Review of the Rybka case

Version of September 2, 2013.

This is a brief recapitulation of the underlying elements of the evidence in the Rybka case considered by the ICGA.

1 The ICGA process

The International Computer Games Association was asked by Fabien Letouzey (the author of Fruit), as joined by an open letter of 15 other programmers, to investigate the "origins" of Vasik Rajlich's entries into its tournaments, and whether in particular said entries misappropriated Letouzey's work.

The ICGA responded by convening a Panel of computer chess experts, to consider particularly the specific Rybka/Fruit question, but not limited therein. In fact, during the investigation of Rybka, a prior version (privately circulated, though presumably a similar engine participated in CCT 6 in Jan 2004) was analyzed for its re-use of Crafty code. The Panel was advisory in nature, with any decisions being taken by the Board (see below for the membership and credentials of the Board). On a broader level, said Panel has also advised the ICGA on how to deal with the recurrent problem of "clones" in computer games.

One specific purpose of the Panel with Rybka was to provide the ICGA Board with an advisory opinion regarding the (voluminous) technical evidence. The discussions of said Panel were done on a private wiki. Approximately 34 persons were admitted to this, some of whom were merely observers. As the Panel was advisory and investigatory in nature, persons with close connections to competitors (and indeed, direct principals of some competitors of Rybka) were also allowed. This was in part due to the fact that the world of computer chess is quite small, and persons with adequate technical knowledge have often been associated with one chess program or another over the years. Vasik Rajlich was also invited to join the Panel discussions if he so chose, though he was also availed the choice of offering a defense directly to the ICGA Board. He opted for the latter (though in the end, did not present a defense even to them).

1.1 Panel process

The Panel was faced with various evidence concerning Rybka, and how to analyze and comprehend it. Rather quickly, it became clear that the "evaluation function" similarity (at the feature level) was a significant item that most everyone could agree would transgress the ICGA "originality" rule if the evidence was indeed as strong as contended. This thus formed the major focus of discussions.

Due to the differences in underlying data structures of various chess programs, it was apparent that some sort of abstraction method must be used in the comparison of evaluation functions (as literal code would differ). At the same time, general chess knowledge is sufficiently refined that almost every program uses the same generic concepts to some extent. The panel thus opted to base its analysis on a variant of the Abstraction-Filtration-Comparison Test. This is discussed in more detail below.

After about 2 months of discussion and presentation of evidence, an internal "vote" was taken. The two questions asked of Panel members were: whether or not they had read all the evidence; and whether of not they thought Rajlich had breached ICGA Rule #2 on originality. The number of such "votes" counted on the wiki was 14,[1] and all agreed that the originality rule had been violated.[2] The Secretariat of the Panel prepared a report, and the ICGA Board voted 5-0 in their verdict, which included a lifetime ban. This seems to be in part due to the gravity of the offense, and in part due to that Rajlich offered no defense, and indeed practically ignored the ICGA President.[3]

1.2 Board members

The ICGA Board (elected on a triennial basis by the membership) consisted of:

- the President David N. L. Levy, co-author of Moby, Cyrus, and other chess programs, who has an extensive history (over 40 years) with computer chess, for instance overseeing a number of previous ICGA investigations into improper re-use of another author's work;
- the Vice-President Yngvi Björnsson, who is Associate Professor (in computer science) at Reykjavik University, and has written a world champion program (YL) in a related game of Lines of Action, and also written a chess engine (The Turk);
- the Secretary-Treasurer Hiroyuki Iida who has written a 4-time world champion Shogi program (he is also a 6-dan professional player) and is a professor at the Japan Advanced Institute of Science and Technology, where he is head of the games laboratory;
- the Programmer's Representative Rémi Coulom who is an Associate Professor at the Université Charles de Gaulle (Lille) and has written a chess playing program (The Crazy Bishop) and more recently a program that plays Go;
- and the long-standing Tournament Director H. J. (Jaap) van den Herik, who has overseen many prior disputes, has a speciality in some aspects of computer law in his professorate at Tilburg University, has been involved with the development of at least 3 computer chess programs (Pion, Dutch, and Much), and has co-authored many papers about the solving of various games over the decades, not to mention being a co-founder of the Dutch computer chess organization (CSVN) in 1980.

None of these is in anyway affiliated to Rybka or any of its competitors.

[1] Some private votes were communicated to the Panel Secretariat. With at most 1 or 2 exceptions, all voters appear to possess suitable technical expertise to comprehend the evidence.

[2] At most 3 of these could said to have had a direct conflict of interest (such as gaining a Championship due to the disqualification). As noted above, the computer chess fraternity is such that it is almost impossible to remove factions entirely. A sample of three "independent" members of the Panel (including myself) is given below.

[3] The slim elements of defense proferred by Rajlich are noted below.

1.3 The verdict

As noted above, the ICGA Board was responsible for judging the evidence and giving a sentence. They chose to disqualify all of Rajlich's entries (even those post-dating the claims with Fruit), on the basis of violating their rule regarding originality: *Each program must be the original work of the entering developers.* They also imposed a lifetime ban against him from entering their tournaments.[4]

The stringency of this verdict seems in part due to the fact that Rajlich made no response to the allegations, and indeed, seems not to have bothered to even have looked at them. Also, since authorship is such a fundamental aspect to the purpose of the ICGA (which was formed by the programmers themselves back in the 70s), the offenses of Rajlich (misappropriation of work) were seen to be about the worst possible, and indeed were in line with prior precedent.[5]

1.4 Some Panel members

The most eminent member of the Panel was Ken Thompson, who has won the highest award of the Association for Computing Machinery in part for his development of the UNIX operating system. He also won the World Computer Chess Championship in 1980 with his program BELLE, later adapted into the Deep Blue program that defeated Kasparov. He has not actively competed in computer chess since the mid-80s, but retains an avid interest in the field.[6]

Another member (quite active) of the Panel was Wylie Garvin, who is a general computer games programmer/maintainer with Ubisoft (in Montreal), who in part suggested the Abstraction-Filtration-Comparison methodology as applied to the ICGA criteria. He has no links to either Rybka or its competitors.

I myself (Mark Watkins) am a professional mathematician working in software development with the MAGMA Computer Algebra Group at the University of Sydney. My interest in computer chess programming has been sporadic over a span of more than 15 years. My competence in enumerating technical evidence is largely due to the fact that (akin to Garvin) I often have to debug code written by others. Again I have no links to either Rybka or its competitors.

Other notable Panel members included Bob Hyatt (the author of Crafty and Cray Blitz), Mark Lefler (author of Zillions of Games, and the chess program Now), Zach Wegner (author of ZCT, current developer of Rondo [*né* Zappa], and other than myself one of the main technical investigators), and Gerd Isenberg (author of IsiChess, and principal editor of Chess Programming Wiki).

[4] Some of the Panel members had suggested more lenient measures, such as changing the relevant WCCC winners to say "Fruit/Rybka" and listing Letouzey as a co-author. On the other hand, all those who expressed an opinion agreed that Rajlich's future participation should be contingent upon the issues raised here being dealt with in a suitable manner.

[5] In this regard, see this link, where the KCC Paduk entry (into the Go competition) was rejected for "past problems with [this] program in other computer Go tournaments" that had occurred a decade previous, and might well have been remedied by that point. In contrast, with the El Chinito case the author admitted his guilt, and was almost immediately accepted back into the programmers' fraternity, with little if any penalty imposed.

[6] It is not completely clear to me, but it seems that one reason that Thompson was encouraged to join the Panel was to ensure the futility of questioning its competence or independence.

2 Evidence

This section briefly describes what the evidence is for various Rybka versions. Later sections expand upon this for some of these.

2.1 Evidence with Rybka 1.6.1

Versions of Rybka from 2004 were obtained for the ICGA Panel via Olivier Deville, to whom Rajlich had sent them (among other places) for participation in Deville's ChessWar tournaments. The numbering can be confusing, as the 1.6.1 version described here is *earlier* than the 1.0 Beta of Rybka that was publicly released in December 2005.

The primary Rybka version from 2004 considered (by Wegner and Watkins, and also Hyatt to some extent) was Rybka 1.6.1. This was found (see below) to have significant usage of elements from Crafty. The most notable were matching `EvaluateWinner()` code and `NextMove()` mechanics.

Other commonalities included the checking of a non-possible 99999 from a `EvaluateMate()` call (similar to the El Chinito case), and a repeated zeroing in array initialization (due to a typo in the Crafty code). There is also the re-use of *en passant* avoidance code with obsolete Edwards tablebases, piece numbering, and bit-packing of moves. This is not necessarily a complete list. Some of these latter are not all that significant themselves, but become more relevant when one considers that Rybka 1.0 Beta dispenses with such Crafty-like internals, and prefers to use Fruit-like ones.

2.2 Evidence with Rybka 1.0 Beta

The analysis of evidence for Rybka 1.0 Beta comes from a variety of sources. The most notable (other than my own) were Rick Fadden's analysis from early 2008 (which was more in the context of Rybka/Strelka), Zach Wegner's analysis from mid-2008 and later, and various bits from assorted other persons (such as Franklin Titus).

The Panel did not consider the "Strelka 2.0" source code, a program produced by Yuri Osipov which Rajlich has claimed is derivative of Rybka 1.0 Beta (indeed, he claims it to be substantially similar, it seems).

2.2.1 Probative similarity

The initial elements of evidence considered are "probative", in that they point to a strong likelihood of fairly direct code copying. These include mechanisms to control the "search" of the engine, identical structures and ordering of idiosyncratic elements of Fruit 2.1, and re-appearance of redundant code (see §4.1).

The fact that Rybka 1.0 Beta differs drastically from Rybka 1.6.1 in the above elements is another facet.

The fact that Rybka 1.0 Beta "obfuscated" its node count and depth (1.6.1 does not) was not found to be of great import, but is another probative element.

The fact that Daniel Mehrmann (acting as an "outside observer", that is, an end-user) was able to suggest[7] a week after Rybka 1.0 Beta appeared (see this link) that its mobility and PST were based on Fruit 2.1 is another issue of mention here, though again the Panel did not directly discuss this.

