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 24, 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 a 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 firstly 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] 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 simulator of the Cray Blitz evaluation) due to the fact that it had been taken from the HITECH project at an earlier time.

Expanding on this, merely re-typing and/or "translating" code (possibly while tweaking/tuning some of the numbers) is also not likely to pass an "originality" test. There is not, however, a cleanly delineated line between "code" and "ideas" (for instance), and I really can't give much guidance for that. This report can perhaps be seen as 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]

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 I am largely going to avoid GPL and/or copyright issues.

The lesser pieces of evidences include:

- various similarities in data structures in hashing;
- the re-appearance of 10-30-60-100 scaling implemented as 26-77-154-256;
- 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 than "code" for various purposes; the standard I adopt herein makes some distinction between the two, but 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 2.1 was evaluation based primarily on mobility. Below I compare the Fruit 2.1 evaluation routine to that found in Rybka 1.0 Beta. While a large (indeed, almost complete) match is found, it is presumably feasible to opine 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 accepted, then the re-use of the same evaluation components might arguably be less 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.[3] 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   Knights (see `0x4018d0` and `0x401eb0` in 64-bit Rybka 1.0 Beta)

Both Rybka and Fruit have mobility as the only knight evaluation component.[4]

3.1.2   Bishops (`0x401971-0401a3a` and `0x401f60-0x40202a`)

Both Rybka and Fruit consider only mobility as the primary evaluation component with bishops. Both Rybka (`0x4026b3-0x402745`) and Fruit have a trapped bishop penalty; the same definition of "trapped" is used in each (computed via bitboards in Rybka), though the penalty is halved in Fruit in one case. Both Rybka (`0x40274b-0x4027f6`) and Fruit have a blocked bishop penalty for (say) a bishop on `c1`, a friendly pawn on `d2`, and a blocking unit on `d3`. Both Rybka (`0x402885-0x4028ba`) and Fruit halve the overall evaluation in an opposite-colour bishop endgame when the pawn counts differ by no more than 2 (both via a flag in a material table, then a separate test for opposite colouring).

---

[3]Note that Rybka 1.0 Beta uses bitboards, making direct code comparison ineffective.

[4]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 (`0x401a70-0x401bf8` and `0x402064-0x4021a7`)

Both Rybka and Fruit consider mobility, open files, semi-open files, whether the opposing king is on/adjacent to an semi-open file (with an identical "sufficient material" criterion for this), 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 7th rank, or the king on the 8th rank. Both Rybka (`0x4027fc-0x40285a`) and Fruit penalise a blocked rook; as with bishops, the definitions coincide.

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 (semi-)"open", but Rybka does.

3.1.4 Queens (`0x401c30-0x401d96` and `0x4021e0-0x402336`)

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 (`0x401db6-0x401e34` and `0x40233c-0x4023bd`)

Both Rybka and Fruit compute king safety by determining which pawns/pieces (ignoring kings) attack a square adjacent to the opponent's king. For each such attacker, a counter is incremented, and a score is added based on the attacker type. 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. The penalty is determined by a table-lookup depending on the number of attacking units, times the score of above, divided by a scaling factor.

3.2.1 King Shelter/Storm (`0x408b97-0x408f85`, `0x401e05`, `0x402394`)

Both Rybka and Fruit compute pawn shelter/storm for a king based upon three adjacent files (this idea seems older than Fruit, so I omit the details). Rybka uses a look-up table of patterns, while Fruit does bit-scanning. Both can reduce any penalties when castling rights exist. This is done via an averaging, and both seem to use the same method. E.g., let x be the shelter/storm value at `e1`, and $y = x$. If kingside castling rights exist, replace y by the min of y and the shelter/storm value at `g1`. Similarly for queenside. Then the penalty is $(x+y)/2$.

3.3 Pawn Evaluation (`0x408870-0x4089da` and `0x408a20-0x408b91`)

Both Rybka and Fruit consider (in order): doubled pawns, isolated pawns (on open/closed files), backward pawns (open/closed), detection of passed pawns, and candidate passed pawns. This is all fairly standard, though I don't know if this exact choice/ordering appeared before Fruit. The definition of "backward" is slightly different in Rybka, as is a minor variation with candidate pawns. In §6.2.2 I discuss the similarity of relative numerology in candidates/passers.

