YAP3: Improved Detection Of Similarities In Computer Program And Other Texts

Article *in* ACM SIGCSE Bulletin · April 1996

DOI: 10.1145/236452.236525 · Source: CiteSeer

YAP3: IMPROVED DETECTION OF SIMILARITIES IN COMPUTER PROGRAM AND OTHER TEXTS[1]

Michael J. Wise
Department of Computer Science,
University of Sydney, Australia

michaelw@cs.su.oz.au

ABSTRACT

In spite of years of effort, plagiarism in student assignment submissions still causes considerable difficulties for course designers; if students' work is not their own, how can anyone be certain they have learnt anything? YAP is a system for detecting suspected plagiarism in computer programs and other texts submitted by students. The paper reviews YAP3, the third version of YAP, focusing on its novel underlying algorithm - *Running-Karp-Rabin Greedy-String-Tiling* (or *RKS-GST*), whose development arose from the observation with YAP and other systems that students shuffle independent code segments. YAP3 is able to detect transposed subsequences, and is less perturbed by spurious additional statements. The paper concludes with a discussion of recent extension of YAP to English texts, further illustrating the flexibility of the YAP approach.

1. INTRODUCTION

The first version of YAP, a *structure-metric* system for detecting suspected plagiarism in computer program texts is described in [9]. That paper also reviews other, *counting-metric* systems described in the literature, and a structure-metric system similar to YAP, *Plague* [8]. A companion paper to this, [7], reports on an extensive comparison between the current version of YAP, YAP3, and reconstructions of two systems reported in the literature: Grier's *Accuse* system [3] and Faidhi and Robinson's system [1].

This paper reviews YAP3, the third version of YAP, focusing on its novel underlying algorithm - *Running-Karp-Rabin Greedy-String-Tiling* (or *RKS-GST*), whose development arose from experience with YAP which indicated that, not only are students adept at breaking in-line code into functions (and vice versa), but they are also adept at shuffling independent code segments. Consider the segment of C code below, which comes from the main function of a program implementing simple operations on a queue:

```
while(fgets(line,MAXLINE,stdin)!=NULL)
 {
 if(sscanf(line,"%s",word)!=1)
  continue;
 switch(word[0])
  {
  case'r':
   printf("Removed:%d\n",remove_item());
   break;
  case'a':
   if(sscanf(line,"%*s%d",&value)!=1)
    {
    fprintf(stderr,"readofitemfails\n");
    continue;
    }
   add_item(value);
   break;
  case'p':
   print_queue();;
   break;
  default:
   fprintf(stderr,"unknown%c\n",word[0]);
  }
 print_queue();
 }
```

It is trivial for a student to change the while to a for, to break single printf statements into multiple statements or to add a break in the default case. It is also trivial to swap the cases in the switch.

YAP3, as with YAP, works in two phases. In the first, the source texts are used to generate token sequences. This phase, which is largely the same as in YAP, involves:

- Removing comments and string-constants
- Translating upper-case letters to lower case
- Mapping of synonyms to a common form, e.g. `strncmp` is mapped to `strcmp`, and `function` is mapped to `procedure`.

1. Presented at SIGCSE'96, Philadelphia, U. S. A., Feb. 15–17, 1996, pp 130-134.

- Reordering the functions into their calling order. In the process, the first call to each function is expanded to its full token sequence; Subsequent calls are replaced by the token `FUN`. Notice, however, that expanding functions (as also done by Plague) exacerbates the effect of transpositions as illustrated above.
- Removing all tokens that are not from the lexicon of the target language, i.e. any token that is not a reserved word, built-in function, etc.

While there have been some improvements to the tokenizers described in [9], they are largely unchanged except that from YAP3 onwards, the tokens are numerical rather than being strings.

What distinguishes the three versions of YAP are the algorithms used in the second, comparison phase, where each token string is (non-redundantly) compared with all the others. There are also differences in the implementation techniques.

