purposes.

6.3.1 Parsing the "position" string

One example of copying seems to be in how Rybka parses the "position" string. The Fruit code has various oddities, such as

```
moves[-1] = '\0'; // dirty, but so is UCI
```

A disassembly of the Rybka code shows a similar hack.

Here is the stripped-down Fruit code:

```
fen = strstr(string,"fen ");
moves = strstr(string,"moves ");
if (fen != NULL) {
   if (moves != NULL) { // "moves" present
      moves[-1] = '\0'; // dirty, but so is UCI
}
board_from_fen(SearchInput->board,fen+4); // CHANGE ME
// else use startpos -- omitted here
if (moves != NULL) { // "moves" present
   ptr = moves + 6;
   while (*ptr != '\0') { // until string is terminated
```

```
        [...] // some code to get the move_string
        move = move_from_string(move_string,SearchInput->board);
        move_do(SearchInput->board,move,undo);
        while (*ptr == ' ') ptr++; // eliminates spaces
      }
   }
```

Here is a Rybka decompilation from Franklin Titus (using the 32-bit version).

```
int __usercall sub_4092E0<eax>(const char *a1<eax>)
{
  char *v1; // esi@1
  const char *v2; // esi@1
  char *v3; // edi@1
  int v4; // esi@6
  int v5; // eax@7
  v2 = a1;
  v3 = strstr(a1, "fen");
  v1 = strstr(v2, "moves");
  sub_403490();      // board_from_fen, for startpos
  if ( v3 ) // fen != NULL
  {
    if ( v1 ) // moves != NULL
      *(v1 - 1) = 0; // moves[-1] = 0, would the compiler do this on its own?
    sub_403490();    // board_from_fen for actual fen -- maybe sub_403490(v3)?
  }
  if ( v1 )   // "moves" present
  {
    v4 = (v1 + 6); // ptr = moves + 6
    while ( *v4 ) // until string is terminated
    {
      v5 = sub_40AAF0(v4); //
      sub_40ABC0(v5);      // some code to get the move_string
      v4 += 5;             // Fruit increments on each character (*ptr++)
      if ( !*(v4 - 1) )    // the -1 here is likely compiler-based
        break;             // i.e. v4[4] is the same as (v4 + 5)[-1]
      for ( ; *v4 == 32; ++v4 ) // eliminates spaces
        ;
    }
  }
  return sub_401100();
}
```

The most interesting part is likely that `moves[-1]` reappears in the Rybka code, while I'm still not sure of its exact purpose in the Fruit code (it appears to ensure the FEN string is `NUL`-terminated, but this is not strictly necessary). The Rybka version could, however, be a compiler optimisation of `fen[moves-fen-1]`, which looks a bit better in `C`. In any case, the fact that "something is done" here that (in the end) serves no purpose makes this a mentionable commonality.

6.3.2 Time management

Referring to Rick Fadden's disassembly efforts available from TalkChess at `http://www.talkchess.com/forum/viewtopic.php?p=187290`, there are various other common elements between Fruit and Rybka that can be mentioned here. The most notable is (the naming of variables is his):

```
// Rybka compares movetime with a double precision value: 0.0
  if (movetime >= 0.0) {
    time_limit_1 =    5 * movetime;
    time_limit_2 = 1000 * movetime;
  } else if (time > 0) {
    time_max = time - 5000;
    alloc = (time_max + inc * (movestogo - 1)) / movestogo;
    if (alloc >= time_max) alloc = time_max;
    time_limit_1 = alloc;
    alloc = (time_max + inc * (movestogo - 1)) / 2;
    if (alloc < time_limit_1) alloc = time_limit_1;
    if (alloc > time_max) alloc = time_max;
    time_limit_2 = alloc;
  }
```