2.2.2 Substantial similarity

The principal (though not only) element considered by the Panel for "substantive" purposes was comparing the evaluation functions of Fruit 2.1 and Rybka 1.0 Beta.[8] These were done at the "feature" level, following a procedure similar to the Abstraction-Filtration-Comparison Test.

The Panel first determined (by consensus) as an abstract proposition that the choice of what features to use, and their implementation characteristics, was indeed something subject to "protection" under the ICGA originality requirement. Furthermore, it was then found (via analysis and discussion of the evidence) that the Rybka 1.0 Beta selection of what features to use (and how to implement them in some cases) was overall substantially similar to those of Fruit 2.1. This was also found to be the case (to a slightly lesser extent) with Rybka 2.3.2a. The Board concurred with the Panel's opinion. The relative import of "features" versus their quantitative valuation ("tuning numbers") was not discussed, as the feature overlap already sufficed to breach ICGA rules.

This comparison procedure (called `EVAL_COMP`) has come under some criticism, some of which will be discussed more below (see §4.2).

The Panel concluded from `EVAL_COMP`, in part when viewed alongside other evidence, that it was quite clear that more than just "ideas" from Fruit 2.1 re-appeared in Rybka 1.0 Beta, but rather quite specific creative choices. Any individual element could be declared to be simply "Fruit influence", but the picture as a whole stretched much beyond that.[9] There was a general consensus that the Fruit/Rybka situation was much beyond the "standard" amount of inter-engine influence that was typically permitted in author-based computer chess competitions.[10] The ICGA Board agreed with this in their verdict.

Another consideration was that the PST tables of Rybka 1.0 Beta could be computed exactly via an abnormally small amount of code modification from the corresponding Fruit 2.1 code.[11]

---

[7] He made this determination from a test suite of chess positions that he analyzed with Fruit 2.1 and Rybka 1.0 Beta. He later, in part due to Rajlich's assurance that Rybka was original, apologized for raising the issue. Now, however, it appears that he has been vindicated.

[8] Depending on one's predilection and the specific case, I would say that the evaluation function can form anywhere from 25-75% of the "value" of an engine.

[9] It can be noted that some Panel members were happy with the "qualitative" statement of Fruit/Rybka overlap as phrased here, but others desired something more quantitative, which eventually led to `EVAL_COMP` being instrumented.

[10] Historical examples can be given here, such as Hsu/Berliner regarding Deep Thought being asked to remove (or re-write) its "Cray Blitz simulator" taken from the HITECH project.

[11] It has been suggested that PST tables have don't really have a lot of "chess knowledge" embodied in them, which is one reason why the "code differential" metric of above was used to phrase this evidence, rather than previous ones based on "templates" or code reconstruction. The question of to how much "value" one should place on *merely* the PST evidence has also been raised, though, like many other aspects, it provides another piece of the picture.

A third element is the re-use of the Fruit method to decide when to start another iteration in the search (see §4.3), that is, criteria for deciding when to make a move.

A few more assorted examples could be given, any of which would be unremarkable by itself, but show a general pattern. The dramatic difference between Rybka 1.6 and Rybka 1.0 is also of import. For instance, Rybka 1.6 uses Crafty's piece numbering scheme, while Rybka 1.0 uses Fruit's. See §4.5 for more.

2.3 Evidence with Rybka versions until Rybka 2.3.2a

A panoply of Rybka versions from Rybka 1.0 Beta (Dec 2005) until Rybka 2.3.2a (June 2007) were disassembled, though none was considered in as much depth as the two "endpoint" versions. There were significant changes to the search, but the evaluation remained much the same. The first notable variations in the latter seem to be approximately contemporaneous with the hiring of Larry Kaufman (Feb 2007?) to re-write the evaluation function. His work had minimal impact by the time of Rybka 2.3.2a, and it seems not until Rybka 3 that the problematic "Fruit overlap" of the Rybka evaluation function was eliminated. An enumeration of the earlier changes in the evaluation code can be found at this link, and a more complete comparison for Rybka 2.3.2a is given here.

2.4 Evidence with Rybka 2.3.2a

As it appeared at the same time as Rybka's victory in the 2007 WCCC[12] and marked the end of the Rybka 2.x series (with a 14 month-gap until Rybka 3), this version saw more extensive analysis by the Panel. The evaluation function was again concluded to have substantial similarity to the Fruit 2.1 evaluation function at the feature-choice level. The overlap was not quite so large as with Rybka 1.0 Beta, but was still a few standard deviations more than one might expect from chance.

The fact that much (perhaps the majority) of Rybka as a whole had changed by this point only served to mitigate the *extent* of the Fruit issue, not to eliminate the complaint all together.

2.5 Evidence concerning later Rybka versions

Later Rybka versions were not considered in depth by the Panel as a whole for a number of reasons. Firstly, these Rybka versions are not publicly available, and not every member has purchased (or otherwise obtained) a copy.[13] Secondly, there was no specific claim that these Rybka versions copied program X, and to sort through all possibilities seemed unmotivated. Finally, the investigation of the Panel had reached a logical "place of rest", with Rybka 2.3.2a (which seems

[12]The Panel surmised that Rybka 2.3.2a was a suitable surrogate for the exact Rybka version(s) that competed, and indeed forum posts by the operator(s) tend to confirm this.

[13]In this regard, the only parts of Rybka 1.6.1 that have been publicly disseminated are some of the ones that are Crafty-derivative.

essentially the same as the first WCCC winner from the Rybka series) being agreed, without explanation by Rajlich, to have transgressed the "originality" rule, and so the evidentiary burden was considered to have been passed to him.

That Rybka 4 used magic multiplication code from Buzz (possibly via Crafty) was noted, and also that Larry Kaufman's re-writing of the evaluation function was likely to have removed any complaint in that regard. However, the Panel as a whole was unwilling to state anything beyond that. Indeed, there was significant disagreement on what should be done to "verify" later Rybka versions (all agreed that *something* was necessary), and leaving the issue until after Rajlich had responded to the ICGA Board seemed a wiser course.

The Buzz license allows the code to be re-used, subject to certain conditions (which were met). For ICGA purposes, the re-use of this move generation code was a minor consideration compared to the other accusations. When Levy inquired whether the Rybka source code was original, Rajlich denoted "magic numbers" as a "standard exception". It was not listed on his 2010 WCCC entry form.

Crafty started using magic multiplication in the Buzz manner starting from version 21.3 (2007). Hyatt indicates that he took the information from forum posts, rather than from the Buzz code.

3 Evidence with Rybka 1.6.1

The evidence here is quite voluminous, and I will simply link to it at times.

3.1 En passant and obsolete tablebases

Here is code from Crafty 19.0 to work around a problem with *en passant* in some versions of Edwards tablebases. These tablebases were essentially obsolete by 2003, due to the widespread switch to Nalimov tablebases (first appearing around 2000).[14]

```
TB_use_ok=1;
if (TotalWhitePawns && TotalBlackPawns) {
  wpawn=FirstOne(WhitePawns);
  bpawn=FirstOne(BlackPawns);
  if (FileDistance(wpawn,bpawn) == 1) {
    if(((Rank(wpawn)==RANK2) && (Rank(bpawn)>RANK3)) ||
       ((Rank(bpawn)==RANK7) && (Rank(wpawn)<RANK6)) ||
       EnPassant(1)) TB_use_ok=0;    }  }
```

Note that the above code is quite particular to KP vs KP (it will fail for KPP vs KP due to FirstOne usage) and avoiding *en passant* troubles.

As noted by Bob Hyatt, there is no earthly reason for any program that claimed to have been started in 2003 to have such code, other than that it was mindlessly copied from Crafty without the slightest understanding of its purpose. Steven Edwards concurred. Hyatt discussed this (and other Rybka/Crafty issues) at this link.

[14] Crafty 16 removed all essential Edwards functionality, though the obsolete "workaround code" with *en passant* remained for many years after. In Feb 2004, Rajlich asked permission from Nalimov to use his tablebase code in Rybka.