As reported in [9], the comparison phase of the original version of YAP is based on the UNIX utility sdiff, whose underlying algorithm is the dynamic programming solution to the Longest Common Subsequence problem. Apart from the fact that YAP is implemented as a Bourne Shell script, and therefore rather slow, a major problem is that, in the dynamic programming solution to the LCS, a translocation of a block of tokens from one part of a string to another may be missed entirely or regarded as a series of individual transpositions rather than a single block move. An alternate formulation of the problem as the Levenshtein metric – or minimum edit distance, the minimal sequence of single token insertions and deletions required to transform one string into the other [6] – has essentially the same problem. The problem arises because both the Levenshtein metric and the LCS algorithm are order preserving so both algorithms deal with single token differences by either signalling an insertion/deletion (Levenshtein) or by skipping over the extraneous elements (LCS). This problem has been noted previously by Heckel [4], who proposes an alternate algorithm. Plague offers a variant on the LCS algorithm as one of its metrics.

YAP2, implemented in Perl and distributed since the end of 1992, uses Heckel's algorithm [4] (which is also offered by Plague). Heckel's algorithm was designed to work with text files. In this algorithm, instances of unique lines common to two files are notionally joined. Then, lines above the joined lines are examined; if they are the same these lines are joined, until a non-match is found. The process is repeated for lines below the joined lines. Thus blocks are formed, the total length of which can serve as a measure of similarity. Although Heckel's algorithm requires several passes, its overall complexity is linear. Furthermore, it is also able to deal effectively with transposed code segments. However, the problem exists of finding the initial set of unique, common lines because Heckel's "lines" are, in this case, simply tokens drawn from an alphabet whose usage is highly skewed.[2] To overcome this problem, tokens may be rewritten as overlapping groups of three, i.e. first, second, third in the first group, second, third and fourth in the second group, and so on. However, this is only a partial solution because there may fail to be any unique tokens or token-groups in two otherwise identical strings. There is another significant problem: because of the way blocks are formed, Heckel's algorithm tends to produce a small number of long blocks (substrings). These long substrings are trivially broken by the introduction of a small number of spurious tokens, e.g. extra `print` statements or the decomposition of a compound assignment-statement into several simpler ones. However, once such a break has occurred it is often the case that no further blocks can be formed from the remaining portions of the strings because there are no unique token-groups to anchor the blocks. What one therefore sees is a steep decline in the extent to which two strings are believed to match.

2. YAP3

Given the difficulties faced by the well-known algorithms in detecting similarities in the presence of block-moves, a total new algorithm, Running-Karp-Rabin Greedy-String-Tiling (RKR-GST), was devised as the basis for the third version of YAP.

The RKR-GST algorithm is based on the notion of a *tile*, which is an indisoluable, one-to-one pairing of a substring from one string (arbitrarily called *P*, the *pattern*) and a substring from the second string (*T*, the *text*); once a token becomes part of a tile, it is said to *marked*. There is also the notion of a *maximal match*, which, like tiles, is a pairing of substrings, but in this case the pairing may only be temporary. Finally, there is the notion of a *minimum-match-length*, which is the minimum length of tiles being sought; potential tiles below this length are ignored as possible artefacts of the matching process. (The default minimum-match-length is 3.)

Ideally what is being sought by the new algorithm is a maximal tiling of P and T, i.e. a coverage of non-overlapping substrings of T with non-overlapping substrings of P which, bearing in mind the

2. Although the lexicons may be large (around 550 for C, when all the libraries are included, and 350 for Lisp), usage statistics for the tokens are highly skewed; the top four tokens by usage can account for more than 50% of occurrences. In particular, there can be long sequences of the same token, notably = (as distinct from =) in C or := in Pascal.

minimum-match-length, maximises the number of tokens that have been covered by tiles. Unfortunately, an algorithm which produces a maximal tiling and computes in polynomial time is an open problem. Part of the difficulty lies in the possibility that several small tiles could collectively cover more tokens than a smaller number of larger tiles.