The Fruit code in comparison is

```
   if (movetime >= 0.0) {
      SearchInput->time_is_limited = true;
      SearchInput->time_limit_1 = movetime * 5.0; // HACK to avoid early exit
      SearchInput->time_limit_2 = movetime;
   } else if (time >= 0.0) {
      time_max = time * 0.95 - 1.0;
      if (time_max < 0.0) time_max = 0.0;
      SearchInput->time_is_limited = true;
      alloc = (time_max + inc * double(movestogo-1)) / double(movestogo);
      alloc *= (option_get_bool(``Ponder'') ? PonderRatio : NormalRatio);
      if (alloc > time_max) alloc = time_max;
      SearchInput->time_limit_1 = alloc;
      alloc = (time_max + inc * double(movestogo-1)) * 0.5;
      if (alloc < SearchInput->time_limit_1) alloc = SearchInput->time_limit_1;
      if (alloc > time_max) alloc = time_max;
      SearchInput->time_limit_2 = alloc;
   }
```

As noted by Zach Wegner and others, the comparison with a floating-point value in Rybka is simply bizarre in itself (it appears in both the 32-bit and 64-bit versions), and only when put side-by-side with the Fruit code (for which it makes sense) does the genesis of this come to light. The multiplication of `movetime` by 5 to get `time_limit_1` is another common element. Further similarities could be mentioned, but it not so clear how important they are.

Another component here is the final block of code in the "go" parser. Here is the Fruit code.

```
if (infinite || ponder) SearchInput->infinite = true;
ASSERT(!Searching);
ASSERT(!Delay);
Searching = true;
Infinite = infinite || ponder;
Delay = false;
search();
search_update_current();
ASSERT(Searching);
ASSERT(!Delay);
Searching = false;
Delay = Infinite;
if (!Delay) send_best_move();
```

Here is the comparative Rybka code from the 64-bit disassembly.[9]

```
0x0040702e: test   %r15b,%r15b           # r15 is "infinite"
0x0040704d: jne    0x407054
0x0040704f: test   %r13b,%r13b           # r13 is "ponder"
0x00407052: je     0x407063* [0x40705b]  # if either is true
0x00407054: movb   $0x1,0x2652d1(%rip) # 0x66c32c SearchInput->Infinite = true
0x00407063* movb   $0x1,0x262677(%rip)   # 0x6696e1  set Searching = true
0x0040705b: test   %r15b,%r15b           # check "infinite" again
0x0040706a: jne    0x40707a
0x0040706c: test   %r13b,%r13b           # check "ponder" again
0x0040706f: jne    0x40707a              # if both false,
0x00407071: mov    %r13b,0x26266a(%rip)     # 0x6696e2, set Infinite = false
0x00407078: jmp    0x407081              # else
0x0040707a: movb   $0x1,0x262661(%rip)      # 0x6696e2, set Infinite = true
0x00407081: movb   $0x0,0x26265b(%rip)   # 0x6696e3  set Delay = false
0x00407088: callq  0x408f90              # call the search function
0x0040708d: movzbl 0x26264e(%rip),%eax   # 0x6696e2  load Infinite variable
0x0040709b: movb   $0x0,0x26263f(%rip)   # 0x6696e1  set Searching = false
0x004070a2: mov    %al,0x26263b(%rip)    # 0x6696e3  set Delay = Infinite
0x00407094* test   %al,%al               # if Infinite
0x004070a8: jne    0x4070af              #  then don't
0x004070aa: callq  0x406aa0              #  call send_best_move()
```

Note in particular that `(infinite || ponder)` gets computed twice in both of these. Also, the ordering of `Searching`, `Infinite`, `Delay` is the same in each. Finally, setting `Delay` to be "false" is redundant, as three lines above it was `ASSERT`ed to be so. There are a few other atypical similarities in UCI parsing (e.g., incremental use of `strtok`), but I won't discuss them here, and leave Fadden's disassembly/decompilation to suffice.

[9] I omit various instructions regarding register preparation for the function-call exit, and also re-ordered 0x407063 and 0x407094 to better emphasise the if-then logic in comparisions.

A Root search analysis: iterative deepening

This appendix is a Fruit/Rybka comparison of a specific "chunk" of code. It is my hope that it will exemplify various aspects of the similarities and differences. I first give the Fruit code, then the Rybka disassembly, and finally a `C` translation of the latter.