Here is the Rybka 1.6.1 disassembly (continued on next page):

```
0x44cad2:  mov    0x6b8d54,%eax     # global pointer
0x44cad7:  mov    0xb10(%eax),%cl   # load TotalWhitePawns
0x44cadd:  add    $0x4,%esp
0x44cae2*  movl   $0x1,-0x1c(%ebp)  # set TB_use_OK to 1
0x44cae0:  test   %cl,%cl           # if TotalWhitePawns
0x44cae9:  je     0x44cbab
0x44caef:  mov    0xb11(%eax),%cl   #    && TotalBlackPawns
0x44caf5:  test   %cl,%cl
0x44caf7:  je     0x44cbab
0x44cafd:  mov    0xa78(%eax),%edx    # load WhitePawns (32 bits)
0x44cb03:  mov    0xa7c(%eax),%eax    # load WhitePawns (other 32)
0x44cb09:  mov    %edx,-0x8(%ebp)
0x44cb0c:  mov    %eax,-0x4(%ebp)
0x44cb0f:  bsf    -0x8(%ebp),%edx     # FirstOne for WhitePawns
0x44cb13:  mov    $0x0,%eax
0x44cb18:  jne    0x44cb2a
0x44cb1a:  bsf    -0x4(%ebp),%edx
0x44cb1e:  mov    $0x20,%eax
0x44cb23:  jne    0x44cb2a
0x44cb25:  mov    $0x20,%edx
0x44cb2a:  add    %edx,%eax
0x44cb2c:  mov    %eax,%ebx           # store FirstOne in ebx (wpawn)
0x44cb2e:  mov    0x6b8d54,%eax       # reload global pointer
0x44cb33:  mov    0xa80(%eax),%ecx    # load Black Pawns (32 bits)
0x44cb39:  mov    0xa84(%eax),%edx    # load Black Pawns (other 32)
0x44cb3f:  mov    %ecx,-0x8(%ebp)
0x44cb42:  mov    %edx,-0x4(%ebp)
0x44cb45:  bsf    -0x8(%ebp),%edx     # FirstOne for BlackPawns
0x44cb49:  mov    $0x0,%eax
0x44cb4e:  jne    0x44cb60
0x44cb50:  bsf    -0x4(%ebp),%edx
0x44cb54:  mov    $0x20,%eax
0x44cb59:  jne    0x44cb60
0x44cb5b:  mov    $0x20,%edx
0x44cb60:  add    %edx,%eax
0x44cb62:  mov    %eax,%ecx           # store FirstOne in ecx (bpawn)
0x44cb64:  mov    %ecx,%edx
0x44cb66:  and    $0x7,%edx           # file of bpawn
0x44cb69:  mov    %ebx,%eax
0x44cb6b:  and    $0x7,%eax           # file of wpawn
0x44cb6e:  sub    %edx,%eax           # compute distance
0x44cb70:  cltd
```

```
0x44cb71:  xor    %edx,%eax
0x44cb73:  sub    %edx,%eax           # absolute value of distance
0x44cb75:  cmp    $0x1,%eax           # if FileDistance is 1
0x44cb78:  mov    0x6b8d54,%eax [rereload global pointer]
0x44cb7d:  jne    0x44cbab
0x44cb7f:  sar    $0x3,%ebx             # get Rank of wpawn
0x44cb82:  cmp    $0x1,%ebx             # if rank is RANK2
0x44cb85:  jne    0x44cb91
0x44cb87:  mov    %ecx,%edx               # copy bpawn to edx
0x44cb89:  and    $0xfffffff8,%edx        # get the rank of it
0x44cb8c:  cmp    $0x10,%edx              # if the rank exceeds RANK3
0x44cb8f:  jg     0x44cba8                  # set TB_use_ok = 0 (@0x44cba8)
0x44cb91:  and    $0xfffffff8,%ecx      # get Rank of bpawn
0x44cb94:  cmp    $0x30,%ecx            # if rank is RANK7
0x44cb97:  jne    0x44cb9e
0x44cb99:  cmp    $0x5,%ebx               # and Rank(wpawn) is < RANK6
0x44cb9c:  jl     0x44cba8                  # set TB_use_ok = 0 (@0x44cba8)
0x44cb9e:  mov    0x10b(%eax),%cl       # load EnPassant(1) [1 is the ply]
0x44cba4:  test   %cl,%cl               # if there is no ep target
0x44cba6:  je     0x44cbab                # skip the next instruction
0x44cba8:  mov    %esi,-0x1c(%ebp)      # [sets TB_use_ok = 0]
0x44cbab:  mov    0x6b0951,%cl
```

3.2 Comparing to 99999 (more dead code mindlessly copied)

In its evaluation code, Crafty calls an `EvaluateMate()` routine, and compares the result to the value of 99999. Rybka 1.6.1 does the same; however, since Rybka uses *even* values therein, the return can never be 99999. See this link.

3.3 Repeated zeroing of a byte (an error that was copied)

Some versions of Crafty had a typo which caused it to redundantly clear a byte in a given data structure. Here is an example from Crafty 19.0, in `option.c`:

```
pawn_hash_mask=(1<<log_pawn_hash)-1;
 for (i=0;i<pawn_hash_table_size;i++) {
   (pawn_hash_table+i)->key=0;
   (pawn_hash_table+i)->p_score=0;
   (pawn_hash_table+i)->protected=0;
   (pawn_hash_table+i)->black_defects_k=0;
   (pawn_hash_table+i)->black_defects_q=0;
   (pawn_hash_table+i)->white_defects_k=0;
   (pawn_hash_table+i)->white_defects_q=0;
   (pawn_hash_table+i)->passed_w=0;
   (pawn_hash_table+i)->passed_w=0;      // repeated zeroing
   (pawn_hash_table+i)->outside=0;
   (pawn_hash_table+i)->candidates_w=0;
   (pawn_hash_table+i)->candidates_b=0; }
```

Crafty has similar code in more than one place, and some of these had the typo fixed in various versions. Rybka 1.6.1 has this repeated zeroing at 2 of 3 possible places, and I give one of them here.

```
0x45c8b5: xor  %ecx,%ecx            # ecx = 0, to be stored
[...]
0x45c9a7: xor  %eax,%eax
0x45c9a9: lea  0x0(%esp),%esp       # loop start
0x45c9b0: mov  0x6b8998,%esi
0x45c9b6: mov  %ecx,(%eax,%esi,1)   # 4 bytes @ 0x0
0x45c9b9: mov  0x6b8998,%esi          # overly strict compiler?
0x45c9bf: mov %ecx,0x4(%eax,%esi,1) # 4 bytes @ 0x4
0x45c9c3: mov  0x6b8998,%esi
0x45c9c9: mov  %cx,0x8(%eax,%esi,1) # 2 bytes @ 0x8
0x45c9ce: mov  0x6b8998,%esi
0x45c9d4: mov  %cl,0xf(%eax,%esi,1) # 1 byte @ 0xf
0x45c9d8: mov  0x6b8998,%esi
0x45c9de: mov  %cl,0xf(%eax,%esi,1) # 1 byte @ 0xf, (as previous)
0x45c9e2: mov  0x6b8998,%esi
0x45c9e8: mov  %cl,0xa(%eax,%esi,1) # 1 byte @ 0xa
0x45c9ec: mov  0x6b8998,%esi
0x45c9f2: mov  %cl,0x11(%eax,%esi,1) # 1 byte @ 0x11
0x45c9f6: mov  0x6b8998,%esi
0x45c9fc: mov  %cl,0x10(%eax,%esi,1) # 1 byte @ 0x10
0x45ca00: mov  0x6b8990,%esi
0x45ca06: inc  %edx
0x45ca07: add  $0x18,%eax            # struct has 0x18 bytes
0x45ca0a: cmp  %esi,%edx
0x45ca0c: jl   0x45c9b0              # END LOOP
```

Note that the Rybka 1.6.1 hash structure does differ (slightly) from that of Crafty, but it keeps the above bug. See this link for more.

3.4 Evaluation and search

The above elements were more probative in nature, but more substantive code from Crafty also appears in the Rybka 1.6.1 executable. The next two sections will describe parts that are copied from the evaluation and search respectively, well agreed to be the main heart of any chess program. To the extent one can conclude such from disassembly, the copying appears to be almost verbatim.

3.4.1 EvaluateWinner()

This is a quite long portion of "copied code" from Crafty that appears in the Rybka 1.6.1 executable. I will just point out the first bits, and link to the rest.

There is first the "top-level" call to it, which has a commonality of comparing `TotalWhitePieces` to 13 (same with `Black`) and the mechanics of a `can_win` variable. See this link for more.

There is then approximately 100 lines of Crafty code, whose assembly appears in the Rybka 1.6.1 executable.[15] A fuller listing is at this link. Here I just give the first two parts (of 8). Perhaps part of this could be argued to be formulaic, though as noted above the copying seems verbatim here in any event.

```
int EvaluateWinner(TREE * RESTRICT tree) {
  register int can_win=3;
/*  if a side is a piece up, but has no pawns, that side cannot win */
  if (WhiteMajors == BlackMajors) {
    if (TotalWhitePawns==0 && WhiteMinors-BlackMinors==1) can_win&=2;
    if (TotalBlackPawns==0 && BlackMinors-WhiteMinors==1) can_win&=1;
    if (can_win == 0) return(can_win); }
```

Here is the corresponding code in Rybka 1.6.1:

```
0x401630: push   %ebp             # start EvaluateWinner
0x401631: mov    %esp,%ebp        # esp fiddle & 3 "push"es omitted
0x401649* mov    $0x3,%esi        # "can_win" = 3
0x401636: mov    0x8(%ebp),%ecx
0x401639: mov    0xb0e(%ecx),%al  # load WhiteMajors
0x40163f: mov    0xb0f(%ecx),%dl  # load BlackMajors
0x401645: cmp    %dl,%al          # if these are equal
0x40164f: mov    %esi,-0x4(%ebp)
0x401652: jne    0x4016a7
0x401654: mov    0xb10(%ecx),%bl    # load TotalWhitePawns
0x40165a: test   %bl,%bl            # if TotalWhitePawns = 0
0x40165c: jne    0x40167b
0x40165e: movsbl 0xb0d(%ecx),%edi     # load BlackMinors
0x401665: movsbl 0xb0c(%ecx),%ebx     # load WhiteMinors
0x40166c: sub    %edi,%ebx            #  WhiteMinors-BlackMinors
0x40166e: cmp    $0x1,%ebx            # if result is 1
0x401671: jne    0x40167b
0x401673: mov    $0x2,%esi
0x401678: mov    %esi,-0x4(%ebp)        # set "can_win" to 2
0x40167b: mov    0xb11(%ecx),%bl    # load TotalBlackPawns
0x401681: test   %bl,%bl            # if TotalBlackPawns is 0
0x401683: jne    0x40169f
0x401685: movsbl 0xb0d(%ecx),%edi     # load BlackMinors
0x40168c: movsbl 0xb0c(%ecx),%ebx     # load WhiteMinors
0x401693: sub    %ebx,%edi            #  BlackMinors-WhiteMinors
0x401695: cmp    $0x1,%edi            # if result is 1
0x401698: jne    0x40169f
0x40169a: and    %edi,%esi            # AND "can_win" with 1
0x40169c: mov    %esi,-0x4(%ebp)
0x40169f: test   %esi,%esi          # if can_win is 0
0x4016a1: je     0x4016e7             # return 0
```

[15] There might be a minor difference in how some of the bitboards are accessed, for instance something like `BishopsAndQueens` versus splitting them.