3.3.1 Passed Pawns (0x402410-0x40251f and 0x402570-0x402698)

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 (this uses SEE). Rybka splits up these three features in a piecemeal fashion with a bonus for each, 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/penalty depending upon the distance of each king to the square in front of the pawn. In Fruit these are 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 rank-based bonus in Rybka is essentially 10-30-60-100, though in units of 256 as in Fruit. 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 (0x4028c1-0x4028dc)

Both Rybka and Fruit interpolate "opening" and "endgame" values to get a final evaluation.[5] Fruit's is linear, while Rybka's is a bit more complicated.[6] After this interpolation, Rybka gives a final 3-centipawn bonus for being on move. At the top level, Rybka also has a "lazy eval" which is not found in Fruit.

4 Identical ordering of root search procedures

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

---

[5]Fruit uses a "phase" 1:2:4 for BN:R:Q, and re-scales the 24 initial units to 256. With Rybka we see only the material-table result, but the Fruit numbers match exactly upon scaling to 64.

[6]Rybka indexes a phase-based table 0-64 for this, but only 25 of these entries are ever used.

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.[7]

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. These occur directly in the Fruit 2.1 code, while in Rybka we only see the end result.[8] Furthermore, there is a re-scaling (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-occurrence; 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).[9]

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` squares). 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).

---

[7]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.

[8]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.

[9]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 both 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 `KnightLine` 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 schematic then looks like:

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

$\alpha_{18}x + \alpha_{ah}x + \beta_8 y - z$	$\alpha_{18}x + \alpha_{bg}x + \beta_8 y$	$\alpha_{18}x + \alpha_{cf}x + \beta_8 y$	$\alpha_{18}x + \alpha_{de}x + \beta_8 y$	$\cdots$
$\alpha_{27}x + \alpha_{ah}x + \beta_7 y$	$\alpha_{27}x + \alpha_{bg}x + \beta_7 y$	$\alpha_{27}x + \alpha_{cf}x + \beta_7 y$	$\alpha_{27}x + \alpha_{de}x + \beta_7 y$	$\cdots$
$\alpha_{36}x + \alpha_{ah}x + \beta_6 y$	$\alpha_{36}x + \alpha_{bg}x + \beta_6 y$	$\alpha_{36}x + \alpha_{cf}x + \beta_6 y$	$\alpha_{36}x + \alpha_{de}x + \beta_6 y$	$\cdots$
$\alpha_{45}x + \alpha_{ah}x + \beta_5 y$	$\alpha_{45}x + \alpha_{bg}x + \beta_5 y$	$\alpha_{45}x + \alpha_{cf}x + \beta_5 y$	$\alpha_{45}x + \alpha_{de}x + \beta_5 y$	$\cdots$
$\alpha_{45}x + \alpha_{ah}x + \beta_4 y$	$\alpha_{45}x + \alpha_{bg}x + \beta_4 y$	$\alpha_{45}x + \alpha_{cf}x + \beta_4 y$	$\alpha_{45}x + \alpha_{de}x + \beta_4 y$	$\cdots$
$\alpha_{36}x + \alpha_{ah}x + \beta_3 y$	$\alpha_{36}x + \alpha_{bg}x + \beta_3 y$	$\alpha_{36}x + \alpha_{cf}x + \beta_3 y$	$\alpha_{36}x + \alpha_{de}x + \beta_3 y$	$\cdots$
$\alpha_{27}x + \alpha_{ah}x + \beta_2 y$	$\alpha_{27}x + \alpha_{bg}x + \beta_2 y$	$\alpha_{27}x + \alpha_{cf}x + \beta_2 y$	$\alpha_{27}x + \alpha_{de}x + \beta_2 y$	$\cdots$
$\alpha_{18}x + \alpha_{ah}x + \beta_1 y$	$\alpha_{18}x + \alpha_{bg}x + \beta_1 y$	$\alpha_{18}x + \alpha_{cf}x + \beta_1 y$	$\alpha_{18}x + \alpha_{de}x + \beta_1 y$	$\cdots$

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 in various ways. It must be first be noted that, while the use of these two `KnightLine/KnightRank` 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.[10] My answer to that would be that there is no reason to keep this specific Rank/Line scaling, and in a fully independent implementation of the Fruit "idea" of PST, I would definitely expect there to be differences at some points.[11] 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 already have.