The Fruit code, reformatted, `ASSERT`s removed, with applicable comments.

```
for (depth = 1; depth < DepthMax; depth++) // DepthMax is 64
{ if (DispDepthStart) send("info depth %d",depth); // DispDepthStart is true
  SearchRoot->bad_1 = false;
  SearchRoot->change = false;
  board_copy(SearchCurrent->board,SearchInput->board);
  if (UseShortSearch && depth <= ShortSearchDepth) // UseShortSearch is true
   search_full_root(SearchRoot->list,SearchCurrent->board,depth,SearchShort);
  else
   search_full_root(SearchRoot->list,SearchCurrent->board,depth,SearchNormal);
  search_update_current();
  if (DispDepthEnd) send("[...]"); // a complicated construct, omitted here
  if (depth >= 1) SearchInfo->can_stop = true;
  if (depth == 1 && LIST_SIZE(SearchRoot->list) >= 2
      && LIST_VALUE(SearchRoot->list,0) >=
         LIST_VALUE(SearchRoot->list,1) + EasyThreshold) // this is 150
   SearchRoot->easy = true;
  if (UseBad && depth > 1) // UseBad is true
  { SearchRoot->bad_2 = SearchRoot->bad_1;
    SearchRoot->bad_1 = false; }
  SearchRoot->last_value = SearchBest->value;
  if (SearchInput->depth_is_limited && depth >= SearchInput->depth_limit)
   SearchRoot->flag = true;
  if (SearchInput->time_is_limited
      && SearchCurrent->time >= SearchInput->time_limit_1
      && !SearchRoot->bad_2)
   SearchRoot->flag = true;
  if (UseEasy && SearchInput->time_is_limited // UseEasy is true
      && SearchCurrent->time >= SearchInput->time_limit_1 * EasyRatio // 0.20
      && SearchRoot->easy)
   SearchRoot->flag = true;
  if (UseEarly && SearchInput->time_is_limited // UseEarly is true
      && SearchCurrent->time >= SearchInput->time_limit_1 * EarlyRatio // 0.60
      && !SearchRoot->bad_2 && !SearchRoot->change)
   SearchRoot->flag = true;
  if (SearchInfo->can_stop
      && (SearchInfo->stop || (SearchRoot->flag && !SearchInput->infinite)))
   break;
}
```