Back to Crafty 19 for the next part of the code:

```
/* ----------------------------------------------------------
  |   if one side is an exchange up, but has no pawns, then  |
  |   that side can not possibly win.                        |
   ---------------------------------------------------------- */
  if (WhiteMajors != BlackMajors) {
    if ((WhiteMajors-BlackMajors) == (BlackMinors-WhiteMinors)) {
      if (TotalBlackPawns==0) can_win&=1;
      if (TotalWhitePawns==0) can_win&=2; }
    if (can_win == 0) return(can_win); }
```

And to the Rybka 1.6.1 disassembly:

```
0x401639: mov    0xb0e(%ecx),%al  # load WhiteMajors
0x40163f: mov    0xb0f(%ecx),%dl  # load BlackMajors
[...]
0x4016a3: cmp    %dl,%al          # if WhiteMajors != BlackMajors
0x4016a5: je     0x4016f0
0x4016a7: movsbl 0xb0d(%ecx),%edi   # load BlackMinors
0x4016ae: movsbl 0xb0c(%ecx),%ebx   # load WhiteMinors
0x4016b5: movsbl %dl,%edx
0x4016b8: movsbl %al,%eax
0x4016bb: sub    %ebx,%edi          # BlackMinors-WhiteMinors
0x4016bd: sub    %edx,%eax          # WhiteMajors-BlackMajors
0x4016bf: cmp    %edi,%eax          # compare these last 2
0x4016c1: jne    0x4016e3           # if equal
0x4016c3: mov    0xb11(%ecx),%al      # load TotalBlackPawns
0x4016c9: test   %al,%al              # if TotalBlackPawns is 0
0x4016cb: jne    0x4016d3
0x4016cd: and    $0x1,%esi              # AND "can_win" with 1
0x4016d0: mov    %esi,-0x4(%ebp)
0x4016d3: mov    0xb10(%ecx),%al      # load TotalWhitePawns
0x4016d9: test   %al,%al              # if TotalWhitePawns is 0
0x4016db: jne    0x4016e3
0x4016dd: and    $0x2,%esi              # AND "can_win" with 2
0x4016e0: mov    %esi,-0x4(%ebp)
0x4016e3: test   %esi,%esi          # if "can_win" is 0
0x4016e5: jne    0x4016f0
0x4016e9: xor    %eax,%eax            # return 0
0x4016ef: ret
[...]
```

Note that Rybka checks `TotalBlackPawns` before `TotalWhitePawns` here, just like Crafty, and the opposite of the order in the first part. Similarly, in Part III (see above link), Rybka and Crafty both choose to compare `TotalWhitePieces` then `TotalBlackPieces` in the first part, but then both switch the order in the latter two segments. A list of such quirks can be extended.

3.4.2 NextMove() mechanics

The above touches on various elements of the Crafty evaluation, but not the search. I did not look completely at the latter (nor indeed with the former), but there are indeed some Crafty parts extant in the Rybka 1.6.1 executable. The most notable one is in the `NextMove()` routine. See this link for more. I give but two small parts below.

First, the numbering of phases is the same in both. Crafty's is given here, while Rybka's can be inferred from their use in the routine.[16]

```
#define NONE                      0
#define HASH_MOVE                 1
#define GENERATE_CAPTURE_MOVES    2
#define CAPTURE_MOVES             3
#define KILLER_MOVE_1             4
#define KILLER_MOVE_2             5
#define GENERATE_ALL_MOVES        6
#define SORT_ALL_MOVES            7
#define HISTORY_MOVES_1           8
#define HISTORY_MOVES_2           9
#define REMAINING_MOVES          10
#define ROOT_MOVES               11
```

Both use these phases in the same way, e.g., Rybka 1.6.1 uses phases 1, 7, and 10 when in check (`NextEvasion`), just as Crafty does.

Here (for instance) is the 9th phase in both Crafty and pre-Beta Rybka. It is already somewhat odd to have this `HISTORY_MOVES_2` phase in the first place.

```
case HISTORY_MOVES_2:
  bestval=0;
  bestp=0;
  for (movep=tree->last[ply-1];movep<tree->last[ply];movep++)
    if (*movep) {
      index=*movep&4095;
      history_value= (wtm) ? history_w[index] : history_b[index];
      if (history_value > bestval) {
        bestval=history_value;
        bestp=movep; } }
  if (bestval) {
    tree->current_move[ply]=*bestp;
    *bestp=0;
    tree->next_status[ply].remaining++;
    if (tree->next_status[ply].remaining > 3) {
      tree->next_status[ply].phase=REMAINING_MOVES;
      tree->next_status[ply].last=tree->last[ply-1];
    }
    return(HISTORY_MOVES_2);
  }
```

[16] Including `ROOT_MOVES` (as 11) in `NextRootMove()`, which is at `0x459f00` in Rybka 1.6.1.

And the Rybka 1.6.1 code for the second half of this:

```
0x44bbd4: test  %ebp,%ebp               # if (bestval)
0x44bbd6: je    0x44bc22
0x44bbd8: mov   (%edx),%eax
0x44bbda: lea   0x2c7(%edi,%edi,2),%ecx
0x44bbe1: mov   %eax,0x1060(%esi,%edi,4)  # tree->current_move[ply]=*bestp;
0x44bbe8: lea   (%esi,%ecx,4),%eax
0x44bbeb: movl  $0x0,(%edx)               # *bestp = 0
0x44bbf1: mov   (%eax),%edx
0x44bbf3: inc   %edx                      # increment #remaining
0x44bbf4: mov   %edx,%ecx
0x44bbf6: cmp   $0x3,%ecx                 # if #remaining > 3
0x44bbf9: mov   %edx,(%eax)
0x44bbfb: jle   0x44bc14
0x44bbfd: mov   0x137c(%esi,%edi,4),%edx    # load tree->last[ply-1]
0x44bc04: movl  $0xa,0xb20(%ebx)            # phase = 10 (REMAINING_MOVES)
0x44bc0e: mov   %edx,0xb18(%ebx)            # tree->next_status[ply].last =
0x44bc14: mov   0x80(%esp),%ebp                     tree->last[ply-1];
0x44bc1b: mov   $0x9,%eax                 # phase 9 is HISTORY_MOVES_2
0x44bc20: jmp   0x44bc71                  # return (HISTORY_MOVES_2)
```

The top half of the Crafty code is perhaps suitably formulaic; the bottom half includes the congruence that both look for *three* remaining moves (not to mention the aforementioned common phase distinction in the first place).

3.5 Conclusion

Rybka 1.6.1 from November 2004 contains large amounts of code taken from Crafty 19. The complete extent of this has not been determined, as the amount found is already thought to be (quite) preponderant.

4 Evidence with Rybka 1.0 Beta

This section will give a recapitulation of the most notable evidence with Rybka 1.0 Beta and Fruit 2.1, particularly with regards to the ICGA originality requirement. Also, as noted above, at the very least, the evaluation function similarity persisted until Rybka 2.3.2a. There could also be various elements of similarity that are not noted here.

4.1 Probative similarity

I will only mention a few elements of this. These serve to indicate that it is very likely that Rajlich copied quite specific elements of Fruit 2.1 in making Rybka 1.0 Beta (whether by copy/paste, re-typing, or minor variation: depends on the specific case, is likely unknowable, and is probably irrelevant). Some of the elements could additionally have "substantive" impact.

4.1.1 Search control

This consists of a rather short piece of code to determine what (and how) to do when a search has concluded (the UCI protocol specifies that the `bestmove` should in some cases not be sent immediately). 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();
```

Here is the corresponding code from the Rybka 1.0 Beta executable.

```
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   $0x1,$(0x66c32c)        # SearchInput->Infinite = true
0x407063* movb   $0x1,$(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,$(0x6696e2)       # set Infinite = false
0x407078: jmp    0x407081              # else
0x40707a: movb   $0x1,$(0x6696e2)        # set Infinite = true
0x407081: movb   $0x0,$(x06696e3)      # set Delay = false
0x407088: callq  0x408f90              # call the search function
0x40708d: movzbl $(0x6696e2),%eax      # load Infinite variable
0x40709b: movb   $0x0,$(0x6696e1)      # set Searching = false
0x4070a2: mov    %al,$(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 (redunantly) computed twice in both. Also, the operations with `Searching`, `Infinite`, and `Delay` are ordered the same.[17] Finally, setting `Delay` to be “false” can be seen as redundant in Fruit, as three lines above it was `ASSERT`ed to be so.

The above code is quite idiosyncratic to Fruit 2.1 (it differs slightly in Fruit 1.0), and re-appears essentially verbatim in Rybka 1.0 Beta.

[17] As are their variable allocations, which is a quite strong indicator of the “origins” of this part of Rybka 1.0 Beta. See the iterative deepening code below (§4.3) for another example.

4.1.2 UCI parsing, time management [and floating-point 0.0]

In a previous version of this document, I had described more fully the occurrence of 0.0 in Rybka's integer-based time management. However, after consulation with Dalke and Schröder, I have chosen to replace much of this section with a different example of code copying (§4.1.3).

One can list various similarities of the time management procedures in Fruit and Rybka (such as multiplying `movetime` by 5). Rybka 1.0 Beta has a integer-based time management procedure, yet at one point compares an integer variable with the floating-point zero (as per Fruit).

More generally, the UCI parsing (e.g., the use of incremental `strtok`), and the splitting of said parsing and time management across subroutines is rather characteristic in Fruit, and yet re-appears in Rybka 1.0 Beta.[18] For comparison, here are renditions of the time management for Fruit 2.1, Rybka 1.0 Beta (by Schröder, correcting Fadden), and Rybka 1.6.1 (by me).

Fruit 2.1 time management

```
if (movetime >= 0.0) { // Fruit 2.1
   SearchInput->time_is_limited = true;
   SearchInput->time_limit_1 = movetime * 5.0; // HACK 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; }
```

Rybka 1.0 time management

Although the 0.0 comparison received much attention, in his original posts concerning it, Wegner had already preferred to note the importance of the *next* line, with the odd multiplication by 5. (This is a somewhat clumsy way of choosing a number that is sufficiently larger than `movetime`, not likely to be replicated by chance).

```
 if (movetime > 0.0) { // Rybka 1.0 Beta, Fadden/Schroeder
   time_limit_1 =    5 * movetime;
   time_limit_2 = 1000 * movetime; // possibly a bug in Rybka
 } 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; }
```

Next is the Rybka 1.6 code; it does not follow the above pattern much. For instance, `movestogo` involves game phase, and combines increment differently.

