String Similarity via Greedy String Tiling and Running Karp–Rabin Matching[1]

Michael J. Wise[2]
Department of Computer Science,
University of Sydney, Australia

michaelw@cs.usyd.edu.au

December 1993

Abstract

A new method, Greedy String Tiling (GST), is proposed for comparing pairs of strings (and hence files) to determine the degree to which they are similar. Greedy String Tiling is based on one-to-one matching, and is able to deal with transposition of substrings. With certain restrictions, GST is a true metric; otherwise an approximation to the metric is found. A novel algorithm is proposed for computing GST values. This algorithms, Running Karp-Rabin matching has a experimentally derived average complexity that is close to linear. The original area of application is the detection of possible plagiarism in computer programs and other texts. However, other areas of application are also being investigated, in particular the comparison of DNA and amino acid sequences.

Keywords: longest common substring, longest common subsequence, string/file comparison metric, Karp-Rabin string matching, plagiarism detection, biological sequence comparison

1. Introduction

The following string matching problem arose from recent work on a system for the automated detection of possible plagiarism, though it has application to other area, e.g. comparison of DNA/protein sequences.

An earlier paper, [Wise92], discusses the first version of a system for detecting potential plagiarism in computer programs and other texts. The system, which has been called YAP, works in two stages. In the first stage, YAP transforms each text into a sequence of tokens, where the tokens are drawn from a lexicon of "significant" words (e.g. keywords and built-in functions of a computer language). What is required in the second stage (and the subject of this article) is a measure for similarity between the token strings, with a high value indicating possible plagiarism.

An algorithm to measure string similarity in this domain must have the following properties:

- Each token in either string must either be counted once (if it matches a token in the other string), or not at all, because it is assumed that procedures have already been expanded during tokenization − in the case of computer languages − or that the notion of procedures/subroutines is irrelevant − in the case of natural language texts.
- Transposed substrings should have a minimal effect on the similarity value because transpositions can only occur when they do not cause a change in the semantics of the source text, .e.g. swapping of cases in a `switch` statement or rearrangement of paragraphs. Stated anther way, if a portion of text is to be swapped with another, the change must be semantically neutral, or other changes must accompany the change in position.
- The value returned must degrade gracefully in the presence of random insertions or deletions of tokens. For example, breaking a statement involving a compound expression into several statements involving simple expressions should not cause the similarity value to fall, or if a decrease in the similarity value does occur, it should not be disproportionate.

---

2. The paper was originally written in December 1993. It has not been published in this form, but implementations of the algorithm have been appeared in the literature: [Wise95] describes Neweyes, a novel system for comparing DNA or protein sequences, while [Wise96] describes YAP3, the third version of a system for detecting suspected plagiarism in student assignments.

2. Author's present address: CCSR, 10 Downing Street, Cambridge CB2 3DS, England.

Definition. When referring to two strings, the shorter of the two will be referred to as the *pattern* string while the longer will be referred to as the *text* string.

There have been two main approaches for measuring the similarity between two strings: Levenshtein distances and Longest Common Subsequences. The Levenshtein distance between two strings (also known as the edit distance, and described in [Krus83] is length of the minimal sequence of single token insertions and deletions that would be required to transform the pattern string into the text string. Identical strings would have a Levenshtein distance of size 0.

The Longest Common Subsequence approach is in many ways similar to Levenshtein distance, but rather than building up one sequence from the other, it finds the longest sequence of tokens common to the two strings. There are many discussions of LCS in the literature (e.g. [Corm90] or [Gonn91]), but in summary, a subsequence of string S is formed by taking elements in order from S, where zero or more elements are ignored before another is taken. The length of the LCS for identical strings is the size of one string.

Algorithms based on either approach are able to deal effectively with single token differences by either signalling an insertion/deletion (Levenshtein) or by skipping over the extraneous elements (LCS). However, both the Levenshtein distance algorithm and the LCS algorithm are order preserving, so transposed substrings will generate a multitude of single line differences rather than being seen as a block move. This problem has been noted previously by Heckel [Heck78] and Tichy [Tich84], who propose alternate algorithms.

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.[3] 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, but 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.

Another alternative to LCS is provided by Tichy [Tich84]. Code for Tichy's algorithm as program `bdiff` can be found in Miller [Mill87]. The central idea of Tichy's algorithm is the transformation of one file into another by a series of block copies (when a block of lines of one file is found in the second file), or single line appends, when a line in the source is not found in the target it is appended. Tichy shows that what is produced is a minimal length edit-sequence, i.e. the minimum number of move/append instructions needed to convert one file into a second file. (The algorithm was intended for use in the code-control systems RCS, but does not appear to generally be used in that role; most implementations of RCS use UNIX `diff`.) Tichy's algorithm is able to deal effectively with block moves, but if the metric is related to the length of the edit-sequence, a single line difference will have the same weight as a block move. The main problem with this approach from the point of view of the current discussion is the fact that blocks (or substrings in the current application) may be double-counted. That is, if the first string is the letters `abcabc` and the second string is the letters `abc`, the first string will be regarded as two copies of the second, rather than one copy followed by three new items.

---

3. 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.

2. Greedy String Tiling

In this section a new similarity measure will be proposed where, like LCS, the largest possible value will be the size of one string (in the case of identical strings), or 0, in the case where the strings do not match at all.

Let P be the pattern (i.e. shorter) string and T the text string.

Definition. A *maximal-match* is where a substring P_p of the pattern string starting at p, matches, element by element, a substring T_t of the text string starting at t. The match is assumed to be as long as possible, i.e until a non-match or end-of-string are encountered, or until one of the elements is found to be *marked*. (Marking will be defined presently.) A maximal-match is denoted by the triple max_match(p, t, s), where s is the length of the match. Maximal-matches are temporary and possibly not unique associations, i.e. a substring involved in one maximal-match may form part of several other maximal-matches.

