Journal of Universal Computer Science, vol. 11, no. 6 (2005), 1083-1101
submitted: 23/9/04, accepted: 25/5/05, appeared: 28/6/05 © J.UCS

A Fast T-decomposition Algorithm

Jia Yang
(Department of Computer Science, The University of Auckland, New Zealand
jyan055@cs.auckland.ac.nz)
Ulrich Speidel [1]
(Department of Computer Science, The University of Auckland, New Zealand
ulrich@cs.auckland.ac.nz)

Abstract: *T-decomposition* was first proposed and implemented as an algorithm by Mark Titchener. It has applications in communication of code sets and in the fields of entropy and similarity measurement. The first implementation of a T-decomposition algorithm by Titchener was subsequently followed by a faster version named `tcalc`, developed in conjunction with Scott Wackrow. An improved T-decomposition algorithm was published in 2003 by the authors with the implementation `tlist`. This paper introduces a new algorithm that builds on our 2003 algorithm. Comparative experimental results are given to show that the new version has a significantly better time performance than previous algorithms.

Key Words: T-decomposition, computable complexity measures, parsing, entropy
Category: E.4

1 Introduction

T-decomposition was first introduced by Mark R. Titchener [Titchener 1993, Titchener and Wackrow 1995, Titchener 1996]. It describes a self-learning automaton that analyzes how a given string can be constructed recursively. It has since been studied within the context of revealing the information structure in strings and led to the development of a computable complexity measure named T-complexity, which in turn serves as the basis for two further measures, T-information and T-entropy [Titchener 1998a, Titchener 1998b, Titchener 1998c, Titchener 2000, Guenther 1998]. The latter of these is also known as *deterministic entropy*. The relationship between T-entropy and the Kolmogorov-Sinai entropy (Pesin entropy) of the logistic map was discussed in [Ebeling et al. 2001]. A further link [Titchener et. al. 2005] with the Shannon entropy has since been pointed out.

The authors' particular interest in T-decomposition lies in its application to the area of similarity measures [Yang and Speidel 2003c]. Using T-decomposition in large-volume tasks requires an efficient T-decomposition algorithm. Examples are the real-time text classification of a large number of files, or the similarity comparison of large files.

[1] nee Günther (Guenther)

In this paper, we will first revisit the principle of T-decomposition. We will then briefly describe the previous T-decomposition algorithms by Wackrow and Titchener as well as our own approach in [Yang and Speidel 2003b], on which the algorithm presented here is based. This is followed by a discussion of our new approach and a presentation of its experimental performance.

1.1 The principle of T-decomposition

Let the set $A = \{a_1, a_2, \ldots, a_{n-1}, a_n\}$ be a finite alphabet. We denote the cardinality of A by $\#A$, thus $\#A = n$. Elements $a_i \in A$ are called *characters*. Let A^* denote the set of all finite strings that can be generated by concatenating characters from A. We denote the empty string by λ. Let $A^+ = A^* \setminus \{\lambda\}$. For two strings $x, y \in A^*$, let xy denote the concatenation of x and y. We use x^k to denote the concatenation of k copies of x. A *token* is a string from A^+.

Assume that $x \in A^+$. The principle of T-decomposition may be described as follows:

1. Parse x over A. Thus each of the $|x|$ characters in x is parsed into a token.
2. Identify the last (rightmost) token in the current parsing of x. We will call this last token a.
3. Set $i = 1$.
4. Identify p_i as the penultimate token. Identify the maximum length k_i (in tokens) of the run $p_i^{k_i}$ of $k_i \geq 1$ instances of p_i which ends in the penultimate token.
5. If the run in the previous step starts with the leftmost token, go to step 8
6. Parse x left-to-right. Each run of one or more tokens p_i in the parsing followed by another token q is combined (merged) into a single token according to one of the following two patterns:
   (a) ${p_i}^l q$, where $q \neq p_i$ and $1 \leq l \leq k_i$; or
   (b) ${p_i}^{k_i+1}$.

   Note that *all* tokens in the parsing of x that are instances of p_i can be merged into a new token under one of the two pattern rules above.
7. Increment i and go to step 4
8. End.

Note that after each parsing pass, the tokens that make up x belong to a prefix-free complete code set called a T-code. This guarantees that the result of the next parsing pass is unambiguous.

T-decomposition thus parses x into a series of tokens $p_m^{k_m} p_{m-1}^{k_{m-1}} ... p_2^{k_2} p_1^{k_1} a$. p_i is called a *T-prefix*, while k_i is called a *T-expansion parameter*. These parameters may be used to build x or, alternatively, construct the T-code set for which x is one of the longest codeword. The existence and uniqueness of the T-decomposed result of a given finite string has been proved by Nicolescu and Titchener [Nicolescu 1995, Nicolescu and Titchener 1998].

[Titchener 1998a, Titchener 1998b, Titchener 1998c, Guenther 1998] give detailed descriptions of the principle of T-decomposition. Here we use an example to demonstrate the T-decomposition process.

Let $A = \{0, 1\}$ and $x = 10100100100110011101$. We will now decompose x using T-decomposition.