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, for which the value can be chosen as 0, 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 has $x = 5$ in the opening, with no endgame value (one can take $x = 0$). Rybka 1.0 Beta has $x = 181$ 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.

---

[10] 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.

[11] 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...

For reference, the White pawn values in Rybka 1.0 Beta 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/bonuses (`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 $x = 104$. Neither has an endgame value (so $x = 0$).
`static const int RookFile[8] = { -2, -1, +0, +1, +1, +0, -1, -2 };`
Again the principal query would be as to why *this* `RookFile` was 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 $(x, y) = (469, 0)$.

Both the schematic for 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 8 bytes of the 16-byte hash structure in Rybka 1.0 Beta and Fruit 2.1 are used in the same manner.[12] I can find no other engines that imitate this – even Fruit 1.0 differs (having a 64-bit lock). The common parts are:

a 32-bit lock, 2 bytes for the move, 1 byte for depth, then 1 byte for date.
To choose a random comparison, Faile orders these as [hash, depth, score, move] with differing bit widths. There is also an atypical commonality in the use of both lower/upper bounds in a PVS engine (though see Appendix B.3 below).

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 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 (essentially a quadratic fit), 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 the `quad()` function returns and those of the arrays included in Rybka 1.0 Beta are that the latter uses truncation rather than rounding.

Fruit 2.1 uses `quad()` to give a rank-based bonus for a passed or candidate pawn. For each such pawn, the `quad()` function is called and returns a score to be applied in the evaluation.
```
static const int PassedOpeningMin    = 10,  PassedOpeningMax    =  70;
static const int PassedEndgameMin    = 20,  PassedEndgameMax    = 140;
static const int CandidateOpeningMin =  5,  CandidateOpeningMax =  55;
static const int CandidateEndgameMin = 10,  CandidateEndgameMax = 110;
```

---

[12]Rybka 1.0 Beta has 4 hashing functions (64-bit): `0x40c2a0`, `0x40c3e0`, `0x40c520`, `0x40c640`.

As noted above, Fruit 2.1 uses a constant multiplier for attacker/defender distances to a passed pawn, while Rybka 1.0 Beta uses a `quad()`-like function dependent on the rank.

```
static const int AttackerDistance = 5;    // always 5   in Fruit 2.1
static const int DefenderDistance = 20;   // always 20  in Fruit 2.1
```

Fruit 2.1 also has 2 other constant bonuses (less important here, but listed for the sake of completeness), the first of which (`UnstoppablePasser`) is also a constant (25600) in Rybka 1.0 Beta, while for the second one, Rybka 1.0 Beta divides it up into more cases (`PassedUnblockedOwn`, `PassedUnblockedOpp`, `PassedFree`), each of which is given a `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 the `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, and further is split into 3 parts in Rybka.

6.2.2   Passed pawn numerology in Rybka 1.0 Beta

As noted above, Fruit 2.1 calls `quad()` every time, while one can note that the values are only dependent on the rank, and then use pre-computation 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 array-values in 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 };
```

In the 64-bit version, see the 56 bytes at `0x660f90` and the 16 bytes at `0x423b40`. 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 truncation) to produce the Rybka 1.0 Beta 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 (cf. the `PassedFree` array) that the values in Rybka 1.0 Beta were generated automatically, and not by hand. While a 10-30-60-100 parabolic scaling might reasonably be considered an "idea" to re-use, the coincidence of a hexapawn-variant of a `quad()`-like function makes the issue trickier to navigate.

There is also the question of whether the re-use of `quad()` is really all that important. The Rybka choice of values for `PassedOpening` (and others) is not too similar to that of Fruit, and the splitting of the bonuses for a passed pawn makes the Rybka method differ from that of Fruit in any event. As with the PST comparison, it seems that there is a *structural* similarity between Fruit and Rybka, and the question of "originality" therein allows multiple approaches.

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

6.3   UCI parsing

The final topic 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 with respect to code copying.

In this section, I use two "decompilations" from other people. Part of this is to show that others who have studied the issue have come to a similar conclusion. Such decompilations can be tendentious, especially if one chooses to use Fruit-like variable names when re-casting into a higher-level language.

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 1.0 Beta code (below) will show a similar hack.

Here is the stripped-down Fruit code from `parse_position()` in `protocol.cpp`:

```
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;     // aside: the "+6" here is a useful ASM locator
       while (*ptr != '\0') // until string is terminated
       {  [...]             // code to get the move_string, advancing ptr
          move = move_from_string(move_string,SearchInput->board);
          move_do(SearchInput->board,move,undo);
          while (*ptr == ' ') ptr++; // eliminates spaces
       }
    }
```

A 32-bit Rybka 1.0 Beta `decompilation` by Franklin Titus, with my comments:[13]

```
int __usercall sub_4092E0<eax>(const char *a1<eax>)
{ char *v1; // esi@1
  const char *v2; // esi@1
  char *v3; // edi@1
  int v4, v5; // esi@6, 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?
    sub_403490();             // board_from_fen(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);    // some code to get the move from the string
      sub_40ABC0(v5);         // move_do_UCI(move) -- in search, prom bits differ
      v4 += 5;                // Fruit uses *ptr++ for each character
      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, needed with proms
    }
  }
  return sub_401100();    // clear various arrays, general bookkeeping
}
```

Some of this formulaic, and the most interesting part is likely that `moves[-1]=0`
reappears in the Rybka 1.0 Beta code; in the Fruit 2.1 code it ensures the FEN
string proper is `NUL`-terminated, but this is not strictly necessary. The Rybka
version could, however, be a compiler optimisation of `fen[moves-fen-1]=0`,
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.

---

[13]There are a few minor errors, but I stick close to the original. My 64-bit version is `here`.

13

6.3.2 Time management

In this section, I largely refer to Rick Fadden's 32-bit Rybka 1.0 Beta disassembly efforts[14] at `http://www.talkchess.com/forum/viewtopic.php?p=187290`.

The first item to mention is that it already seems not completely natural to place the time management code at the end of the "go" parser as done in both Fruit 2.1 and Rybka 1.0 Beta. Both contain "white/black" selection code as a separator between the "go" parsing proper and the time management code, for Rybka 1.0 Beta as

```
if ((Board->turn) == 0) { time = wtime; inc = winc; } // Fadden's Rybka
else                    { time = btime; inc = binc; } // decompilation
```

compared to Fruit's

```
if (COLOUR_IS_WHITE(SearchInput->board->turn)) {
   time = wtime;
   inc = winc;
} else {
   time = btime;
   inc = binc;
} // Fruit code
```

The subsequent lines in Fadden's Rybka 1.0 Beta decompilation have:

```
// 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;
  }
```

As noted by Zach Wegner and others, the comparison with a floating-point value by Rybka 1.0 Beta 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.

---

[14] I find Fadden's decompilation to be very precise. For instance, he keeps all the variables (such as `btime/wtime/binc/winc`) in the same order as in the ASM code. There is an alternative disassembly available (`link`) which is lacking in such necessities, and even goes so far as to re-arrange the parsing of the `wtime/winc/btime/binc` elements in Rybka to match the alphabetical order of Fruit.

Here is the Fruit code for this (see `parse_go()` in `protocol.cpp`):

```
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;
}
```

The multiplication of `movetime` by 5 to get `time_limit_1` is another common element, while Zach Wegner has noted that the multiplication by 1000 with `time_limit_2` actually seems like a bug that prevents Rybka 1.0 Beta from properly handling a UCI command like "`go movetime 60000`" in certain cases.

There are a few other atypical similarities in the respective UCI parsing routines (e.g., incremental use of `strtok`), but I won't discuss them here, and just leave Fadden's disassembly/decompilation to suffice for a general sense.

However, as a final specific similarity for this section, I will mention the concluding block of code of this "go" parser. There is a significant overlap in how Fruit 2.1 and Rybka 1.0 Beta handle the difficulty of "delaying" sending a best move when in "infinite" or "ponder" mode.[15] Here is the Fruit 2.1 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();
```

---

[15]Rybka 1.0 Beta has no `Ponder` UCI option *per se*, but does send a `ponder` token when reporting `bestmove`, and also correctly parses `go` strings which include a `ponder` token. Indeed, the UCI protocol notes that the `Ponder` option is largely a way for the GUI to notify the engine about pondering ahead of time, so that the engine can modify its time management if desired.