Definition. A *tile* is a permanent and unique (one-to-one) association of a substring from P with a matching substring from T. In the process of forming a tile from a maximal-match, tokens of the two substrings are *marked*, and thereby become unavailable for further matches. A tile of length s starting at P_p and T_t is written as tile(p, t, s).

With the definitions of tiles and maximal-matches in place it is worth noting that in many situations isolated, short maximal-matches can be ignored. For example, in the context of computer languages, it is unlikely that a maximal-match of length 1 or 2 would be significant. The following definition is therefore made:

Definition. A *minimum-match-length* is defined such that maximal-matches (and hence tiles) below this length are ignored. The minimum-match-length can be 1, but in general will be an integer greater than 1.

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 maximizes the number of tokens covered. 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 similar passages in the source texts rather than chance similarities arising from the skewed distribution of token usage. With this in mind, the following greedy algorithm is proposed.

Firstly, assume there is a current-maximum-match-length maxmatch (greater than or equal to the global minimum-match-length) which is the length of the largest maximal-matches remaining obtainable from P and T. Then:

```
length_of_tokens_tiled := 0
Repeat
   maxmatch := minimum-match-length
   starting at the first unmarked token of P, for each P_p do
         starting at the first unmarked token of T, for each T_t do
            j := 0
            while P_{p+j} = T_{t+j} AND unmarked(P_{p+j}) AND unmarked(T_{t+j}) do
                  j := j + 1
            if j = maxmatch  then add match(p, t, j) to list of matches of length j
            else if j > maxmatch then start new list with match(p, t, j) and maxmatch := j
   for each match(p, t, maxmatch) in list
         if not occluded then                  /* Create new tile */
            for j:= 0 to maxmatch - 1 do
                  mark_token(P_{p+j})
                  mark_token(T_{t+j})
            length_of_tokens_tiled := length_of_tokens_tiled + maxmatch;
Until maxmatch = minimum-match-length
```

Note that "not occluded" is taken to mean that none of the tokens P_p to $P_{p+maxmatch-1}$, and T_t to $T_{t+maxmatch-1}$ has been marked during the creation of an earlier tile. However, given that smaller tiles cannot be created before larger ones, it suffices that only the ends of each new putative tile be tested for occlusion, rather than the whole maximal-match. The first statement in the repeat loop, i.e. the search for maximal-matches, is implemented by the function `scanpattern`, while the task of tile creation is implemented by function `markarrays`.

In other words, finding all the maximal-matches involves initially assuming that maxmatch is the (global) minimum-match-length. If a maximal-match of that length is found, it is added to the list. Otherwise, if a longer maximal-match is found a new list is started and maxmatch becomes the length of the new maximal-match. By the time all the unmarked portions of P have been traversed, maxmatch is the largest maximal-match obtainable given the current markings of P and T.

During the second phase of each iteration of the algorithm, all the maximal-matches in the list either become tiles or are occluded by tiles laid earlier in the processing of the list. Note that with each iteration the value of maxmatch will decrease monotonically until it becomes the global minimum-match-length. Note also that, even though a maximal-match may fail to become a tile because it is partially occluded by a sibling tile, a shortened version of the same maximal-match may become a tile during subsequent iterations.

Theorem:. The algorithm above is optimal, in terms of maximizing the coverage of the strings, and computes a metric, provided that strings down to length 1 are allowed.

Outline of Proof. Both are easily seen after observing that if a token in P has a match anywhere in T, it must at least find a partner in a tile of length 1. Therefore, any token that can participate in a tile, will be in a tile and tile coverage is maximized. To obtain the metric, one needs simply to minimize the number of tokens left uncovered (zero for a complete match).
End of Proof Outline

However, if the restriction of matches to be above a minimum-match-length of 1 is enforced, suboptimal behaviour can result (and only an approximation to the metric is computed). For example, if the minimum match-length is 2, and:

P = c a a b a a d
T = b a a d c a a a a b a a

the greedy algorithm will choose the substring `aabaa` of P to match with T, rather than the two

substrings `caa` and `baad` – a match of length 5 rather than 7. The problem is one of fragmentation, i.e. matchable items remaining but the resulting tiles being below the minimum match-length. In other words, when the minimum-match-length is greater than 1, the value returned by GST is an approximation to the optimal value obtainable with that minimum-match-length.

Theorem. The greedy string tiling algorithm described above has worst case complexity O(n^3).

Outline of Proof. Of the two phases of each iteration, the first is the more expensive. The reason for this is that there can be at most $(|T| - L) \times (|P| - L)$ maximal-matches, but at most |P| / L tiles will be created (at linear cost). All the other maximal-matches will be quickly eliminated; occlusion is resolved by a simple pair of tests. Furthermore, the more tiles that are formed during one iteration, the shorter the unmarked substrings that remain for subsequent iterations. The worst case is therefore where there is exactly one string of any given length. (Alternatively, one can view this as maximizing the number of iterations.)