Here is the disassembly from the Rybka 1.0 Beta 64-bit version with comments.

```
0x04095a5: mov    $0x1,%esi
0x04095aa: mov    %esi,%ebx
0x04095b0: cmp    $0x5,%ebx            # compare depth to 5
0x04095b3: jb     0x4095c4             # if at least 5
0x04095b5: lea    -0x2(%rbx),%edx          then subtract 2 before...
0x04095b8: lea    0x25af79(%rip),%rcx  # 0x664538 [info depth string]
0x04095bf: callq  0x40d0b0             # ...printing the ‘‘info depth’’ string
0x04095c4: mov    %ebx,%ecx
0x04095c6: movb   $0x0,0x262e81(%rip)  # 0x66c44e set ‘‘change’’ to false
0x04095cd: movb   $0x0,0x262e78(%rip)  # 0x66c44c  set ‘‘bad_1’’ to false
0x04095d4: callq  0x40ba70             # call search_full_root
0x04095d9: callq  0x4070c0             # some sort of update function
0x04095de: mov    0x26706f(%rip),%r11d # 0x670654, get score
0x04095e5: cmp    $0xffff8300,%r11d    # fiddle around
0x04095ec: jle    0x4095fe             # ...
0x04095ee: cmp    $0x7d00,%r11d        # with mate scores
0x04095f5: movzbl 0x262e54(%rip),%edx  # 0x66c450 load ‘‘flag’’
0x04095fc: jl     0x409601             # if mate score,
0x04095fe: mov    %sil,%dl               # set flag to true (esi is always 1)
0x0409601: cmp    %esi,%ebx            # compare depth (%ebx) to 1
0x0409603: jne    0x409631
0x0409605: cmpl   $0x0,0x267058(%rip)  # 0x670664 think this is RML[1]
0x040960c: je     0x40962f             # if only one legal move, skip next
0x040960e: movzbl 0x262e3a(%rip),%ecx  # 0x66c44f ‘‘easy’’
0x0409615: mov    0x267449(%rip),%eax  # 0x670a64 (value of move 1)
0x040961b: add    $0x96,%eax           # EasyThreshold of 150 [as in Fruit]
0x0409620: cmp    %eax,0x26743a(%rip)  # 0x670a60 (value of move 0)
0x0409626: cmovae %esi,%ecx            # if move values differ by enough
0x0409629: mov    %cl,0x262e20(%rip)     # 0x66c44f  set ‘‘easy’’ as true
0x040962f: cmp    %esi,%ebx            # if depth > 1
0x0409631: jbe    0x409647
0x0409633: movzbl 0x262e12(%rip),%eax    # 0x66c44c  load old bad_1
0x040963a: movb   $0x0,0x262e0b(%rip)    # 0x66c44c  bad_1 = false
0x0409641: mov    %al,0x262e06(%rip)     # 0x66c44d  bad_2 = (previous) bad_1
0x0409650* mov    %r11d,0x262df1(%rip)   # 0x66c448  last_value = score
0x0409647: cmp    0x262cdb(%rip),%ebx  # 0x66c328  see if depth>=depth_limit
0x040964d: movzbl %dl,%eax             # %dl: 1@4095fe (mate), ‘‘flag’’@4095f5
0x0409657: cmovae %esi,%eax            # if depth>=depth_limit
0x040965a: mov    %al,0x262df0(%rip)     # 0x66c450  then set ‘‘flag’’
0x0409666: mov    0x262cb3(%rip),%r8d # 0x66c320 load SearchInput->time_limit_1
0x040966d: movzbl 0x262dd8(%rip),%r9d  # 0x66c44d  load bad_2
0x0409660* callq  *0x139ca(%rip)       # 0x41d030  GetTickCount -> %eax
0x0409675: mov    %eax,%r11d
0x040967c* sub    0x262db5(%rip),%r11d # 0x66c438  (subtract StartTime)
0x0409678: lea    (%r8,%r8,1),%ecx     # 3 * time_limit_1
```

```
0x0409683: mov    $0xaaaaaaab,%eax     #  then mult by 2/3
0x0409688: mul    %ecx
0x040968a: shr    %edx                 #  and div by 2 ... hmm = time_limit_1 ?
0x040968c: cmp    %edx,%r11d           # compare to time taken
0x040968f: jb     0x4096a6               # if small, ignore next
0x0409691: movzbl 0x262db8(%rip),%ecx  # 0x66c450 ‘‘flag’’
0x0409698: test   %r9b,%r9b            # if bad_2 is 0 [that is, false]
0x040969b: cmove  %esi,%ecx              # ecx = 1 (esi is always 1)
0x040969e: mov    %cl,0x262dac(%rip)   # 0x66c450 store ecx in ‘‘flag’’
0x04096a4: jmp    0x4096ac
0x04096a6: mov    0x262da4(%rip),%cl   # 0x66c450
0x04096ac: mov    $0xaaaaaaab,%eax
0x04096b1: mul    %r8d                 # mult time_limit_1 by 2/3
0x04096b4: shr    $0x2,%edx            #  and div by 4
0x04096b7: cmp    %edx,%r11d           # compare to time taken
0x04096ba: jb     0x4096d1               # if small, ignore next
0x04096bc: cmpb   $0x0,0x262d8c(%rip)  # 0x66c44f see if ‘‘easy’’
0x04096c3: movzbl %cl,%eax             # if not ‘‘easy’’, then eax is ‘‘flag’’
0x04096c6: cmovne %esi,%eax            # if it is ‘‘easy’’, then eax is ‘‘true’’
0x04096c9: mov    %al,%cl
0x04096cb: mov    %al,0x262d7f(%rip)   # 0x66c450 store eax in ‘‘flag’’
0x04096d1: shr    %r8d                 # time_limit_1 divided by 2
0x04096d4: cmp    %r8d,%r11d           # compare to time taken
0x04096d7: jb     0x4096f3               # if small, ignore next
0x04096d9: test   %r9b,%r9b            # if bad_2 is true
0x04096dc: jne    0x4096f3               # then ignore next
0x04096de: cmp    %r9b,0x262d69(%rip)  # 0x66c44e ‘‘change’’, see if ‘‘false’’
0x04096e5: movzbl %cl,%eax             # if not, then eax is as above
0x04096e8: cmove  %esi,%eax            # if ‘‘change’’ is ‘‘false’’ -> eax ‘‘true’’
0x04096eb: mov    %al,%cl
0x04096ed: mov    %al,0x262d5d(%rip)   # 0x66c450 store eax in ‘‘flag’’
0x04096f3: cmpb   $0x0,0x262d36(%rip)  # 0x66c430 see if ‘‘stop’’ is true
0x04096fa: jne    0x409714             # if so, then exit this function
0x04096fc: test   %cl,%cl              # see if ‘‘flag’’ is true
0x04096fe: je     0x409709               # if so
0x0409700: cmpb   $0x0,0x262c25(%rip)    # 0x66c32c and SearchInput->infinite
0x0409707: je     0x409714               # is false, then exit this function
0x0409709: add    %esi,%ebx            # increment depth
0x040970b: cmp    $0x48,%ebx           # if depth < 72
0x040970e: jb     0x4095b0             # then loop
```