[18] Again one can note that Rybka 1.6.1 lacks any Fruit-similarity herein. One example here is that Rybka 1.6.1 implements the `searchmoves` UCI command, while the "successor" Rybka 1.0 Beta does not, and there appears little if any reason to remove such functionality. Crafty is (of course) not UCI, and thus any such code in Rybka 1.6.1 did not derive from it.

Rybka 1.6 time management. (This is my C-code reconstruction of the disassembly).

```
mtg=movestogo; mat=9*Q+5*R+3*(B+N); // my material, 0 to 31
if (mat>=23 && mtg==40) mtg=(mtg*4)/3; // start of game
if (mat<13 && mtg==40) mtg=(mtg*2)/3; // end of game
if (depth>0 && depth<=64 || nodes>0 && nodes<INT_MAX) goto search;
tm=time-1200; // subtract off 12s, working in centiseconds
X=min(4,(4*inc)/(tm/mtg));
if (movestogo==40) Y=((tm*(X+4))/5)/mtg; // 40 is default
else Y=(4*tm)/5/movestogo; // use movestogo here (not mtg)
T=Y+inc; if (T>movetime) T=movetime; // user-specified ``movetime''
if (T<100) T=100; // minimum 1 second
```

The Rybka 1.6 code uses integer arithmetic throughout. For instance, if `time` is 100s and `inc` is 1s, the result is 2.99s, coming from 100+199, the latter being (8800·6)/5/53, with 6 coming from 4+(4·100)/(8800/53). This computes `T` the target time (`time_limit_1`), while the absolute time (`time_limit_2`) is `time+(inc-100)*movestogo-1200`, except when `inc` is less than 1s, when it is `time+(100-inc)*movestogo-1200`, with again 100 being the minimum.

4.1.3 BadThreshold and flag usage

This section did not appear in previous versions.

Fruit 2.1 and Rybka 1.0 Beta both do the same to check if a previously preferred move has suddenly become bad. Here is the Fruit 2.1 code:

```
if (UseBad && SearchBest->depth > 1) {
   if (SearchBest->value <= SearchRoot->last_value - BadThreshold) {
      SearchRoot->bad_1 = true;
      SearchRoot->easy = false;
      SearchRoot->flag = false;
   } else {
      SearchRoot->bad_1 = false; }
```

One thing to note is that unsetting `flag` here is useless. Indeed, it essentially operates as a local variable. The only way `flag` could be `true` is in the iterative deepening loop (see below), when the search is immediately exited, or when the user sends a command like "stop", when again the search is exited (via `setjmp`). Its usage in Fruit 2.1 seems to be a remnant from when a different method of search control was used. For this code segment, Rybka 1.0 Beta has:

```
0x000040b86f: cmpl   $0x1,0x264de2(%rip)        # 0x670658 depth > 1
0x000040b876: jbe    0x40b8a3
0x000040b878: mov    0x260bca(%rip),%eax        # 0x66c448 last_value
0x000040b87e: add    $0xfffffffffffffce,%eax    // -50
0x000040b881: cmp    %eax,%ecx
0x000040b883: jg     0x40b89c
0x000040b885: movb   $0x1,0x260bc0(%rip)        # 0x66c44c  bad_1
0x000040b88c: movb   $0x0,0x260bbc(%rip)        # 0x66c44f  easy
0x000040b893: movb   $0x0,0x260bb6(%rip)        # 0x66c450  flag
0x000040b89a: jmp    0x40b8a3
0x000040b89c: movb   $0x0,0x260ba9(%rip)        # 0x66c44c  bad_1
```

One can see the same test(s), and the same ordering of results, including the quirky (and unneeded) unsetting of `flag`. The margin of 50 is also the same.

This section is in some ways complementary to the `EasyThreshold` seen below in the iterative deepening section (§4.3). Indeed, if one believes the compiler kept the maths on the proper side of the equation, one sees that Rybka does $V \leq L - 50$ just as Fruit here, and $V \geq X + 150$ for the case of `EasyThreshold`. One can note that Strelka, a reverse-engineering of Rybka 1.0 Beta, erroneously gets the comparison here as $V < L - 50$, while Rybka retains the Fruit usage.

4.1.4 Root search ordering

A third probative aspect is the ordering of procedures in the root search. Again there are some aspects in this that are forced upon a computer chess program, while there are other aspects that can vary. For the latter, Fruit 2.1 follows its own course, typically different from other programs.

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

Except for a few obvious constraints (for instance, move generation must precede scoring/sorting them), much of the above can be done (or re-factored) in a notably different manner.[19] Here are the comparative operations/ordering in Phalanx XXII (for example): generate legal moves, init killers/history, increment Age (date), setup time limits, sort root moves, start timer, return move if forced (there are various bits about book/learning that I omit).

Finally, the concluding part of "root search", namely the iterative deepening, is analyzed more completely below, as it is more substantive in nature.

4.1.5 Probative similarity conclusion

One natural conclusion of the above probative similarity analysis (and the Crafty evidence) is that it really difficult to give Rajlich the "benefit of doubt" on more substantive questions.

In all these aspects (and others that could be given), one can note a "jump discontinuity" between the internals of Rybka 1.6.1 and Rybka 1.0 Beta, and

[19] 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.

indeed the latter usually looks quite similar to Fruit 2.1. This is discussed a bit more in a subsection below.

4.2 The comparison of evaluation functions

This element is noted here in depth because it played a major part in forming the consensus of the ICGA Panel. However, other elements are still of import on an overall basis.

4.2.1 The significance of the evaluation function

It is well-agreed that heuristics used in the evaluation and search form the primary basis (80-90%) of the creative content of any computer chess program. Of course, "technical" aspects such as speed of code cannot be ignored, and indeed affect various design decisions which must be made, but since the days of Turing and Shannon the search and evaluation have been considered to be the dominant aspects of the play of an engine, and have formed the major part of computer chess research.

One can divide both of these into subheadings. For search, one can talk about high-level differences such as (say) alpha-beta versus best-first search, or (once the general search framework is chosen) aspects that are more idiosyncratic to each individual engine, such as specific heuristics for pruning/extensions that shape the search tree. Both Fruit 2.1 and Rybka 1.0 Beta stay in the general alpha-beta framework, with only a few (though notable for each) divergences from a "textbook" implementation.[20]

Similarly, one can divide the evaluation process into subheadings, namely first *what* to evaluate (which can be constrained by efficiency issues), and secondly how to weight the things that are evaluated. The relative importance of these is not always easy to decipher. For instance, modern technology can allow the computations of weights to be relatively automated, which tends to delimit the scope of creativity here.[21]

The choice of what "evaluation features" to use is a critical one for any author to make. One can try to follow textbook enumerations (often given in rather generic form), or to borrow alternatively from the pool of general knowledge, but inevitably specific choices must be made, and these can have a large impact on style and strength of an engine. Furthermore, each "evaluation feature" itself can be implemented in a variety of ways (see below for an example), which further adds to the idiosyncratic nature of each program.

This being said, external factors (such as strength) do play a part in determining which evaluation features to use, though the process appears to retain much more of a sense of "art" than "science" at the current time.[22] Note too,

[20] Later Rybka versions differ substantially from Rybka 1.0 Beta in the search aspects. Some of this is related to adding multi-processor search, but Rybka 1.1 (Mar 2006) had already added nearly 100 Elo over Rybka 1.0 Beta while making almost no evaluation changes.

[21] Going back to the 1990s, it seems more (relatively) common then for the weights to be derived from "expert knowledge", some of which is now thought suspect.

[22] Alan Sassler noted that one could presumably take all known features and do a primary component analysis of them to determine the most relevant (this being a standard practice in

one of Kaufman's primary tasks with Rybka 3 was to find good features, and the Fruit 2.1 features were developed over a 15-month period (Fruit 1.0 had very little eval at all). Rajlich himself stated in Nov 2004: *Unfortunately at the moment Rybka follows an even more basic law: if your evaluation is bad, at least keep it cheap ;) There's nothing worse than a big mess of untested eval terms ...*

My own feeling is that approximately 50% of the "creative value" in Fruit 2.1 was in its choice of evaluation features, another 25% in the search,[23] and 25% for other assorted items (such as evaluation numerology).

4.2.2 What was compared

In order to determine whether the Fruit/Rybka evaluation feature overlap was out of the ordinary, a representative selection of 6 open-source[24] engines was made across varying strengths. These included popular engines such as Crafty and some lesser-known ones, some with "minimal" evaluation functions (Faile), and some with much more weighty ones (Phalanx XXII). Some of the engines to include were suggested by Panel members. However, the choice has come criticism, for which see below. This made a pool of 9 engines, when Fruit 2.1, Rybka 1.0 Beta, and Rybka 2.3.2a were included.

4.2.3 How it was compared

The basis for comparison was on the evaluation features. These were agreed by the consensus of the Panel to be "protectable" in the sense of ICGA originality, in a similar sense to the plot of a book being protectable under copyright.

There were approximately 50 elements that could be distinguished in one engine or another. In some cases, closely related evaluation features were merged, and it was not always clear how to sub-divide features (particularly king safety). With some of the features, aspects of relative weighting were also considered.

Since each feature is not a binary yes/no, but can (e.g.) vary in many ways with subconditions, each *pair* of engines was compared for its total "overlap" for each feature. The level of abstraction chosen was a written description of what each engine considered for each feature (many features were absent in many of the programs). Only aspects which were essentially non-variable were filtered (for instance, the definition of "isolated pawns" in the example below).[25]

The mechanism of pool comparison (be it pairwise or otherwise) also induces a natural filtration, for any overly common element will appear in many engines.

Here is an example of what EVAL_COMP produced, for isolated pawns.

many fields). This is only partially applicable to computer chess, where features often have large correlations, and also the computation time for a feature must be considered.