If we assume that |P| = |T| = n, then the maximum number of distinct substrings (each of different length) to fit into length n is approximately $\sqrt{2n}$. Assume also that all the marked substrings are forced to one end of the pattern and text strings. Then, the last maximal-match of length 1 will fit in only one place on each of the pattern and text strings, which will require only a single comparison. The maximal-match of length 2 will be able to fit into 3 slots (leaving one for the last), i.e. 2 alternatives of size 2, so 2^2 pairwise comparisons or 2×2^2 token comparisons. There will be 4 possible locations on each of the pattern and text strings for a maximal-match of size 3, which means 4^2 pairwise comparisons of size 3. In general, for maximal-matches of length k, there will be $k(\sum k - 1 + 1)^2$ token comparisons, (From this it should be clear why the worst-case number of comparisons in this phase greatly exceeds that required by the second phase.) $\frac{k}{4}(k^2 - k + 2)^2$ is bounded above by kn^2 as all strings are less than or equal to $\sqrt{2n}$. Summing over k for all maximal-match sizes from 1 to $\sqrt{2n}$, results in O(n^3).
End of Proof Outline

3. Tuning the Greedy String Tiling Algorithm

While the worst-case complexity of the GST algorithm is O(n^3) for the reasons outlined above, the complexity observed in practice can be improved by the following, more or less standard techniques:

- Tokens of the pattern string are doubly linked so that each unmarked token points to the next and previous unmarked tokens.
- Tokens of the text string are also doubly linked, but in this case so that unmarked tokens with the same value are linked, starting with an array which records the first occurrence of each possible token value. This means that for a given pattern token value, only those text tokens with the same value will be tried. In general, the linking of tokens in the text and pattern strings means that marked tokens can be skipped.
- If the distance to the next tile on the pattern string is less than the minimum-match-length, the remaining unmarked portion of the pattern plus the following tile are skipped.
- When searching for maximal-matches greater than or equal to the current-maximum-match-length maxmatch, having found a potential starting point for a new maximal-match (i.e. P_p matches T_t) starting testing at $P_{p+maxmatch-1}$ and $T_{t+maxmatch-1}$ rather than P_{p+1} and T_{t+1}. In particular, a hash-value for the minimum-match-length tokens up and including to the current token is kept for each pattern and text token, so after finding the starting point for a new match, a hash-value comparison is performed on $P_{p+maxmatch-1}$ and $T_{t+maxmatch-1}$. (The Karp-Rabin algorithm is used to generate the hash values as the strings are being read in. The Karp-Rabin algorithm is discussed in greater detail below.)
- If comparison of hash-values at $P_{p+maxmatch-1}$ and $T_{t+maxmatch-1}$ succeeds, comparison of the actual substrings is done from $P_{p+maxmatch-1}$ and $T_{t+maxmatch-1}$ back to P_{p+1} and T_{t+1}, because failure to match the two substrings is more likely to occur the further one is from the start of the match.

The revised algorithm for `scanpattern` is:

```
Starting at the first unmarked token of P, for each unmarked P_p do
   if distance to next tile ≤ minimum_match_length then advance t to first unmarked token after next tile
   else
        matchsize := minimum_match_length
        for each t such that T_t = P_p do            /* IE skip to next unmarked, matching T_t */
           if hash-value(P_{p+matchsize−1}) = hash-value(T_{t+matchsize−1})
                                    AND unmarked(P_{p+matchsize−1})
                                    AND unmarked(T_{t+matchsize−1}) then
                if for all j from matchsize-1 down to 1, P_{p+j} = T_{t+j} then
                   k: = matchsize
                   while P_{p+k} = T_{t+k} AND unmarked(P_{p+k}) AND unmarked(T_{t+k}) do
                            k := k + 1
                   if k =  matchsize then add to list of matches
                   else if k >  matchsize then
                            restart list with new maximal-match and update matchsize
```

The revised version of `markarrays` differs little from that used in the first GST implementation:

```
for each match(p, t, maxmatch) in list
     if not occluded then                /* Create new tile */
        for j:= 0 to maxmatch - 1 do
             mark_token(P_{p+j})
             unlink_token(P_{p+j})
             mark_token(T_{t+j})
             unlink_token(T_{t+j})
```

In other words, creating a new tile entails relinking each pattern and text token so that marked tokens are bypassed. This requires a traversal of both the pattern and text substrings for each new tile, but the cost is small compared to the cost of scanning for the set of matches.

It should be emphasized that the tactics employed in tuning the original algorithm only improve the average behaviour of the GST algorithm − the worst-case behaviour remains unaltered. None-the-less, experiments with this algorithm demonstrate that these tactics do yield useful improvements in practice. In later discussions this will be referred to as the Tuned Greedy String Tiling algorithm.

4. Running-Karp-Rabin Greedy String Tiling

As will be seen in the section below describing some experimental results, even with considerable tuning the complexity of the Tuned Greedy String Tiling algorithm evident from the experiments is above $O(n^2)$. While this is probably acceptable for all but the longest texts or largest groups of texts, it is still too high. With this in mind, a totally different algorithm has been devised, based on the well known Karp-Rabin string-match algorithm [Karp87], also described, among others, by Cormen, Leiserson and Rivest [Corm90] and Gonnet and Baeza-Yates [Gonn91]. The central notion behind the Karp-Rabin algorithm is that if a hash-value exists for a string of length s starting at t, the hash-value for a string of length s starting at t+1 can be calculated using a simple recurrence relation. In particular, if the length of the pattern string is |P|, the hash-value of every substring of length |P| from the text string is compared with the hash-value of the pattern string. If two hash-values are identical, the pattern and text substrings are compared item-by-item. Note that if there are no matching substrings, or only the first is sought, the complexity is linear on the sum of the two strings. More to the point, while the worst case complexity is $O(n^2)$ when all matching substrings are sought, the complexity one sees in practice is still close to linear. (The version of the recurrence relation used in this work is due to Gonnet and Baeza-Yates.[Gonn90])

The new algorithm − called Running Karp-Rabin matching − extends Karp-Rabin matching in the following ways:

1. 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 substring from P_p to P_{p+s-1}. Hash-values are similarly created for each (unmarked) substring of the length s of the text string.
2. Each of the hash-values for the pattern string is then compared with the hash-values for text string; for each pair of pattern and text hash-values found to be equal, there is a possible match between a substring each of the pattern and text strings. A hash-table of the Karp-Rabin hash-values of text substrings 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, matching is extended forward as is done in the Tuned Greedy String Tiling algorithm, resulting in maximal-matches. (In other words, length s is the minimum match-length being sought during one iteration.)
3. The length of the matches being sought, i.e. s, is reduced after each iteration until the minimum-match-length is reached.