The asterisks here denote instructions that I have re-ordered, typically when the ASM code starts laying the groundwork for the next operation prior to the completion of the previous.

Here is a translation into a higher-level language, with Fruit as a template.

```
for (depth = 1; depth < 72; depth++)
{ if (depth >= 5) printf("info depth %d\n",depth-2);
  change = false;
  bad_1 = false; // order is switched from Fruit -- could be the compiler
  search_full_root(depth-2); // yields "score" in a global var
  some_sort_of_update_function();
  if (score <= -32000 || score >= 32000) // mate scores
   flag = true;
  if (depth == 1 && RootMoveList[1].move != MOVE_NONE &&
      RootMoveList[0].value >= RootMoveList[1].value + 150) // 409601-40962f
   easy = true;
  if (depth > 1) // 409631-409641
   {bad_2 = bad_1;
    bad_1 = false;}
  last_value = score;
  if (depth >= depth_limit) // 409647
   flag = true;
  TimeUsed = GetTickCount() - StartTime;
  if ((3*time_limit_1)/3 <= TimeUsed && !bad_2) // 409691, has mult/div by 3
   flag = true;
  if ((time_limit_1)/6 <= TimeUsed && easy) // 20% in Fruit
   flag = true;
  if ((time_limit_1)/2 <= TimeUsed && !bad_2 && !change) // 60% in Fruit
   flag = true;
  if (stop || (flag && !SearchInput->infinite))
   break;
}
```

As can be seen, there are various differences, but (particularly in the ordering of various parts) there still seems to be more similarities than one might expect.[10]

It can also be noted that the 6 variables in Rybka 1.0 Beta here are allocated in exactly the same order as in comparative Fruit code.

```
struct search_root_t {
[...]
   int last_value; // 66c448
   bool bad_1;     // 66c44c
   bool bad_2;     // 66c44d
   bool change;    // 66c44e
   bool easy;      // 66c44f
   bool flag;      // 66c450
};
```

[10] For instance, all the settings of `flag` can be re-ordered, as can the parts of the compound `&&` statements. Another point is that condition "`depth > 1`" (before the `bad_` indicators are updated) looks somewhat superfluous and/or unnecessary.

B Other comments

In this appendix, I pull together some other comments I have made regarding Rybka and Fruit. In particular, as this document is largely aimed at listing similarities, it might be good to list some differences also.

B.1 Principal differences for Rybka 1.0 Beta and Fruit 2.1

Rybka 1.0 Beta uses bitboards, has an extensive material imbalance table, and does not have the history pruning of Fruit 2.1, but rather a more vanilla LMR approach that considers only the location in the move list. The weightings of evaluation features are also of course different (and perhaps more well-tuned).