[23] Whether or not Fruit's history reductions are that valuable for *strength* is another matter – here I speak of "value" in terms of creativity, and thus must also ignore Letouzey's exquisite engineering. I would give similar percentages for Rybka 1.0 Beta, but the greatly improved search heuristics of (say) Rybka 2.3.2a should teeter a larger percentage to that aspect.

[24] I use this term here simply to mean a program whose source was provided, and not to imply anything about the licensing subheadings therein.

[25] This seems in line with the general advice given for the Abstraction-Filtration-Comparison Test. External factors such as strength play a minor rôle, for engines of the same strength can have quite different evaluation features. A claim of *scènes à faire* also appears unmotivated, except at the most general level (all engines consider "king safety" as a broad concept, for instance). The issue of "public domain" has been raised *inter alia* in the sense of generally known chess knowledge, but it seems not to apply to what *specific* choices are made in a realization.

Fruit 2.1, Rybka 1.0 Beta, and Rybka 2.3.2a all give a penalty for an isolated pawn that depends on whether the file is half-open or closed [and make no other consideration].

Crafty 19.0 counts the number of isolated pawns, and the subcount of those on open files, and then applies array-based scores to these.

Phalanx XXII gives a file-based penalty, and then adjusts the score based upon the number of knights the opponent has, the number of bishops we have, and whether an opposing rook attacks it. There is then a correction if an isolated pawn is a ‘‘ram’’, that is, blocked by an enemy pawn face-to-face, and also a doubled-and-isolated penalty.

Pepito 1.59 has a file-based array for penalties, though the contents are constant except for the rook files. There is also a further penalty for multiple isolani.

Faile 1.4 penalises isolated pawns by a constant amount, with half-open files penalised more (same as Fruit/Rybka).

RESP 0.19 also penalises isolated pawns by a constant amount, and gives an additional penalty to isolated pawns that are doubled.

EXchess also gives a constant penalty for isolated pawns, and further stores it in a king/queenside defect count.

Here are the numerical pairwise scores that were determined for this feature.

Table 2: Isolated pawns

-	Craf	RESP	Ryb1	R232	Phal	Fail	Fr21	Pepi	EX5b
Craf	1.0	0.2	0.4	0.4	0.2	0.4	0.4	0.4	0.2
RESP	0.2	1.0	0.3	0.3	0.3	0.3	0.3	0.3	0.7
Ryb1	0.4	0.3	1.0	1.0	0.2	1.0	1.0	0.3	0.5
R232	0.4	0.3	1.0	1.0	0.2	1.0	1.0	0.3	0.5
Phal	0.2	0.3	0.2	0.2	1.0	0.2	0.2	0.2	0.2
Fail	0.4	0.3	1.0	1.0	0.2	1.0	1.0	0.3	0.5
Fr21	0.4	0.3	1.0	1.0	0.2	1.0	1.0	0.3	0.5
Pepi	0.4	0.3	0.3	0.3	0.2	0.3	0.3	1.0	0.5
EX5b	0.2	0.7	0.5	0.5	0.2	0.5	0.5	0.5	1.0

It can be noted that isolated pawns is a bit odd in some respects, in that all engines had it, and all used the same definition. This was certainly not true of all features; for instance, only Fruit and Rybka 1.0 Beta had knight mobility, only Phalanx, Pepito, and EXChess had bishop outposts (variously defined), etc. Another aspect here was that definitions might sometimes be influenced by external factors (such as board representation), but this was largely ignored.[26]

The full EVAL_COMP analysis can be obtained at this link.

[26]One reason for ignoring this is that it was not clear how to codify it. Another reason was that, if anything, the adjustments would tend to increase the Fruit/Rybka overlap (for instance, they define open files differently, but this possibly might be simply due to efficiency).

4.2.4 The result

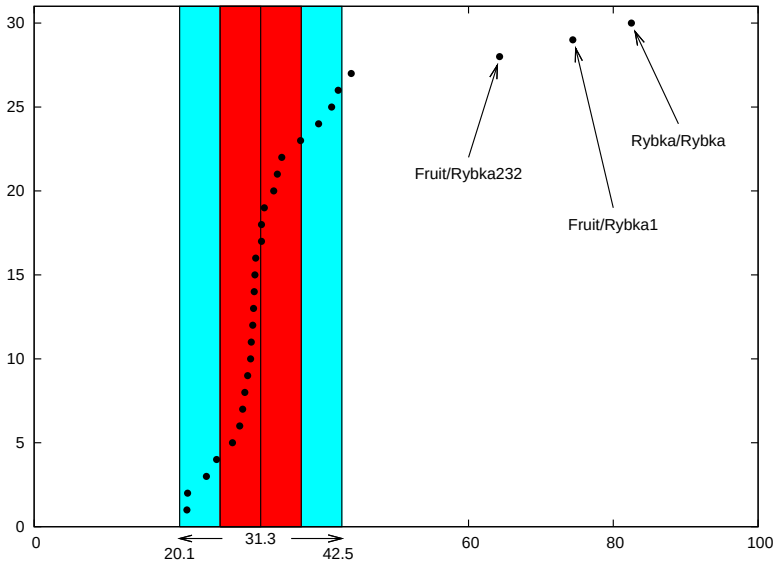
From the above engines, upon excluding the partially-dependent Rybka 2.3.2a, one obtains 28 overlap numbers from the pairs of engines. Although a fuller statistical analysis for the EVAL_COMP result remains obscure, it is abundantly clear that the overlap between Fruit 2.1 and Rybka 1.0 Beta is dramatically more than would be expected by chance. Using rudimentary statistics and crude assumptions of distribution, the rarity of the Fruit/Rybka overlap appears to be more than 1 in a million, most likely more than 1 in a billion, and possibly more than 1 in a trillion. The same was concluded concerning Rybka 2.3.2a and Fruit 2.1, with slightly smaller numbers.

Here are the final overlap numbers, which though on a scale of 0-100, should perhaps not be thought of merely as "percentages". In any event, it is the rarity of the Rybka/Fruit overlap that is of significance, and not the raw numbers.

Table 3: Evaluation Feature Overlap

-	Craf	RESP	Ryb1	R232	Phal	Fail	Fr21	Pepi	EX5b
Craf	100	39	32	31	33	21	34	41	30
RESP	39	100	31	34	29	30	34	31	44
Ryb1	32	31	100	82	31	25	74	31	30
R232	31	34	82	100	28	24	64	28	28
Phal	33	29	31	28	100	21	30	42	29
Fail	21	30	25	24	21	100	27	24	28
Fr21	34	34	74	64	30	27	100	30	30
Pepi	41	31	31	28	42	24	30	100	37
EX5b	30	44	30	28	29	28	30	37	100

The graph below displays 30 data points, with the mean of 31.3 for the 27 "control" points (removing Rybka 2.3.2a), and the shaded regions corresponding roughly to one and two standard deviations ($\sigma \approx 5.6$ for the control group).



4.2.5 Criticisms of EVAL_COMP

Some critical yet useful remarks about EVAL_COMP have been made, but too often they get lost in grandstanding and other various Internet behaviors.

One criticism of EVAL_COMP was that it was "subjective". As much of the basis for the numerical scores was given in the textual descriptions, it is difficult, however, to say it is "arbitrary". Presumably someone could independently repeat the analysis.

Various claims were made along the lines that the evaluation features in Fruit 2.1 are not "unique" to it, and thus not subject to any consideration of originality. However, the specific choice of *which* features to use, and what subcomponent properties to instrument therein, do seem to fall under such a heading, regardless of whether any part is itself sufficiently novel (e.g., a popular book on the history of the Netherlands has little unique to it, but the collative aspects are indeed creative).[27]

Another criticism was that too many "weak engines" were included. This was analyzed in part by Adam Hair, who did in fact note that there was a (weak) correlation between strength differential (for a given engine pair) and the observed evaluation feature overlap. However, he also indicated that the Rybka/Fruit data point was still an outlier by many orders of deviation. Also, if (say) one eliminates the weakest engine (Faile), the observed average overlap is affected only slightly, while the spread of overlaps tightens more noticeably, making the Rybka/Fruit overlap even more unlikely.[28]

Finally, it was mentioned that other "common" open-source engines should be used, such as TSCP 1.81 and GNUChess 5.07, as these have served as the starting point for a number of computer chess programs. Given that TSCP is quite weak, it is not clear how to integrate this with the previous point. Even GNUChess[29] is not all that strong compared to the chosen set of engines. I made a cursory glance of these two engines, but saw nothing that would lead me to think that their inclusion would dramatically change the EVAL_COMP result. One can also note that Crafty should be a suitable example of "common" starting point for many, yet it showed little additional feature overlap.

As was noted by Letouzey, the Fruit eval is defined as much by what it *omits* as what it includes. As an example, Fruit has no consideration of pieces attacking or defending each other.

Another point that is discernible from EVAL_COMP is that Rybka 1.0 did not simply take the "best" or "most useful" parts from Fruit 2.1 (which indeed would be quite arguably permissible, given the nature of progress) and then combine them with other ideas. Rather Rybka 1.0 took almost all the Fruit parts in evaluation; comparison with Rybka 1.6.1 shows that it kept almost nothing more than the Kaufman-esque material imbalance table.

[27] A related argument was: each evaluation feature could be described as an "idea", which was noncopyrightable when labelled as such, and the evaluation function was thus the sum of noncopyrightable parts, ergo itself unprotectable. I will refrain from mocking this too severely.

[28] Another obvious consideration here is the following. Rybka 2.3.2a had a light-weight Fruit-like evaluation. Rybka 3 contained a much fuller evaluation written by Larry Kaufman. Rybka 4 proceeded to remove a lot of these components. All were top-class engines. So it seems the external factor of engine strength is rather limited in any dictating of evaluation similarity.

[29] Meaning the *original* 5.07 version, and not later patches, as the latter are notably superior.

