1 A comparison of Rybka 1.0 Beta and Fruit 2.1

This document is designed to 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 January 26, 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 programme in question may be borrowed from a competitor. In some sense then, this guide 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.[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% 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 same type of PST-scheme re-using the File/Rank/Line weighting;
- the use of *exactly* the same evaluation features;
- the ordering of operations at the root node in the search.

[1] Just to recapitulate, this means that I am not going to consider 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-apperance 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 otherwise integer-based time management code of Rybka.

Some of these could be considered to be "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 what is "fair game" to re-use from an open source programme is not entirely clear. For instance, pursuant to the second major point of above, one of the more notable "ideas" of Fruit was its evaluation based mainly on mobility. In an appendix, 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 Common structure of PST computations

The first major point is that of Piece-Square-Table (PST) computations, also known as "static" values. This occurs directly in the code in Fruit 2.1, while in Rybka 1.0 Beta, we only see the end result.[2] 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).[3]

[2] 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.

[3] 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.

3.1 The example of the knights

I give one example in fuller detail. For various reasons, the (white) knights are the best, as there are two components (file and rank), and there are is only one minor variation (the `a8/h8` square).

Here are the raw PST white knights values for Fruit 2.1 and Rybka 1.0 Beta in the opening, the latter on the bottom.

Table 1: Fruit 2.1 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

Table 2: Rybka 1.0 Beta White Knight Opening PST values

-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

The co-incidence of these can be seen when we make a formulaic representation (omitting the left-right symmetry).

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 above array 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

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

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.

3.1.1 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.[4] 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.[5] 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.

3.2 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.

[4] 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.

[5] 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...

3.2.1 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 programmes have some sort of dependence on the rank in Pawn PST.

3.2.2 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 };
```

3.2.3 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).

3.2.4 Rooks PST

Table 8: Common PST schematic for rooks (opening)

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

3.2.5 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 };
```

3.2.6 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 };
```

4 Commonality of evaluation features

Most of the work here has been done by Zach Wegner. 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 without much comment, as in almost all cases, the functionality is the same.

4.1 Piece evaluation

4.1.1 Bishops

Both Rybka and Fruit consider only mobility as the primary evaluation component with bishops. 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).

4.1.2 Knights

Both Rybka and Fruit consider only mobility as the primary evaluation component with knights.

4.1.3 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 pawns 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 exactly 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.

4.1.4 Queens

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

4.2 King Safety

Both Rybka and Fruit compute king safety by computing whether a piece (ignoring pawns and 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 1 for minors, 2 for a rook, and 4 for a queen. In Rybka these are 941, 418, 666, and 532.

4.2.1 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 bit-scanning. Fruit has a consideration with castling potential, while Rybka does not. There are likely as many differences as similarities here.

4.3 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.

4.3.1 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 SEE is not specifically used).

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

The similarity of relative numerology for passed pawns is discussed more below, but here I mention that the relative scaling for each is essentially 10-30-60-100, though done in units of 256. In some sense, this is the most trenchant piece of evidence with passed pawns, as the other components either have slight variations in Rybka or are not "original" in Fruit.

4.4 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.

5 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`.

Table 11: 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 and their 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).

The 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 plagiariam is generally an *ethical* standard and not a legal one). There is also the question of how important this item is.

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 depth.

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

```
   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;}
```

This is the `quad()` function in Fruit 2.1, with ASSERTs stripped out, and the Bonus values above made explicit, with my comment about percentage of 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 just trunctating in division by 256. The only difference between the values that `quad()` returns and those of the arrays including 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.

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 arrays 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 as a side-point, the disassembly of the code itself (from Rick Fadden) already shows the obfuscation of depth in Rybka via subtracting 2.

6.3.1 Parsing the "position" string

A first hint of copying is apparent 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.

```
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, compiler would not do this on 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.

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, 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 "time_limit_1" by 5 is another common element. Further similarities could be mentioned, but it not so clear how important they really are.