Here is the comparative Rybka code I obtained from a 64-bit disassembly.[16]

```
0x40702e: test    %r15b,%r15b         # r15 is "infinite"
0x40704d: jne     0x407054
0x40704f: test    %r13b,%r13b         # r13 is "ponder"
0x407052: je      0x407063* [0x40705b]  # if either is true
0x407054: movb    $0x1,0x2652d1(%rip)   # 0x66c32c SearchInput->Infinite = true
0x407063* movb    $0x1,0x262677(%rip)  # 0x6696e1, set Searching = true
0x40705b: test    %r15b,%r15b         # check "infinite" again
0x40706a: jne     0x40707a
0x40706c: test    %r13b,%r13b         # check "ponder" again
0x40706f: jne     0x40707a            # if both false,
0x407071: mov     %r13b,0x26266a(%rip)  # 0x6696e2, set Infinite = false
0x407078: jmp     0x407081            # else
0x40707a: movb    $0x1,0x262661(%rip)   # 0x6696e2, set Infinite = true
0x407081: movb    $0x0,0x26265b(%rip)  # 0x6696e3  set Delay = false
0x407088: callq   0x408f90            # call the search function
0x40708d: movzbl  0x26264e(%rip),%eax  # 0x6696e2  load Infinite variable
0x40709b: movb    $0x0,0x26263f(%rip)  # 0x6696e1  set Searching = false
0x4070a2: mov     %al,0x26263b(%rip)   # 0x6696e3  set Delay = Infinite
0x407094* test    %al,%al             # if Infinite is true
0x4070a8: jne     0x4070af            #  then don't
0x4070aa: callq   0x406aa0            # call send_best_move()
```
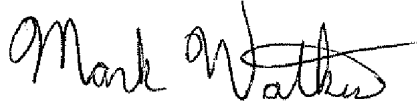
Note in particular that (`infinite || ponder`) gets computed twice in both. Also, the operations with `Searching`, `Infinite`, and `Delay` are ordered the same (as are their variable allocations). Finally, setting `Delay` to be "false" can be seen as redundant in Fruit, as three lines above it was `ASSERT`ed to be so.

7   Conclusion

This document has highlighted a number of places where Rybka 1.0 Beta can be said to have over-stretched an "originality" barrier with respect to Fruit 2.1. These include a borrowing of arrays in PST, a peculiar match in data structures for hashing, a wholesale re-use of evaluation features, and a repetition of the same ordering of operations in root search. There is furthermore a re-appearance of a 26-77-154-256 scaling rather than the more natural 10-30-60-100, and some specific examples of "code copying" from the UCI parsing.

The first appendix below gives another example of "code copying" (from iterative deepening at the end of root search), while the second appendix tries to determine whether later Rybka versions have remedied the above problems, and draws together some related issues.

Mark Watkins, mwatkins@e-mile.co.uk

---

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

A Root search analysis: iterative deepening

This appendix is a Fruit/Rybka comparison for 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 2.1 code, then the Rybka 1.0 Beta disassembly, and finally
a `C++` translation of the latter. The code is the iterative deepening at the end
of root search. The Fruit 2.1 code is at the end of `search()` in `search.cpp`.