Definition The parameter s – the minimum match-length being sought during one iteration – will be called the *search-length*.

The function `scanpattern` now has as a parameter the search-length, s. The algorithm for `scanpattern(s)` is:

```
Starting at the first unmarked token of T, for each unmarked T_t do
   if distance to next tile ≤ s then advance t to first unmarked token after next tile
   else create the KR hash-value for substring T_t to T_{t+s-1} and add to hashtable
Starting at the first unmarked token of P, for each unmarked P_p do
   if distance to next tile ≤ s then advance p to first unmarked token after next tile
   else
        create the KR hash-value for substring P_p to P_{p+s-1}
        check hashtable for hash of KR hash-value
        for each hash-table entry with equal hashed KR hash-value do
           if for all j from 0 to s-1, P_{p+j} = T_{t+j} then          /* IE match is not hash artifact */
                k: = s
                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 return(k)          /*  and restart scanpattern with s = 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. Each queue records maximal-matches of the same length. 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.

The question arises as to what is an appropriate value for the parameter s passed to `scanpattern`. 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.

The algorithm for `markarrays` is similar to that used for the Tuned GST algorithm, except that, it is applied to a list of queues (each queue containing maximal-matches of the same size) rather than a single list. This version of `markarrays` also has the parameter s, the search-length.

```
Starting with the top queue, while there is a non-empty queue do
   if the current queue is empty then drop to next queue   /* corresponding to smaller maximal-matches */
   else
        remove match(p, t, L) from queue          /* Assume the length of maximal-matches in the
                                                     current queue is L */
        if match not occluded then
           for j:= 0 to L - 1 do
                mark_token(P_{p+j})
                mark_token(T_{t+j})
           length_of_tokens_tiled := length_of_tokens_tiled + L
        else if L – L_occluded ≥ s then          /* IE the unmarked part remaining of the maximal-match */
           record unmarked portion on list of queues
```

The first thing to notice is that after KR hashing has revealed a potential substring match in `scanpattern`, the pairwise test:

```
if for all j from 0 to s-1 P_{p+j} = T_{t+j} then
```

can be deferred until `markarrays` (though the extension of the match of length s to to a maximal match should still take place in `scanpattern`). There are two reasons why this is a very effective strategy. In practice, failures of the KR hashing are very rare, i.e. very few of the putative matches revealed by KR hashing turn out not to be matches on closer inspection. Secondly, a large number of matches revealed by KR hashing, whether genuine or not, will be occluded by sibling tiles. It is therefore worthwhile deferring the pairwise comparison until creation of a new tile is imminent.

The top-level algorithm for the system is similar to that used by the Tuned GST algorithm, namely:

```
Search-length s := initial-search-length    /* Currently 20 */
stop := false
Repeat
   L_max := scanpattern(s)          /* L_max is the size of the largest maximal-matches found in this iteration */
   if L_max > 2 × s then s := L_max    /* Very long string; don't mark tiles but try again with larger s */
   else
        markarrays(s)               /* Create tiles from matches takes from list of queues */
        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
```

From this top-level algorithm it should be clear that "very long" strings will only be found during the first iteration; after the first iteration there will be no matches greater than twice the current search-length.

Theorem: The worst-case complexity of Running Karp-Rabin Greedy String Tiling is $O(n^3)$.

Outline of Proof.
Assume that there are N tokens in both the pattern and text strings. Assume also that all the text tokens have the same value, and that the pattern string contains a substring of length 2×m of these tokens, say between P_p and P_{p+2m-1}. Finally, assume that in the current iteration we are looking for maximal-matches

between (m + 1) and 2 × m in length. Let r = $N - 2m + 1$, and note that there are r substrings of T of length $2 \times m$ matching the $2 \times m$ tokens of the pattern string. Remember that we are looking for maximal-matches, there is also the substring of length $2m - 1$ from P_{p+1} and P_{p+2m-1} that can form $r + 1$ maximal-matches with substrings of T. Similarly, for the substring from P_{p+2} to P_{p+2m-1} there will be $r + 2$ maximal-matches to be found in T. In general, for the substring of length $2M - k$ from P_{p+k} to P_{p+2m-1} there will be $r + k$ maximal-matches with substrings from T. Therefore, the total number of maximal-matches with lengths greater than m and less than or equal to $2M$ is: $\sum_{k=0}^{m-1} r + k$; The number of comparisons is: $\sum_{k=0}^{m-1} (r + k)(2m - k)$. The latter expression can be reduced to: $\frac{3}{2} m^2 N - \frac{7}{3} m^3 + \frac{mN}{2} + \frac{m}{3}$, after substituting for r. Remember that this value is due to a single substring of length 2m. In other words, when searching for maximal-matches between m and 2m in length, the worst-case appears when the maximal-match is of length 2m. Nothing has been said so far about the relationship of m to N; it can now be seen that the worst case appears when there are as many maximal-matches of length 2m as will fit in the pattern string. In other words, the worst case is the expression above multiplied through by $\frac{N}{2m}$, or

$$\frac{3}{4} mN^2 - \frac{7}{6} m^2 N + \frac{N^2}{4} + \frac{N}{6}$$

This is at a maximum when m is $\frac{9N}{28}$, which when substituted back gives a worst-case complexity of $O(m^3)$.

End of Proof Outline

A less formal argument can also be made, suggesting that a more realistic estimate of the complexity in practice is between O(n) and $O(m^2)$. Firstly, with an initial search-length set at 20 (or some other small, fixed value), only a constant number of iterations will be required. Then, during a single iteration the unmarked portions of both the pattern and text strings will need to be traversed, which means that the lower bound on the complexity is inevitably O(n). It is also true in practice that both KR-hashing and the hashing of the KR-hash values do not lead to a significant number of false positives (i.e. substrings with equal hash values but which do not match); indeed, there often were no false positives at all in the experiments to be described below. The determiner of the complexity is therefore the number of maximal-matches found, and in particular the number of maximal-matches as a function of the unmarked string-lengths. The average number of maximal-matches per unmarked token turns out to be relatively small because initially the search-length is large and there are few strings of that length, and then as shorter search-lengths are being sought, an increasing proportion of the strings have already been tiled. More to the point, while the average number of maximal-matches per unmarked token is not constant, it appears to increase only very slowly with increasing string length. Finally, the worst-case assumed that all maximal-matches are of the largest possible size, in this case $2M - 1$. In practice this is highly improbable; rather there will be a range of lengths between m and $2M - 1$, and the number of comparisons actually performed can be further diminished by adopting the suggestion made earlier of deferring the testing of the first search-length tokens until `markarrays`, at which point the tests will no longer be needed because the tile would be occluded.

5. Experimenting with Greedy String Tiling

The token strings used in the experiments fall into three broad groups: those composed of tokens drawn from actual programs, those which initially were tokenizations of actual programs but which also had been altered and finally token strings that are purely synthetic.

5.1 Program Derived Token Strings

Three token strings fall into this group: StudA, StudB and MJW. These strings represent tokenizations of C programs, solutions to a Third Year Operating System assignment. MJW is my solution for the assignment, while StudA is from a solution submitted by a student who adopted an approach to the assignment that was similar to mine. StudB represents the tokenization of a solution submitted by

another student, who copied StudA's solution and then spent considerable effort covering this fact.

5.2 Altered Program Tokenizations

In order to examine experimentally the run-time complexities for strings drawn from real text (rather than the synthetic strings to be described below), two series of token strings were created: random.StudA.X and random.MJW.X, where X goes from 1 to 10. In each case, X copies of StudA and MJW strings (respectively) were taken and randomized. (That is, each string is a different randomization, rather than being X copies of a random string drawn from StudA and MJW.) The reason for using randomizations of actually programs rather than random selections from the alphabet, is that the token strings derived from actual programs will better reflect the skewed distribution of token usage evident in real programs.

The string MJW.mod is essentially the string MJW, with a small number of changes, including the addition of a token and the transposition of two substrings.

5.3 Synthetic Token Strings