To motivate the transition to a computationally more reasonable measure of similarity it is worth observing that longer tiles are preferable to shorter ones because long tiles are more likely to reflect significant, similar regions in the source texts rather than chance similarities. With this in mind, the following greedy algorithm is proposed. The algorithm involves multiple passes, each pass having two phases. In the first phase, called *scanpattern*, all maximal-matches of a certain size or greater are collected. This size is called the *search length*. In the second phase, called *markstrings*, maximal-matches are taken, one at a time starting with the longest and tested to see whether they have been occluded by a sibling tile (i.e. part of the maximal-match is already marked). If not, a tile is created by marking the two substrings. When all the maximal-matches have been dealt with, a new, smaller search length is chosen. The top-level algorithm is:

```
search-length s := initial-search-length;
stop := false
Repeat
  /*L_max is size of largest maximal-matches from this scan*/
  L_max := scanpattern(s);
  /*If very long string no tiles marked. Iterate with larger s*/
  if L_max > 2 × s then s := L_max
  else
    /* Create tiles from matches taken from list of queues*/
    markstrings(s);
    if s > 2×minimum_match_length then s := s div 2
    else if s > minimum_match_length then
      s := minimum_match_length
    else stop := true
until stop
```

So far, what has been described relates solely to Greedy String-Tiling. It is in the algorithm for `scanpattern` that Running Karp-Rabin matching [5] is used to find all the maximal matches above a certain size. (The version of the recurrence relation used in this work is due to Gonnet and Baeza-Yates [2].) Running Karp-Rabin matching extends Karp-Rabin matching in the following ways:

Firstly, instead of having a single hash-value for the entire pattern string, a hash-value is created for each (unmarked) substring of length s of the pattern string, i.e. for the substrings $P_p \cdots P_{p+s-1}$, p in the range $1 \cdots |P|-s$. Hash-values are similarly created for each (unmarked) substring of the length s of the text string. Secondly, each of the hash-values for the pattern string is then compared with the hash-values for text string and for those pairs of pattern and text hash-values found to be equal, there are possible matches between the corresponding pattern and text substrings. A hash-table of the text-string Karp-Rabin hash-values is used to reduce the otherwise O(n^2) cost of this comparison. That is, rather than having to scan the entire text string for the matching hash-value corresponding to a particular pattern substring, the pattern Karp-Rabin hash-value is itself hashed and a hash-table search returns the starting positions of all text substrings (of length s) with the same Karp-Rabin hash-value. Note that after a successful match of both the Karp-Rabin hash-values and the actual substrings, the element-by-element matching continues until two elements fail to match or until a marked element or end-of-string are found. In this way, the matches are converted to maximal-matches. (In other words, length s is the minimum match-length being sought during one iteration.) The algorithm for `scanpattern(s)` – s the current search-length – is:

```
starting at the first unmarked token of T,
 for each unmarked T_t do
   if distance to next tile ≤ s then
     move t to first unmarked token after next tile
   else
     create KR hash-value for substring T_t..T_{t+s-1}
     add to hashtable

Starting at the first unmarked token of P,
 for each unmarked P_p do
   if distance to next tile ≤ s then
     move p to first unmarked token after next tile
   else
     create KR hash-value for substring P_p..P_{p+s-1}
     check hashtable for P_p..P_{p+s-1} KR hash-value
     for each hit in hash-table do
       k: = s
       /*Extend match until non-match or element marked*/
       while P_{p+k} = T_{t+k}
             AND unmarked(P_{p+k})
             AND unmarked(T_{t+k}) do
         k := k + 1
       if k > 2×s then
       /*abandon scan. Will restart with s = k*/
         return(k)
       else record new maximal-match
return(length of longest maximal-match)
```