4.3 Control of iterative deepening

Another substantive element of Fruit 2.1 that appears in Rybka 1.0 Beta is in the iterative deepening, and deciding when to halt the search. It is possibly arguable this is more probative than substantive, but it plays a notable role in determining how the program behaves to the end-user (e.g., timing of moves). Also, being a "search" element, it book-ends the "evaluation" considerations above.

Fruit 2.1 code reformatted, ASSERTs 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    $(0x664538),%rcx   # [``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   $0x0,$(0x66c44e)   # set ``change'' to false
0x4095cd: movb   $0x0,$(0x66c44c)   # set ``bad_1'' to false
0x4095d4: callq  0x40ba70           # call search_full_root(depth)
0x4095d9: callq  0x4070c0           # some sort of update function
0x4095de: mov    $(0x670654),%r11d  # get score
0x4095e5: cmp    $0xffff8300,%r11d  # fiddle around
0x4095ec: jle    0x4095fe           # ...
0x4095ee: cmp    $0x7d00,%r11d      # ... with mate scores
0x4095f5: movzbl $(0x66c450),%edx   # load ``flag''
0x4095fc: jl     0x409601           # if mate score,
0x4095fe: mov    %sil,%dl             # set ``flag'' to true (esi=1)
0x409601: cmp    %esi,%ebx          # compare depth (%ebx) to 1
0x409603: jne    0x409631           # if depth == 1
0x409605: cmpl   $0x0,$(0x670664)     # this is RootMoveList[1] (move #1)
0x40960c: je     0x40962f             # if only 1 legal move, skip next
0x40960e: movzbl $(0x66c44f),%ecx     # read``easy''
0x409615: mov    $(0x670a64),%eax     # read (value of move #1)
0x40961b: add    $0x96,%eax           # EasyThreshold 150 [as in Fruit]
0x409620: cmp    %eax,$(0x670a60)     # read (value of move #0)
0x409626: cmovae %esi,%ecx            # if move values differ by enough
0x409629: mov    %cl,$(0x66c4   4f)     # set ``easy'' as true
0x40962f: cmp    %esi,%ebx          # if depth > 1
0x409631: jbe  0x40964d* [0x409647]
0x409633: movzbl $(0x66c44c),%eax     # load old bad_1
0x40963a: movb   $0x0,$(0x66c44c)     # bad_1 = false
0x409641: mov    %al,$(0x66c44d)      # bad_2 = (previous) bad_1
0x40964d* movzbl %dl,%eax           # flag = true, if mate score
0x409650* mov    %r11d,$(0x66c448)  # last_value = score
0x409647: cmp    $(0x66c328),%ebx   # see if depth>=depth_limit
0x409657: cmovae %esi,%eax          # if depth >= depth_limit
0x40965a: mov    %al,$(0x66c450)      # then ``flag'' is true
0x409666: mov    $(0x66c320),%r8d   # load SearchInput->time_limit_1
0x40966d: movzbl $(0x66c44d),%r9d   # load bad_2
0x409660* callq  *0x41d030          # GetTickCount() -> %eax
0x409675: mov    %eax,%r11d
0x40967c* sub    $(0x66c438),%r11d  # (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 ->edx with mul here)
0x40968a: shr    %edx               #  and div by 2
0x40968c: cmp    %edx,%r11d         # compare to time taken
0x40968f: jb     0x4096a6           # if small, ignore next
0x409691: movzbl $(0x66c450),%ecx     # ``flag''
0x409698: test   %r9b,%r9b            # if ``bad_2'' is false
0x40969b: cmove  %esi,%ecx              # ecx = 1 (esi is always 1)
0x40969e: mov    %cl,$(0x66c450)     # store ecx in ``flag''
0x4096a4: jmp    0x4096ac
0x4096a6: mov    $(0x66c450),%cl    # (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   $0x0,$(0x66c44f)     # see if ``easy''
0x4096c3: movzbl %cl,%eax             # if not ``easy'', eax = ``flag''
0x4096c6: cmovne %esi,%eax            # if is ``easy'', eax = true
0x4096c9: mov    %al,%cl
0x4096cb: mov    %al,$(0x66c450)      # store eax in ``flag''
0x4096d1: shr    %r8d               # time_limit_1 divided by 2
0x4096d4: cmp    %r8d,%r11d         # compare to time taken
0x4096d7: jb     0x4096f3           # if small, ignore next
0x4096d9: test   %r9b,%r9b            # if ``bad_2'' is true
0x4096dc: jne    0x4096f3             # then ignore next
0x4096de: cmp    %r9b,$(0x66c44e)       # ``change'', see if false
0x4096e5: movzbl %cl,%eax               # if not, eax = ``flag''
0x4096e8: cmove  %esi,%eax              # if change = false, eax = true
0x4096eb: mov    %al,%cl
0x4096ed: mov    %al,$(0x66c450)       # store eax in ``flag''
0x4096f3: cmpb   $0x0,$(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   $0x0,$(0x66c32c)     # 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 (as elsewhere) 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. I will elide my C reconstruction (derided as "fantasy code") of this disassembly herein; it can be seen in the RYBKA_FRUIT document.

One can note the identical condition for `easy` to be used (a difference of 150), while the conditions with `bad_2` and such merely have the percentages modified slightly (20% becomes 1/6 for `easy`). The setting of `last_value` equal to `value` occurs at the same point in these routines, the `bad` flags are only updated when `depth` exceeds 1, the latter conditional checks are all arbitrarily in the same order (both the high-level ordering of checking depth-limited, then `bad_2`, then `easy`, then `early`, and also the conditions inside each of these) as in Fruit 2.1, etc. The probative aspects are clear; the major "difference" is perhaps that Rybka does have a check for mate scores (±0x7d00).

Indeed, already the six variables used are rather idiosyncratic to Fruit 2.1 (note that Fruit 1.0 differs, as do later Rybka versions). Furthermore, the variables are allocated in the same order in Rybka 1.0 Beta here are allocated in exactly the same order as in the comparative Fruit 2.1 code (see `search.h`):[30]

```
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 };
```

One concludes that the Rybka 1.0 Beta code here originated in Fruit 2.1. The substantive elements are the characteristic (to Fruit 2.1) use of `bad` and `change` parameters to decide when the search should be stopped, and also the same `easy` margin of 150. The minor variations of percentages in the Rybka 1.0 Beta are insignificant compared to this.[31]

4.4 Other sundry elements of commonality

4.4.1 PST tables

As noted above, the PST tables in Rybka 1.0 Beta can be reproduced exactly via an abnormally small variation from the code in Fruit 2.1. Since this is a bit tangential and voluminous to discuss, I simply refer to this link.

Similarly, the PST scheme in Rybka 1.6.1 does not follow the same pattern.

In this regard, one can note that (for instance) the Fruit 1.0 PST tables use much the same "general concept", but require many more code changes from the Fruit 2.1 code to be reproduced exactly.[32]

[30] Beyond this ordering, there is no particular reason to group these six elements together in the first place, as Rybka and Fruit do.

[31] While many computer chess programs might have something "similar" to this for deciding when to stop a search, the specific methods of Fruit 2.1 are used in Rybka 1.0 Beta, and seemingly nowhere else.

[32] It has occurred to me that this same "code-change" metric can be applied to the evaluation functions, though there one would have to consider whether "functional equivalence" (same outputs) would suffice, or whether the method to produce said result should also be of import. Due to the vast differences in (say) internal representation of the chess board in various engines, I would argue that the method should be ignored if such a comparison were to be made.

4.4.2 Data structures with hashing

The hashing structure of Fruit 2.1 is rather charactertistic of it, and it re-appears with inessential modifications in Rybka 1.0 Beta (again note that Rybka 1.6.1 differed; the structure was kept until Rybka 2.3.1 switched to a 64-bit format).

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. 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 the corresponding elements therein as [hash, depth, score, move] with differing bit widths.

For the latter 8 bytes, Rybka 1.0 Beta has the same fields as Fruit 2.1, but re-ordered in batches of 4. The Rybka 1.0 Beta structure has:

- 2 bytes for min value, 2 bytes for max value, a byte for move depth, an unused byte, a byte for min depth, and a byte for max depth.

The Fruit 2.1 structure is:

- a byte for move depth, an unused byte (called "flags"), a byte for min depth, a byte for max depth, 2 bytes for min value, 2 bytes for max value.

As can be seen, Rybka 1.0 Beta merely switches bytes 8-11 with bytes 12-15.[33]

4.4.3 Et cetera

Another minor element might be the somewhat non-standard 1-2-4 scaling for "game phase" computations, whereas Fruit 1.0 used 3-5-9 (which is a common weighting of chess pieces). A list of happenstances such as this could be readily extended (for instance, 10-30-60-100 scaling at various junctures).

4.5 Rybka 1.0 Beta a continuation of Rybka 1.6.1?

It is unclear whether (or to what extent) Rajlich considers Rybka 1.0 Beta to be a continuation of Rybka 1.6.1. Private email correspondence with Zach Wegner indicates the affirmative, as do his TalkChess postings (in late 2005, to Daniel Mehrmann asserting that the Rybka source code was original and pre-dated all the Fruit releases, and to Nalimov regarding use of his tablebase code).

However, the internal comparison indicates that Rybka 1.0 Beta shares little in common with Rybka 1.6.1, even less than Fruit 1.0 and Fruit 2.1 share (say). For instance, the piece numbering in Rybka 1.0 Beta is the same as that of Fruit 2.1 (which differs slightly from Fruit 1.0), while that of Rybka 1.6.1 is the same as that of Crafty. Similarly, the bit-packing of moves with Rybka 1.0 Beta is no longer akin to Crafty, but follows Fruit.

The same is true with the UCI parsing[34], the hash structure, the underpromotion functionality in Rybka 1.6.1 that was missing in the later Rybka 1.0 Beta,

[33] There is also an atypical commonality in the use of both lower/upper bounds in a PVS engine. Perhaps this could occur because Rybka 1.6.1 used MTD(f) at one point, though the evidence does not exemplify this.