In string match_3.1, there appear unique substrings each containing exactly 3 tokens (i.e. equal to the minimum string length). These triples are redistributed randomly across the string match_3.2. The string nomatch is the sequence of integers from 0 to the length of the string (minus 1) in ascending order; nomatch.rev has the same tokens in descending order (i.e. each token will match one other, but there will be no strings greater than length 1). The string nomatch1 contains a completely different set of tokens, so in this case there will be no matches of any length.

There are two series of strings to examine experimentally the worst-case analyses discussed earlier. The first of these worst-case series can be characterised by the following strings. The pattern strings, wc1.X, are of the form:

0 1 0 0 1 0 0 0 1 0 0 0 0 1 0 0 0 0 0 1 ... 0 0 0 0 0 1

where X from 0 to 5 indicates strings of increasing length. In other words, the strings wc1.X are made up of substrings of N zeros followed by a 1, where N is initially 1 and then increases in steps of one to a maximum value. The corresponding text strings, wc2.X, are made up entirely of zeros, padded out to a length equal to the pattern strings. This combination simulates the worst case for the Tuned Greedy String Tiling for the following reasons:

- In the pattern string (and the text string), at each pass only a single tile is marked. (By contrast, if there are predominantly short strings, many will be marked in a pass; on the other hand, if there are a few very long strings, only one may be marked in a pass, but it may account for a large number of tokens.) There are approximately $\sqrt{2N}$ single substrings of increasing length in a string of of length N.
- Placing the current longest substring at the end of the pattern ensures that the current search-length will be as short as possible for as long as possible (remembering that in the T_GST algorithm the search-length will grow during a single iteration until it reaches the maximum obtainable from that iteration).
- Having the text string be N repetitions of the same (0) token means that there will be O(N) possible matches for every pattern 0 token.

Name	Longest 0 Substring
wc1.0	17
wc1.1	25
wc1.2	36
wc1.3	44
wc1.4	51
wc1.5	59

The second series of strings intended to test worst-case behaviour are wc3.X (pattern) and wc4.X (text). Each of the wc3.X strings contains four repetitions of N 0 tokens followed by a single 1 token.[4] Like the

wc2.X strings, the wc4.X strings contain only 0 tokens, padded out the the length of the pattern string. Once again, X in the string-name runs from 0 (the shortest string) to 5.

5.4 Results and Discussion

Testing the runtime performance of programs is complicated by the fact that the commonly used utilities, such as System V `times`, are affected by page-faults and other interrupts. The result is that the values returned tend to vary, often greatly, due to a number of factors, particularly system load. An alternative approach was taken in the current set of experiments, viz. to use the `pixie` suite available under the MIPS implementation of UNIX, RISCos. `pixie` implements a virtual CPU. In particular, `a.out` (object) files generated by the standard compiler are converted to run on the pixie machine. Counts are then maintained of the number of pixie-cycles for blocks of code or individual lines. Listed below are the counts of pixie-cycles for two programs, one implementing the Tuned Greedy String Tiling (T_GST) Algorithm and the other implementing the Running Karp Rabin Greedy String Tiling (RKR_GST) algorithm. Results for two flavours of the RKR_GST program are reported: one with the initial search-length set to 20, the other with the initial search-length set to half the pattern-string length (so as to generate the worst-case RKR execution pattern described earlier).

---

4. In the worst-case analysis for the Running-Karp-Rabin Greedy String Tiling algorithm The worst-case occurs when $m = \frac{9N}{28}$. While this is a good upper bound, it does not accurately reflect the fate of any "leftover" substrings (i.e. less than m in length); in the analysis they are treated the same as the original strings, when in practice they are relegated to a subsequent iteration which generates are fewer maximal-matches. Experimentation reveals that, when the leftover strings are taken into account, the worst-case appears when $m = \frac{N}{4}$.

TEST	|P|+|T|	Counts of Pixie Cycles		
		T_GST	**RKR_GST** ($S_I = 20$)	**RKR_GST** ($S_I = \|P\|/2$)
MJW MJW 3	654	411,080	458,150	420,222
MJW MJW.mod 3	655	798,961	517,869	518,188
match_3.1 match_3.2 3	720	522,752	647,866	759,932
nomatch nomatch.rev 3	700	456,028	597,750	707,546
nomatch nomatch1 3	700	441,252	601,249	710,423
match_3.1 match_3.2 4	720	486,154	591,725	703,791
MJW StudA 3	682	4,990,001	627,931	738,495
StudB StudA 3	778	7,234,943	774,989	838,323
MJW random.MJW.1 3	654	3,297,408	589,804	692,020
StudA random.StudA.1 3	710	3,558,140	725,339	836,043
random.MJW.1 random.StudA.1 3	682	2,627,677	605,486	715,976
random.MJW.2 random.StudA.2 3	1,364	12,856,994	1,324,328	1,638,899
random.MJW.3 random.StudA.3 3	2,046	25,990,474	2,063,705	3,145,392
random.MJW.4 random.StudA.4 3	2,728	60,250,724	2,835,116	3,677,644
random.MJW.5 random.StudA.5 3	3,410	110,476,132	3,440,936	5,420,220
random.MJW.6 random.StudA.6 3	4,092	151,216,979	4,303,916	7,675,353
random.MJW.7 random.StudA.7 3	4,774	204,783,983	5,159,828	6,648,506
random.MJW.8 random.StudA.8 3	5,456	213,891,252	6,186,818	8,405,061
random.MJW.9 random.StudA.9 3	6,138	420,255,184	7,219,113	9,750,312
random.MJW.10 random.StudA.10 3	6,820	513,342,529	8,063,634	14,523,443
wc1.0 wc2.0 3	341	14,912,146	2,312,546	2,346,297
wc1.1 wc2.1 3	701	123,486,880	6,155,913	7,096,287
wc1.2 wc2.2 3	1,405	965,590,560	46,417,658	42,304,561
wc1.3 wc2.3 3	2,069	3,044,245,150	168,348,011	60,095,044
wc1.4 wc2.4 3	2,755	7,127,629,129	221,928,791	129,407,183
wc1.5 wc2.5 3	3,659	16,574,716,089	359,134,572	386,066,263
wc3.0 wc4.0 3	345	9,386,335	382,201	7,479,319
wc3.1 wc4.1 3	705	74,883,455	754,917	46,303,839
wc3.2 wc4.2 3	1,409	580,824,555	1,502,278	315,699,051
wc3.3 wc4.3 3	2,073	1,833,531,948	2,211,090	959,256,962
wc3.4 wc4.4 3	2,761	4,312,259,724	2,935,302	2,195,723,973
wc3.5 wc4.5 3	3,665	10,052,476,274	3,853,468	4,991,707,632

To help you interpret the above table, the columns have the following meaning:

- The TEST lists the two strings being compared plus the minimum-match-length (generally 3).
- $|P|+|T|$ is the sum of the two string-lengths (computations of runtime complexity will be done against this figure).
- The counts of Pixie cycles for each tests when applied first to the Tuned GST algorithm, and then to the two versions of Running-Karp-Rabin GST algorithm, the first with the initial search-length set to 20 and the second with initial search-length set to half the pattern-string-length.

The following points can be noted from the figures listed above:

- In the "boundary" cases, i.e. a complete match using a single tile or no tiles being created at all, both algorithms return low pixie-cycle count values, and in each case the T_GST algorithm is somewhat faster than the RKR_GST algorithm (25%–35%). The same is true in the case of a large number of

minimal tiles, which can be marked in a single pass. However, as soon as there are even a small number of differently sized tiles, e.g. in the comparison of MJW and MJW.mod, the situation reverses and RKR_GST algorithm is 50% faster.

- The difference between the two algorithms becomes much more significant when the strings drawn from actual programs are compared; the T_GST implementation is approximately 7 times slower.
- One would expect that the $S_I = |P|/2$ version of RKR_GST would be slower than the $S_I = 20$ version due to the additional, empty passes that the former must do before it encounters some matches. It should be noted from the figures tabulated above that for both the boundary cases and the strings drawn from real text (i.e. MJW/StudA, StudB/StudA or the Random series), the difference is not large; excluding the exact match test, MJW/MJW, where the $S_I = |P|/2$ version is slightly faster and the two worst-case series the $S_I = |P|/2$ version takes on average 29% more cycles than the $S_I = 20$ version, though the difference will rise as string-length increases.
- Curve fitting was performed on the pixie-cycle counts for the sequence of experiments comparing random.MJW.X and random.StudA.X. In particular, the pixie-cycle counts were plotted against total string-lengths using the relation: $c = As^B$. The values for the coefficients A and B, and the correlation coefficient r^2 are:

random.MJW.X random.StudA.X			
	T_GST ($S_I = 20$)	**RKR_GST** ($S_I = \|P\|/2$)	**RKR_GST**
A	0.99	414.0	244.0
B	2.26	1.12	1.23
r^2	0.99	1.00	0.98

  In other words, the RKR_GST algorithm has, in examples representing real text, higher proportionality constants but, more importantly, slower rates of growth relative to the combined lengths of the input strings.
- Curve fitting was also performed on the pixie-cycle counts for the sequence of experiments comparing the strings wc1.X with wc2.X, i.e. looking experimentally at the worst case behaviour of the two algorithms. Once again the pixie-cycle counts were plotted against total string-length using $c = As^B$. In this case, the values for A, B and r^2 are:

		T_GST	**RKR_GST** ($S_I = 20$)	**RKR_GST** ($S_I = \|P\|/2$)
wc1.X wc2.X	A	0.48	2.99	10.63
	B	2.96	2.29	2.08
	r^2	1.00	0.98	0.98
wc3.X wc4.X	A	0.29	1,220.1	0.68
	B	2.95	0.98	2.76
	r^2	1.00	1.00	1.00

  It can be seen that the values returned for the curve fitting appear to confirm the worst-case complexities suggested by the theory, viz. that the wc1.X/wc2.X series represents a worst-case for the T_GST algorithm, while the wc3.X/wc4.X represents a worst-case for the RKR_GST algorithm when the initial search-length (i.e $2m$ in the algorithm) is set to $N/2$.
- The behaviour of the $S_I = 20$ version of the RKR-GST algorithm also illustrates the benefits to be gained by the very-long-string test (particularly evident in the wc3.X/wc4.X series). As was mentioned earlier, what happens is that, when a very long string is detected the search is recommenced with a new search-length equal to the length of the very long string. This means that longer strings are marked sooner, which allows them to be bypassed in subsequent iterations.

- Pixie-cycles have the distinct advantage that they accurately reflect the amount of work that has taken place during a computation and are not affected by vagaries of the system such as interrupts. However, to give some sense of reality to the pixie-cycle counts some experiments were rerun and timed using the System V UNIX utility `times`. However, because `times` takes into account page faults, run-times for the same program tend to vary depending on the number of users on the system (a MIPS R3030 running at 25MHz with 64 Mbytes memory). For this reason, experiments listed below were run repeatedly until the mean user-time stabilized.

TEST	Algorithm	Pixie Cycles	Mean User Time	Mean Sys Time
random.MJW.10 random.StudA.10	RKR_GST($S_I = 20$)	8,063,634	0.35	0.01
wc1.5 wc2.5	RKR_GST($S_I = 20$)	359,134,572	13.42	0.22
wc1.5 wc2.5	T_GST	16,574,716,089	335.44	0.01

With User and System time being in seconds, the average time per pixie-cycle for each of these experiments is therefore 4.46×10^{-8}, 3.80×10^{-8} and 2.2×10^{-8}, respectively. The first is probably slightly unrepresentative because it would have been overly influenced by the time required to read the strings from disk. The second of the two values has probably been influenced by the larger system component due to a greater use of `malloc`, but it is still representative of the RKR_GST execution times. Overall, a single pixie-cycle on the MIPS 3030 takes approximately 27nsec.

In the analysis of the RKR_GST algorithm, a number of assumptions were made, particularly about where time is being spent. To test these assumptions the $S_I = N/2$ − the version producing the worst case − was instrumented and counts taken of a number of values. The results are tabulated below:

	Initial Search Length N / 2						
TEST	|P| + |T|	MMPT	Tokens Seen	MM Produced	List Moves	KR fail (scan)	KR fail (mark)
MJW MJW 3	654	0.00	653	1	0	0	0
MJW test 3	655	0.24	1109	97	43	0	0
match_3.1 match_3.2 3	720	0.17	719	120	0	0	0
nomatch nomatch.rev 3	700	0.00	0	0	0	0	0
nomatch nomatch1 3	700	0.00	0	0	0	0	0
match_3.1 match_3.2 4	720	0.00	0	0	0	0	0
MJW StudA 3	682	0.46	1717	286	122	0	0
StudB StudA 3	778	0.87	2213	396	112	0	0
MJW random.MJW.1 3	654	0.31	1260	194	18	0	0
StudA random.StudA.1 3	710	0.77	1383	524	81	0	0
random.MJW.1 random.StudA.1 3	682	0.23	1297	152	22	1	0
random.MJW.2 random.StudA.2 3	1364	0.61	2602	778	130	0	0
random.MJW.3 random.StudA.3 3	2046	1.71	2045	3491	985	1	1
random.MJW.4 random.StudA.4 3	2728	0.91	5082	2229	287	0	0
random.MJW.5 random.StudA.5 3	3410	1.74	6600	5588	1267	3	3
random.MJW.6 random.StudA.6 3	4092	2.72	8043	10768	2803	3	0
random.MJW.7 random.StudA.7 3	4774	0.54	12373	3328	757	28	6
random.MJW.8 random.StudA.8 3	5456	1.32	9843	6246	857	3	1
random.MJW.9 random.StudA.9 3	6138	1.48	11102	7833	1091	6	1
random.MJW.10 random.StudA.10 3	6820	3.39	13133	21524	4509	17	7
wc1.0 wc2.0 3	341	17.21	497	6822	264	0	0
wc1.1 wc2.1 3	701	32.32	1338	21111	688	0	0
wc1.2 wc2.2 3	1405	66.23	2075	110057	2547	0	0
wc1.3 wc2.3 3	2069	67.14	3600	169532	3145	0	0
wc1.4 wc2.4 3	2755	124.68	5369	315237	6019	0	0
wc1.5 wc2.5 3	3659	214.39	8168	769209	13176	0	0
wc3.0 wc4.0 3	345	39.32	344	13527	2499	0	0
wc3.1 wc4.1 3	705	77.68	704	54689	10320	0	0