The structure used to record the maximal matches is a doubly-linked-list of queues, where each queue records maximal-matches of the same length and the list of queues

is ordered by decreasing length. A pointer is also kept to the queue onto which the most recent maximal-match was appended because there is a high probability that the next maximal-match will be similar in length to the last and therefore will be appended to the same queue or one that is close by. `markstrings` also has the parameter s, the search-length.

```
starting with the top-queue do
 if current queue is empty then drop to next queue
 /*corresponds to smaller maximal-matches */
 else with queue of maximal-matches length L do
  remove match(p, t, L) from queue
  if match not occluded then
   if for all j:0..s−1, P_{p+j} = T_{t+j} then
   /* IE match is not hash artefact */
    for j:= 0 to L - 1 do
     mark_token(P_{p+j})
     mark_token(T_{t+j})
     count_tokens_tiled := count_tokens_tiled + L
  else if L − L_occluded ≥ s then
   replace unmarked portion on list of queues
```

Note that the test of whether maximal-match is really a match has been deferred from `scanpattern` – where it normally would reside – to `markstrings`. (Remember that all putative matches found due to hashing must be tested element-by-element because equivalence of hash-values does not guarantee that the corresponding strings are equal. However, it has been observed that KR-hashing appears to fail so rarely that deferring the component-wise test to `markstrings` turns out to be far more efficient [10].)

The question arises as to what is an appropriate value for the parameter s passed to `scanpattern` and `markstrings`. More precisely, what is to be its initial value and how is that value to be decremented? While one might consider half the length of P as an appropriate starting value for s, it turns out in practice that a much smaller value will suffice. There are two reasons for this. Firstly, very long maximal-matches are rare, so in general a large initial value for s would generate a number empty sweeps by scanpattern until a match is finally found. Secondly, if a long maximal-match is found ("long maximal-match" defined to be where k, the maximal-match length is $2\times$s) the creation of a tile from this string will absorb a significant number of the pattern and text tokens. It is therefore worthwhile stopping the current scan and restarting with the larger initial value of s = k for this special case. This implies that the initial search-length can be set to a small constant value (it is currently 20), rather than being dependent on the string lengths.

Finally, in [10] it is argued that although the worst-case complexity is $O(n^3)$ – n the sum of the lengths of the input strings – the circumstances where that arises are entirely pathological and using curve-fitting a complexity of $O(n^{1.12})$ is shown experimentally, i.e. close to linear.

3. COMPARING THE YAP VERSIONS

To illustrate the increased performance provided by YAP3 compared to its predecessors, YAP2 and YAP, two versions of the previous small example (run as a completed program, Q.c) have been compared. In version Q1.c, all that has been done is to reorder the cases in the switch statement so that case 'a' is followed by case 'p' and then case 'r'; the rest is the same. In Q2.c, the other functions are the same but main has been altered as follows:

```
while(fgets(line,MAXLINE,stdin)!=NULL)
 {
 if(sscanf(line,"%s",word)!=1)
  continue;
 switch(ch=word[0])
  {
  case 'r':
   value=remove_item();
   printf("Removed:%d\n",value);
   break;
  case 'a':
   if(sscanf(line,"%*s%d",&value)==1)
    {
    add_item(value);
    break;
    }
   fprintf(stderr,"readofitemfails\n");
   continue;
  case 'p':
   print_queue();
   break;
  default:
   fprintf(stderr,"unknown%c\n",word[0]);
  }
 }
```

The percent-match values are:

	Q/Q1	Q/Q2	Q1/Q2
*Yap*1	73	55	65
*Yap*2	98	56	62
Yap.3^3	97	69	69
Yap.3^2	100	75	75

Yap.3^3 is YAP3 with minimum-match-length set to 3; in *Yap*.3^2 it is set to 2 via a command-line argument. The latter is justified in shorter token sequences (this one is around 60) because there is a greater tendency for the string to be broken into unplaceable

fragments, and each will have a disproportionate effect. In reading the above table, specific values are not important; rather note the effect text alterations have on the final value.