[34] E.g., as noted previously, Rybka 1.0 Beta suddenly drops the `searchmoves` implementation that Rybka 1.6.1 possessed.

and more.[35] For instance, it seems rather inscrutable that Rybka 1.6.1 had pondering, but the "successor" Rybka 1.0 Beta did not. The notable similarities of UCI and time management code make it quite likely that these were adapted from Fruit 2.1, rather than coming from Rybka 1.6.1.

Given then, that for something for which he already had his own working code Rajlich was willing to borrow from Fruit 2.1, it is a difficult argument to make that other and more substantive "Fruit-like" parts in Rybka 1.0 Beta were merely "Fruit influenced" rather than lifted more directly.

4.6 Conclusion

Rybka 1.0 Beta had its origins in Fruit 2.1 in many aspects, both probative and substantive. For instance, its choice of evaluation features is substantially similar to that of Fruit 2.1. The same is true (to a slightly lesser extent) with Rybka 2.3.2a. With Rybka 1.0 Beta, there are multiple other places of unwarranted congruence to Fruit 2.1, all of which diverge distinctly from the comparative aspects of Rybka 1.6.1. Furthermore, any claim that Rybka 1.0 Beta is a "continuation" of Rybka 1.6.1 (or other pre-Fruit version) in the typical meaning of this word is not apparent from the evidence.

5 The verdict, and some final comments

The ICGA Board (listed above) was responsible for judging the evidence and giving a sentence. They chose to disqualify all of Rajlich's entries (even those post-dating the claims above), on the basis of "plagiarism" contravening their rule regarding "originality". They also imposed a lifetime ban against him from entering their tournaments.[36]

To exemplify the nature of complaints about and against the ICGA process, I will relate the following story. Back in 2006, the ICGA disqualified LION++ on the grounds that it failed to adequately cite Fruit, as it did so only in its program notes, not on the entry form. Despite this interpretation, some Rybka defenders have contended that writing `www.rybkachess.com` on the entry form suffices for Rajlich, since Letouzey is thanked (rather inspecifically) in a README that can be downloaded from this website. But even assuming this credit to be enough, there is still one problem: *This README does **not** appear in the (March 2006) Rybka 1.0 Free Download available from said website, but rather was only distributed when the initial Beta was free in its first week (Dec 2005).* And so, the very premise is seen to be false, strained at it is. Yet, having failed to even make a basic fact-check on their theory, the defenders will likely just scupper this point, and pass on to the next least likely proposal... Hopefully this indicates to some extent why most current Rybka claims are simply ignored.

5.1 Why it took 5+ years

There are many reasons that the investigation of Rybka took 5 years, and I catalogue some of them here. Firstly one must note Rajlich's brazen assurance that Rybka was indeed (all) his own work, as can be noted by the above-mentioned statement to Daniel Mehrmann, and also one (a few days later) to Andrew Wagner, saying: *As far as I know, Rybka has a very original search and evaluation framework.*[37] Eventually (and in retrospect) these were seen to be more of signs of duplicity, but to at least some extent there is a still a "gentleman's" nature to computer chess programming, and so his word was accepted.

[35] These last two seem related to the aforementioned bit-packing, as the scheme in Crafty [and too Rybka 1.6.1] used 21 bits for a move, while the Fruit 2.1 hashing structure restricted a move to 16 bits.

[36] This "lifetime ban" is also presumably liftable if Rajlich cares to address the matters herein to the satisfaction of the ICGA. There is also the "repeat offender" aspect, with Rybka/Crafty taken into account; though such versions did not compete in any ICGA events, Rajlich did enter them into author-based tournaments.

[37] The word "framework" here is particularly dubious with the EVAL_COMP evidence.

Secondly, Rybka quickly made progress, gaining about 100 Elo in the five months about the Rybka 1.0 Beta release. Combined with Rajlich's quite knowledgeable discussions at TalkChess and his quasi-celebrity status as an International Master, accusations thus tended to be dismissed as farfarel. No one at the WCCC in 2006 (where Rajlich participated as Rajlich, rather than under the Rybka name) found there to be sufficient reason to make a formal complaint.[38]

Furthermore, due to the amount of resources needed to investigate a program via reverse engineering, there was no explicit link to Fruit 2.1 until about mid-2007 at the earliest, and then it was only of a spurious nature from somebody (Yuri Osipov) who seemed to trying to cover up his own Rybka copying (with Strelka).[39] Indeed, when Letouzey was contacted (by Dann Corbit) in April 2008 regarding the Strelka/Fruit situation, there was not even a mention of a possible Fruit/Rybka link, and the presentations of disassembled code by Rick Fadden at that time similarly focused on Rybka/Strelka rather than Fruit.

Zach Wegner was the first to muster courage to explicitly state that Rybka 1.0 Beta had many questionable aspects that appeared to derive from Fruit 2.1 – this was August 2008, which coincided exactly with the time of the Rybka 3 release (another 100 Elo gain), and Wegner was largely shouted down,[40] with his evidence still in a preliminary state.[41] I myself found Wegner's analysis to be of interest, and researched the subject on-and-off over the next couple of years. The issue largely faded away, being only partially revived when IPPOLIT (which Rajlich claimed was derivative of Rybka) became public in late 2009.[42]

It was really not until Fabien Letouzey reappeared (being finally been informed of a Fruit link with *Rybka*) in early 2011 and made a specific complaint to the ICGA that anything was done. By that time, between Wegner and myself (and others) there was enough evidence assembled for other programmers to be convinced that something was amiss, and that further investigation was needed. This was then carried out in the ICGA Panel as described above.

Through all this, Rajlich never really addressed the issues as they arose, but rather gave incomplete and somewhat evasive answers to any inquiries. Indeed, my personal impression (partially retrospectively) is that he was always trying to

[38] Given that Rybka had gained 100 Elo from the 1.0 Beta release (over the period of Dec 2005 to June 2006, with much of this gain already seen in Rybka 1.1 of March 16), and the general buzz about Fruit 2.1 influence, one might be led to assume that Rajlich had rectified any derelictions in this area (via improvements or code changes). However, such an assumption (in retrospect) seems incorrect, as almost the totality of Rybka changes were in search heuristics for quite some time, the above evidence indicating the evaluation function similarity to Fruit persisted through Rybka 2.3.2a (June 2007).

[39] Osipov speculated that Rajlich took Fruit as a basis and rewrote it. Rajlich directly addressed this by stating: *Rybka is and always was completely original code* (noting obvious exceptions, such as public domain). He also ridiculed a "cloner" claiming the same of him.

[40] There was also some thinly embellished innuendo that his reputation would be shot forever if he pursued this and turned out to be wrong.

[41] It could be of interest to note that there were some rumblings that Rybka 3 should be checked out by the ICGA before being allowed to compete in the WCCC later that year, but that other commercials (Rybka competitors, no less) were notable in their objection to this without there being a particular complaint with suitable backing to its merits.

[42] Wegner seemed too busy with school and work. I had similar outside time constraints, and was more immediately interested in Rybka/IPPOLIT when I first became more involved.

"buy time", hoping that the storm would pass by and leave him unharmed. He has been, if nothing else, quite notably consistent in his claim that Rybka "is and always was completely original code," even when the evidence demonstrates much the contrary (except at the most literal level, perhaps).

Indeed, Rajlich's current position appears to be that Rybka 1.0 Beta was (much) influenced by Fruit 2.1, but that whatever was taken was done "legally", and was perhaps not of much value. The Panel investigation did not address the issue of legality *per se*, but did opine that Rybka was much more than merely "influenced" by Fruit, but rather had its "origins" in Fruit. The ICGA Board concurred with this in their verdict. Furthermore, substantial and significant parts of Fruit remained in Rybka at least through Rybka 2.3.2a (June 2007), and these were sufficient for Rajlich's entries into the ICGA tournaments to amount to (partial) misappropriation of Letouzey's work.

5.2 Rajlich's (minimal) defense, and move selection

Rajlich offered only a few brief words to the ICGA Board. He stated that Zach Wegner's code reconstruction for PST tables was "bogus", queried what rule he was thought to have broken, and directed Levy to the CCRL `ponderhit` data to evince (it seems) that Rybka 1.0 Beta did not possess any great overlap of move selection when compared to Fruit 2.1.[43]

However, this is a poor metric of comparison when trying to determine originality (particularly when disproving a link between engines),[44] and is at best rather indirect. One can easily change (say) half of a program and get quite different move selections. The other half would still be taken from someone else. Also, given the methodology chosen by the CCRL, one could presumably reduce `ponderhit` correlation simply by speeding up an engine via "engineering methods", but the result would retain much the same creative content.[45]

With Dailey and Hair, I have now published a paper that discusses issues of move similarity in more detail.

A superior method of "move selection" correlation has been championed by Don Dailey and Adam Hair. Even with their method, one can presumably reduce a "definite clone" reading to a "unsure, seek additional help" prognosis by (say) perturbing the evaluation function by enough centipawns, or making Elo-agnostic changes to something like check-extensions in quiescence search.

Such methods of detection will at best remain a "first signal" of copying, and will always remain secondary to such methods as actual code comparison.

[43] Levy responded by noting that the ICGA Rules (as interpreted in the LION++ case from 2006) gave such commonality of move selection as merely one example [and indeed, in somewhat of an injunctive sense to warrant further investigation] of a basis for nonoriginality, and again asked Rajlich to address the material contained in the Panel Report.

[44] Even the ability to show "cloning" from the CCRL data is debatable. Although their numbers are indeed of interest, there are a number of caveats (e.g., they exclude drawn games completely, and also "instant moves", which can be quite engine-dependent). I was also unable to determine, via a short look, what the "expected" `ponderhit` correlation should be. It seemed that 60% with 3% standard deviation was a crude estimate; though the distributional aspects need thought, only a few engine pairs could be said to be "outliers" of this. As an aside, the idea of trying to match move selection to another entity dates at least back to Deep Thought.

[45] While such methods might not amount to very much in general, it could be more notable in the case of Fruit, for Letouzey himself admits that Fruit 2.1 was sort of a development "snapshot", and much was done to improve various efficiency aspects in the 2-month interim leading up to the 2005 WCCC.