wc3.2 wc4.2 3	1409	154.69	1408	217797	41934	0	0
wc3.3 wc4.3 3	2073	228.31	2072	473067	91977	0	0
wc3.4 wc4.4 3	2761	303.56	2760	837836	163314	0	0
wc3.5 wc4.5 3	3665	401.44	3664	1470864	286710	0	0

The columns have the following interpretation:

- Test – As before, the pair of string being compared and the minimum search-length.
- |P|+|T| – As before, the sum of the pattern and text string lengths.
- MMPT – The largest average number of maximal-matches per unmatched token by one of the iterations of that TEST.
- Tokens Seen – The total number of pattern and text tokens traversed.
- MM Produced – The total number of maximal matches produced.
- List Moves – When inserting a new maximal-match into the list of maximal-match queues and the pointer to the current queue does not correspond to the queue into which the addition is to occur, this is the sum of the number of pointer traversals required to get to the appropriate queue.
- KR fail (scan) – The number of times KR-hashing would have failed in `scanpattern` if testing were carried out there (i.e. in this case the test was carried out, but failure did not prevent

continuation to the marking stage).
- KR fail (marking) – This failure of KR-hashing detected just before creation of a new tile is to occur in `markarrays`.

A number of things can be see from these figures:

- As was predicted, KR-hashing rarely fails in `scanpattern`, and even less often when the test is moved to `markarrays`, so it was reasonable to assume that effect of KR-hashing failure could be ignored.
- For all the tests that represent the sorts of texts that are likely to occur in practice, viz. the boundary condition tests (no match or complete match) or the random series, the number of matches per unmarked token is typically very small (often less than one), which supports the assertion in the discussion of the RKR_GST average time complexity.
- In the worst-case experiments, the number of tokens in each string is not the dominant cost, but rather the number of maximal-matches produced. (More tokens are traversed in the random.MJW.1a/random.StudA.10 than in any of the worst-case experiments, but the number of pixie-cycles is far less than all but the shortest worst-case strings).
- The cost of moving between the lists of maximal-match queues when adding new maximal-matches is also not dominant, though it can be significant. In other words, when the number of maximal-matches is large, the cost is is exacerbated by the large number of list movements required.

6. Conclusion and Further Work

Greedy String Tiling is proposed as a method for comparing pairs of strings to determine the degree to which they are similar, and two algorithms are presented. The second of these, Running-Karp-Rabin Greedy String Tiling, appears to have an average complexity that is close to linear. No doubt the algorithm can be improved. For example, at the moment, after strings with length greater than or equal to search-length s have been found, s is halved and the process is repeated. A better strategy might be to vary the decrease in the length of s according to the number of matches pushed onto the heap in the previous pass. If the number was large, it may be prudent to decrease s rather more slowly. In other words, the decrement of s could be made a function of both the number of matches found in the previous pass and the length of (untiled) string remaining.

The Running Karp-Rabin incarnation of the Greedy String Tiling metric has been returned to its original application area, the YAP plagiarism detection system, and the new version, YAP3, is currently being tested.

A second application area suggests itself for the RKR_GST algorithm and the Greedy String Tiling metric: computational biology, and in particular, sequence matching, as this area appears to share many of the problems seen with plagiarism detection, e.g. the need for an efficient algorithm that can deal with transposed substrings and where each element is matched only once. In this, the RKR_GST algorithm appears to combine features of the two dominant algorithms currently in use, dot metric methods and dynamic programming.[5] For example, a dot matrix plot can be obtained very simply by suppressing the marking of tiles and instead returning all matches. On the other hand, techniques based on dynamic programming face the problems discussed earlier in relation to use of the use of the UNIX utility `sdiff` and the longest common subsequence algorithm in general, namely an inability to deal effectively with transposed substrings. (In particular, faced with a mismatch, all that can be done is to create a *gap* with an associated gap-penalty; the value of penalties for gaps of different sizes is a more-or-less ad hoc decision.) On the other hand, while dot matrix plots are able to deal with transposed substrings, interpretation of the plots is left to the practitioner and can be difficult for complicated alignments.

7. Bibliography

5. For a discussion of these techniques, see for example the review by States and Boguski [Stat92].

[Corm90] Cormen, Thomas H., Charles E. Leiserson and Ronald L. Rivest, *Introduction to Algorithms*, MIT Press (1990).

[Gonn90] Gonnet, G. H. and R. A. Baeza-Yates, ‘‘An Analysis of the Karp-Rabin String Matching Algorithm’’, *Information Processing Letters* **34**, p. 271−274 (1990).

[Gonn91] Gonnet, G. H. and R. Baeza-Yates, *Handbook of Algorithms and Data Structures (Second Edition)*, Addison-Wesley (1991).

[Heck78] Heckel, Paul, ‘‘A Technique for Isolating Differences Between Files’’, *Communications of the ACM* **21**(4), p. 264−268 (April 1978).

[Karp87] Karp, Richard M. and Michael O. Rabin, ‘‘Efficient Randomized Pattern-Matching Algorithms’’, *IBM Journal of Research and Development* **31**(2), p. 249−260 (March 1987).

[Krus83] 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).

[Mill87] Miller, Webb, *A Software Tools Sampler,* Prentice-Hall (1987) (Chapter 3, "File Comparison Programs").

[Stat92] States, David J. and Mark S. Boguski, ‘‘Similarity and Homology’’, *Sequence Analysis Primer*, ed. Michael Gribskov and John Devereux, pp. 89−157, W. H. Freeman (1992).

[Tich84] Tichy, Walter F., ‘‘The String-to-String Correction Problem with Block Moves’’, *ACM Transactions on Computer Systems* **2**(4), p. 309−321 (November 1984).

[Wise92] 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, p. 268−271 (March 5−6, 1992).

[Wise95] 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., p. 393−401, AAAI Press (July 16-19, 1995).

[Wise96] Wise, Michael J., ‘‘Improved Detection of Similarities in Computer Program and other Texts’’, *Twenty-Seventh SIGCSE Technical Symposium*, Philadelphia, U.S.A., pp. 130-134 (February 15-17, 1996).