4. TOKENIZING ENGLISH

The strength of the YAP approach is that is does not attempt a full parse of the target language, as does Plague, but rather compares token strings made up of keywords drawn from the target language's lexicon. This greatly simplifies the task of porting the system to new computer languages, especially for relatively complex languages such as C (versus Pascal). However, in the case of English, obtaining a full parse would be practically impossible and it is here that the YAP approach becomes invaluable. What has been done is to create a tokenizer-generator, whose first pass across a number of texts determines the lexicon which is then used to generate the token strings. The first, lexicon-generating, pass eliminates numbers, instances of the 150 most common English words, words consisting of one or two letters and proper nouns. The remaining words are stemmed using the PC-Kimmo (Version 1.08) recognizer (slightly modified for this purpose) and the Englex10 rule and lexicon sets.[3] The tokenizer-generator and the resulting tokenizer were applied with YAP3 to a set of short essays submitted in a Operating Systems class. While serviceable token strings were generated, no significant plagiarism was reported. This may have something to do with the fact that YAP3 had been applied to two earlier assignments, resulting in some students not being awarded any marks for those assignments. Nonetheless, the ability to generate tokenizers on-the-fly and then to use the YAP3 comparison phase unaltered, illustrates the flexibility of the YAP approach. Indeed, the underlying algorithm is the basis for *Neweyes*, a system for aligning nucleotide or amino acid sequences [11].

5. REFERENCES

[1] FAIDHI, J. A. W. AND S. K. ROBINSON, ''An Empirical Approach for Detecting Program Similarity within a University Programming Environment'', *Computers and Education* **11**(1), pp. 11-19 (1987).

[2] GONNET, G. H. AND R. BAEZA-YATES, *Handbook of Algorithms and Data Structures (Second Edition)*, Addison-Wesley (1991).

[3] GRIER, SAM, ''A Tool that Detects Plagiarism in Pascal Programs'', *Twelfth SIGCSE Technical Symposium*, St Louis, Missouri, pp. 15–20 (February 26-27, 1981) (SIGCSE Bulletin Vol. 13, No. 1, February 1981).

[4] HECKEL, PAUL, ''A Technique for Isolating Differences Between Files'', *Communications of the ACM* **21**(4), pp. 264–268 (April 1978).

[5] KARP, RICHARD M. AND MICHAEL O. RABIN, ''Efficient Randomized Pattern-Matching Algorithms'', *IBM Journal of Research and Development* **31**(2), pp. 249–260 (March 1987).

[6] KRUSKAL, JOSEPH B., ''An Overview of Sequence Comparison'', *Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison*, ed. David Sankoff and Joseph B. Kruskal, pp. 1–44, Addison Wesley (1983) (Chapter 1).

[7] VERCO, KRISTINA L. AND MICHAEL J. WISE, ''Software for Detecting Suspected Plagiarism: A Comparison'', *Communications of the ACM* (Submitted to journal).

[8] WHALE, G., ''Identification of Program Similarity in Large Populations'', *The Computer Journal* **33**(2), pp. 140–146 (1990).

[9] WISE, MICHAEL J, ''Detection of Similarities in Student Programs: YAP'ing may be Preferable to Plague'ing'', *Twenty-Third SIGCSE Technical Symposium*, Kansas City, USA, pp. 268–271 (March 5–6, 1992).

[10] WISE, MICHAEL J, ''Running Karp–Rabin Matching and Greedy String Tiling'', Basser Department of Computer Science Technical Report, Sydney University (1994) (ftp://ftp.cs.su.oz.au/michaelw/rkr_gst.ps Revises Basser Technical Report 463, March 1993).

[11] WISE, MICHAEL J, ''Neweyes: A System for Comparing Biological Sequences Using the Running Karp-Rabin Greedy String-Tiling Algorithm'', *Third International Conference on Intelligent Systems for Molecular Biology*, Cambridge, England., ed. Christopher Rawlings, Dominic Clark, Russ Altman et. al., pp. 393–401, AAAI Press (July 16-19, 1995).

3. Both PC-Kimmo and the Englex10 rule and lexicon sets are available from the Consortium for Lexical Research, clr.nmsu.edu. A Postscript catalogue off all tools held by CLR is at: CLR/catalog.ps.