The search function of Rybka 1.0 Beta follows that of Fruit 2.1 in a vague sense (probably not more than one would expect from two PVS engines), but seems not to be as mature; for instance, Rybka 1.0 Beta simply announces mate and does not try to minimise the distance to it (which can lead to overly lengthy mating chases). The principal "new idea" in search seems to me to be the loosening of the stringent cutoff values for futility previously used in the AEL pruning of Heinz.

These add up to about a 75-100 Elo improvement on a 32-bit machine, comparable to the amount that Fruit itself gained over the second half of 2005.

B.2 Later versions of Rybka

I have not checked every version of Rybka, but have verified that the evaluation function in Rybka 2.3.2a is substantially the same as in Rybka 1.0 Beta. Much of the numerology is still extant, and only a few features have changed.[11] The re-writing of the evaluation function in Rybka 3 (by Larry Kaufman) has removed much of any complaint here.

For the search, I have not checked too much, but my impression is that the search was already being modified in the various Rybka 1.01 Betas (there were 13 iterations of this in all, with Rybka 1.1 released on March 16, 2006). I have not bothered to see when/if the UCI parsing was changed, though a crude check[12] finds code similarities in Rybka 2.1o, but not in Rybka 2.3.2a.

B.3 The question of a re-write

Another question is whether Rybka 1.0 Beta is to be considered as a "successor" of earlier Rybka versions, as opposed to being a complete re-write. A version called Rybka 1.5.32 from April 2004 participated in Chess War V (Olivier Deville) and Le Système du Suisse (Claude Dubois). I have been unable to obtain a copy of this version, or any other pre-Beta versions.

Rybka 1.5.32 is a UCI engine, which should seem to make any copying of the Fruit UCI and/or time management code rather unnecessary. It also played

[11] For instance, knight mobility was removed, and pawn anti-mobility was added.

[12] I searched for adding 6 (see `ptr = moves + 6`), and looked at surrounding code segments.

two rook underpromotions in approximately 60 games, whereas Rybka 1.0 Beta was not able to play underpromotions.

On the other hand, Rajlich fully acknowledges that Rybka went through a lot of twists and turns in the early years, especially as he started with an `MTD(f)`-based search rather than PVS.[13] It remains unclear, however, why underpromotions would disappear, or why the UCI parsing and time management would follow the Fruit template so closely.

B.4 Statements of Rajlich

This is perhaps not the best place to dredge up statements of Rajlich, but I list here two specific ones. Here "Rybka" would seem to refer to Rybka 1.0 Beta (having "Rybka" mean Rybka 1.5.32 would contextually be *non sequitur*).

The first is in reply to Daniel Mehrmann, whose tests suggested the mobility and PST in Rybka 1.0 Beta was from Fruit 2.1 (Mehrmann later retracted this).

```
Subject: Re: Rybka - How much Fruit is inside ?
From: Vasik Rajlich
Message Number: 469187
Date: December 12, 2005 at 03:34:15
[...]
The Rybka source code is original and pre-dates all of the Fruit releases.
[...]
```

A few days later, in response to Andrew Wagner asking him whether he had done anything radically different, Rajlich uses the phrase "very original" to describe the search and evaluation framework of Rybka.

```
Subject: Re: Unmasking the Secrets of Rybka and Fruit
From: Vasik Rajlich
Message Number: 470751
Date: December 16, 2005 at 03:42:44

> [...] If I were able to ask Vasik one question, which I doubt he would have
> time to answer at the moment, it would be whether he did anything radically
> different (different heuristic(s), algorithms, etc.), or if he just did what
> everyone else is doing, better than they did it.

Andy,

I will just end up teasing you by answering this. :)

As far as I know, Rybka has a very original search and evaluation framework.
A lot of things that have been dismissed by "computer chess practice" can
in fact work. [...]
```

[13] One can note that both Fruit and Rybka store both upper and lower score bounds in hash entries. This is not overly common with PVS engines, but is more valuable when using `MTD(f)`, and it is plausible that Rybka 1.0 Beta inherited this from earlier `MTD(f)` days.