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 a commented disassembly from the Rybka 1.0 Beta 64-bit version:

```
0x4095a5: mov    $0x1,%esi            # %esi will be equal to 1 throughout
0x4095aa: mov    %esi,%ebx
0x4095b0: cmp    $0x5,%ebx            # compare depth to 5
0x4095b3: jb     0x4095c4            # if at least 5
0x4095b5: lea    -0x2(%rbx),%edx          then subtract 2 before...
0x4095b8: lea    0x25af79(%rip),%rcx  # 0x664538 ["info depth" string]
0x4095bf: callq  0x40d0b0            # ...printing the "info depth" string
0x4095c4: mov    %ebx,%ecx            # copy depth to %ecx reg for func call
0x4095c6: movb   $0x0,0x262e81(%rip)  # 0x66c44e set "change" to false
0x4095cd: movb   $0x0,0x262e78(%rip)  # 0x66c44c set "bad_1" to false
0x4095d4: callq  0x40ba70            # call search_full_root(ecx) [ecx=depth]
0x4095d9: callq  0x4070c0            # some sort of update function
0x4095de: mov    0x26706f(%rip),%r11d # 0x670654, get score
0x4095e5: cmp    $0xffff8300,%r11d    # fiddle around
0x4095ec: jle    0x4095fe            # ...
0x4095ee: cmp    $0x7d00,%r11d        # with mate scores
0x4095f5: movzbl 0x262e54(%rip),%edx  # 0x66c450 load "flag"
0x4095fc: jl     0x409601            # if mate score,
0x4095fe: mov    %sil,%dl               # set "flag" to true (esi is always 1)
0x409601: cmp    %esi,%ebx            # compare depth (%ebx) to 1
0x409603: jne    0x409631            # if depth == 1
0x409605: cmpl   $0x0,0x267058(%rip)   # 0x670664 think this is RML[1]
0x40960c: je     0x40962f               # if only one legal move, skip next
0x40960e: movzbl 0x262e3a(%rip),%ecx   # 0x66c44f "easy"
0x409615: mov    0x267449(%rip),%eax    # 0x670a64 (value of move 1)
0x40961b: add    $0x96,%eax             # EasyThreshold of 150 [as in Fruit]
0x409620: cmp    %eax,0x26743a(%rip)    # 0x670a60 (value of move 0)
0x409626: cmovae %esi,%ecx             # if move values differ by enough
0x409629: mov    %cl,0x262e20(%rip)       # 0x66c44f  set "easy" as true
0x40962f: cmp    %esi,%ebx            # if depth > 1
0x409631: jbe    0x40964d* [0x409647]
0x409633: movzbl 0x262e12(%rip),%eax    # 0x66c44c  load old bad_1
0x40963a: movb   $0x0,0x262e0b(%rip)    # 0x66c44c  bad_1 = false
0x409641: mov    %al,0x262e06(%rip)     # 0x66c44d  bad_2 = (previous) bad_1
0x40964d* movzbl %dl,%eax            # %dl: 1@4095fe (mate), "flag"@4095f5
0x409650* mov    %r11d,0x262df1(%rip) # 0x66c448  last_value = score
0x409647: cmp    0x262cdb(%rip),%ebx  # 0x66c328  see if depth>=depth_limit
0x409657: cmovae %esi,%eax             # if depth >= depth_limit
0x40965a: mov    %al,0x262df0(%rip)     # 0x66c450  then "flag" is true
0x409666: mov    0x262cb3(%rip),%r8d  # 0x66c320 load SearchInput->time_limit_1
0x40966d: movzbl 0x262dd8(%rip),%r9d  # 0x66c44d  load bad_2
0x409660* callq  *0x139ca(%rip)       # 0x41d030  GetTickCount -> %eax
0x409675: mov    %eax,%r11d
0x40967c* sub    0x262db5(%rip),%r11d # 0x66c438  (subtract StartTime)
```

```
0x409678: lea    (%r8,%r8,1),%ecx     # compute 3 * time_limit_1
0x409683: mov    $0xaaaaaaab,%eax     #  then mult by 2/3
0x409688: mul    %ecx                 #   (result goes in edx with mul here)
0x40968a: shr    %edx                 #  and div by 2 ... hmm = time_limit_1 ?
0x40968c: cmp    %edx,%r11d           # compare to time taken
0x40968f: jb     0x4096a6             # if small, ignore next
0x409691: movzbl 0x262db8(%rip),%ecx   # 0x66c450 "flag"
0x409698: test   %r9b,%r9b             # if "bad_2" is false
0x40969b: cmove  %esi,%ecx               # ecx = 1 (esi is always 1)
0x40969e: mov    %cl,0x262dac(%rip)    # 0x66c450 store ecx in "flag"
0x4096a4: jmp    0x4096ac
0x4096a6: mov    0x262da4(%rip),%cl    # 0x66c450 (reload "flag")
0x4096ac: mov    $0xaaaaaaab,%eax
0x4096b1: mul    %r8d                 # mult time_limit_1 by 2/3
0x4096b4: shr    $0x2,%edx            #  and div by 4
0x4096b7: cmp    %edx,%r11d           # compare to time taken
0x4096ba: jb     0x4096d1             # if small, ignore next
0x4096bc: cmpb   $0x0,0x262d8c(%rip)   # 0x66c44f see if "easy"
0x4096c3: movzbl %cl,%eax             # if not "easy", then eax is "flag"
0x4096c6: cmovne %esi,%eax            # if it is "easy", then eax is true
0x4096c9: mov    %al,%cl
0x4096cb: mov    %al,0x262d7f(%rip)    # 0x66c450 store eax in "flag"
0x4096d1: shr    %r8d                 # time_limit_1 divided by 2
0x4096d4: cmp    %r8d,%r11d           # compare to time taken
0x4096d7: jb     0x4096f3             # if small, ignore next
0x4096d9: test   %r9b,%r9b             # if "bad_2" is true
0x4096dc: jne    0x4096f3             # then ignore next
0x4096de: cmp    %r9b,0x262d69(%rip)    # 0x66c44e "change", see if false
0x4096e5: movzbl %cl,%eax               # if not, then eax is "flag"
0x4096e8: cmove  %esi,%eax               # if "change" is false, eax is true
0x4096eb: mov    %al,%cl
0x4096ed: mov    %al,0x262d5d(%rip)     # 0x66c450 store eax in "flag"
0x4096f3: cmpb   $0x0,0x262d36(%rip)  # 0x66c430 see if "stop" is true
0x4096fa: jne    0x409714             # if so, then exit this function
0x4096fc: test   %cl,%cl              # see if "flag" is true
0x4096fe: je     0x409709             # if so
0x409700: cmpb   $0x0,0x262c25(%rip)   # 0x66c32c and SearchInput->infinite
0x409707: je     0x409714               # is false, then exit this function
0x409709: add    %esi,%ebx            # increment depth (%esi is 1)
0x40970b: cmp    $0x48,%ebx           # if depth < 72
0x40970e: 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 high-level 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 -- might be the compiler
  search_full_root(depth); // 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.[17]

It can also be noted that the 6 variables in Rybka 1.0 Beta here are allocated in exactly the same order as in the comparative Fruit 2.1 code (see `search.h`):

```
struct search_root_t {
[...]               // Rybka location
   int last_value;  // 0x66c448
   bool bad_1;      // 0x66c44c
   bool bad_2;      // 0x66c44d
   bool change;     // 0x66c44e
   bool easy;       // 0x66c44f
   bool flag;       // 0x66c450
};
```

---

[17]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. At a higher level, an alternative implementation is: near the top, `break` when "`stop`" is true and `continue` when "`infinite`" is; then "`flag`" is unneeded, as `break` can instead be used when the various conditions are true.

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

Among the more well-known differences of Rybka 1.0 Beta are the use of bitboards (rather than a mailbox representation) and the extensive material imbalance table. Rybka 1.0 Beta also does not have the history-based reductions made prominent by 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).

Except for the issues listed in §4 above, the search routines of Rybka 1.0 Beta and Fruit 2.1 do not seem to me to be more similar than I would expect for two PVS engines. Rybka 1.0 Beta seems not so mature, e.g., simply announcing mate and not trying to minimise the distance to it. The principal "new idea" in Rybka 1.0 Beta 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.[18] The re-writing of the evaluation function in Rybka 3 (by Larry Kaufman) has likely removed much of any complaint here.

I have not bothered to see when/if the UCI parsing was changed, though a crude check[19] finds code similarities in Rybka 2.3.2a (June 2007). The use of "0.0" in time management is also still extant in Rybka 2.3.2a.[20] The hash structure looks to be the same in Rybka 2.3 as in Rybka 1.0 Beta (the first 8 bytes as in Section 6.1), but was changed to a 64-bit format[21] in Rybka 2.3.1, and again modified slightly in Rybka 2.3.2a.

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

---

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

[19]I searched for adding 6 (see `ptr = moves + 6`), and looked at surrounding code segments. The desired instruction appears at `0x550bee` in Rybka 2.3.2a (single-cpu 64-bit version), with a nearby `lea` usage having offset of `-0x188` as with Rybka 1.0 Beta. Similarly at `0x4c606b` in Rybka 3, where nearby at `0x4c6031` one finds the "`moves[-1]=0`" instruction.

[20]I searched for `cvtsi2sd` with a 32-bit input. It appears at `0x551ef5` in the 64-bit version of Rybka 2.3.2a. This instruction appears multiple times in Rybka 3, and it seems to me that the one that corresponds to this is at `0x4c8759`.

[21]Rajlich mentioned (`link`) that there was some "significant" problem with hashing in Rybka 2.3 (no word about previous versions), which might be that 128-bit hash entries were not modified atomically, so as to cause occasional problems in SMP mode.

iterations of this in all, with Rybka 1.1 being released on March 16, 2006). The use of `setjmp` in a 64-bit compilation can be tracked fairly well via the `fnstcw` instruction in an ASM dump (especially with a relative address involving `0x5c`), and then by back-tracking the calling function (possibly in a debugger), one can attempt to identify the root search. Even in Rybka 2.3.2a a few similarities with the Rybka 1.0 Beta code seem to remain, but overall a comparison as in Table 1 would not be so conclusive. Some of the changes already appear in Rybka 1.01 (the updating of the hash date/depth table and the resetting of killers/history are moved to near the beginning, for instance). Similarly, the iterative deepening code[22] has been progressively modified (e.g., to consider mate scores), but various Fruit remnants seem detectable if such parentage is suspected, particularly with the `flag`-based implementation.

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 from April 2004 called Rybka 1.5.32 participated in Chess War V (Olivier Deville) and Le Système du Suisse Saison n°3 (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. Rybka 1.5.32 also played two rook underpromotions in approximately 60 games, whereas Rybka 1.0 Beta was (somewhat curiously) 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.[23] It remains unclear, however, why underpromotions would disappear, or why the UCI parsing and time management would follow the Fruit template so closely.

B.3.1 Node counting

One of the more wearying parts of the Rybka 1.0 Beta saga has been the "obfuscation" of nodes and depth. This has developed into a long and convoluted subject, and so I try to be brief here. In particular, while many were annoyed to find out that Rybka was actually searching deeper and faster than other engines (when the NPS/depth numbers told the opposite story), this has no direct bearing on the Fruit issue.

For nodes, responding to Anthony Cozzie's inquiry, Rajlich said (`link`): "Actually, if you go in a debugger, you can trivially see that two quantities are being combined. One I call "gulp", this is for me the interesting figure (for my private tests). The second is a simple ticker for the next I/O check." [Feb 2006]

---

[22]This seems to be located from `0x4822a7-0x4826a0` in single-cpu 64-bit Rybka 3.

[23]One can note that both Fruit 2.1 and Rybka 1.0 Beta 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 Rybka 1.0 Beta inherited this from earlier `MTD(f)` days.

However, previous statements of his seem to indicate that he was using a more typical notion of a node in earlier Rybka versions. For instance, in `post #356880` (March 27, 2004) he reports that "Rybka 1.3" gained about 20% from a 64-bit compile, from 1.2Mn/s to 1.5Mn/s. Four days previously, in a `thread` musing about Shredder's search, he agreed that depth could be tricky to compute with reductions/pruning in the mix, but: "There aren't too many ways to calculate NPS, though, this is real info IMO." Finally, later Rybkas determine "nodes" by counting White `make_move()` calls, and dividing by 7. [For depth "obfuscation", early Rybkas subtracted 2, and later versions subtract 3].

B.4   Statements of Rajlich

This is perhaps not the best place to dredge up more statements of Rajlich, but I list here a few 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*).

First there is Rajlich's well-known statements from an `interview` with Alexander Schmidt and others (see #20-21, this is from Dec 2005):

```
[...] I went through the Fruit 2.1 source code forwards and backwards and
took many things. [...] Anyway, if I really had to give a number - my wild
guess is that Rybka would be 20 rating points weaker had Fruit not appeared.
```

The next (`link`) is in reply to Daniel Mehrmann, whose tests had suggested that the mobility and PST in Rybka 1.0 Beta might derive from Fruit 2.1 (Mehrmann later apologetically retracted this).

```
Subject: 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 (`link`) 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: 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. [...]
```