1. Parse x over A. Using commas to separate the tokens, we get
   $x = 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1.$

2. Identify a. Since the last character of x is 1, $a = 1$. Thus
   $x = 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, \underbrace{1}_{a}.$

3. The penultimate token p_1 is 0 in this case. There are no more instances of p_1 immediately to the left of the penultimate token, Thus the length of the run is $k_1 = 1$. We get
   $x = 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, \underbrace{0}_{0^1}, \underbrace{1}_{a}.$

4. Parse x left-to-right. Consecutive tokens are merged into a new token if they combine as:

   (a) ${p_1}^l q$, where $q \neq p_1$ and $1 \leq l \leq k_1$; or

   (b) ${p_1}^{k_1+1}$

   After this parsing, we thus get

$$x = 1, 01, 00, 1, 00, 1, 00, 1, 1, 00, 1, 1, 1, \underbrace{0}_{0^1} \underbrace{1}_{a}.$$

5. The penultimate token of x here is $p_2 = 1$. Since there are three consecutive tokens equal to p_2 to the left of and including the penultimate token $k_2 = 3$, the second T-decomposition step yields $p_2^{k_2} = 1^3 = 111$.
   $x = 1, 01, 00, 1, 00, 1, 00, 1, 1, 00, \underbrace{111}_{1^3}, \underbrace{0}_{0^1} \underbrace{1}_{a}.$

6. Repeating for p_2 what we did in step 4 for p_1, we parse x and get $x = 101, 00, 100, 100, 1100, \underbrace{111}_{1^3}\underbrace{0}_{0^1}\underbrace{1}_{a}$.

   As in 5, we complete the third T-decomposition step, which yields $p_3^{k_3} = 1100^1 = 1100$. Thus $x = 101, 00, 100, 100, \underbrace{1100}_{1100^1}\underbrace{111}_{1^3}\underbrace{0}_{0^1}\underbrace{1}_{a}$

7. Keep repeating the previous steps until we obtain all T-decomposition parameters of x. The final result is

   $p_1 = 0, p_2 = 1, p_3 = 1100, p_4 = 100, p_5 = 00, p_6 = 101, and$

   $k_1 = 1, k_2 = 3, k_3 = 1, k_4 = 2, k_5 = 1, k_6 = 1.$

   $x = \underbrace{101}_{101^1}\underbrace{00}_{00^1}\underbrace{100100}_{100^2}\underbrace{1100}_{1100^1}\underbrace{111}_{1^3}\underbrace{0}_{0^1}\underbrace{1}_{a}.$

Thus after T-decomposition, $x = p_6^{k_6} p_5^{k_5} p_4^{k_4} p_3^{k_3} p_2^{k_2} p_1^{k_1} a$
$= 101^1 00^1 100^2 1100^1 1^3 0^1 1$

1.2 Algorithm by Titchener and Wackrow

T-decomposition was first implemented by Titchener in 1993 [Titchener 1993]. It is a simple implementation of the principle of T-decomposition as shown above. In each parsing pass, the implementation uses a character-by-character string comparison (literal comparison) to establish whether a token under consideration is equal to p_i, and therefore whether it belongs to the run of $_i$ in one of the two patterns in step 6 of the fundamental algorithm. Note that for most tokens in most strings, this comparison is unsuccessful – finding an instance of p_i is the exception, not the rule.

Together with Scott Wackrow, Titchener improved the original algorithm in 1995 by recording the start position of each token and by skipping the comparison for tokens whose length (=difference between the start position of the current and the subsequent token) was not equal to $|p_i|$ [Titchener and Wackrow 1995]. Only if the lengths match, a literal comparison between the token and p_i is carried out. This produced a faster version of the algorithm and was published under the program name `tcalc` [Wackrow and Titchener 1998]. However, their implementation still requires each token length to be compared against $|p_i|$ in each parsing pass.

Like the fundamental algorithm, Wackrow and Titchener's algorithm executes in $O(|x|^2 \log |x|)$. This may be seen when considering a string x that consists of $|x|$ different characters. Such a string requires $|x|$ parsing passes with a total number of $|x|(|x| + 1)/2$ comparisons. The comparison of token length is $O(\log |x|)$.

In practical measurements on real computers (which absorb the $\log |x|$ complexity factor into hardware by limiting $|x|$), Wackrow and Titchener's algorithm shows a quadratic time behaviour for most strings.

1.3 Algorithm by Speidel and Yang (2003)

The authors proposed an improved T-decomposition algorithm in 2003 [Yang and Speidel 2003b].

Similar to Wackrow and Titchener's algorithm, our 2003 algorithm, called `tlist`, also uses the information such as the boundaries and lengths of tokens and T-prefixes to facilitate comparisons. However, in Wackrow and Titchener's algorithm, the length of *each* token is compared against the respective $|p_i|$ in *each* parsing pass.

In contrast, our 2003 algorithm classifies each token by length as it is created. In each parsing pass, only tokens of length $|p_i|$ are thus considered for comparison with p_i. This yields a considerable reduction in the number of length comparisons, which is reflected in a much faster execution.

This is achieved by the use of a number of intertwined doubly linked lists. One of these doubly linked lists (called *string list*) is used to record all existing tokens in the order in which they appear in x. Each list item represents exactly one token in the current parsing of x.

Each list item in the string list is simultaneously part of another doubly linked list that links all existing tokens with the same length. We call this linked list the *length list* for the length of the token. For example, all tokens of length m are linked together in the length list $list_{length}[m]$.

During each parsing pass, comparisons are thus only carried out between p_i and tokens of length $|p_i|$. This is achieved by following the respective length list for $|p_i|$ rather than the string list. The string list is used only to establish whether a token following an instance of p_i equals p_i or not. Thus, all other tokens whose lengths do not match $|p_i|$ are automatically skipped.

During the parsing, items that represent instances of p_i and are merged with subsequent tokens are removed from the length list for $|p_i|$. Items representing tokens that are merged with a preceding p_i token (or a run of p_i tokens) are moved from their old length list to the length list for the length of the new combined token. The string list and the corresponding length lists have to be updated accordingly.

The string list and old length lists are subject to removal only, and the removal point is given by the current token. Insertion into the length list for the new length, however, is more complicated. We need to retrieve the head of the new length list and then search along the list to find the correct insertion point.

To retrieve the length lists, `tlist` maintains another linked list (called the *entry list*) to store the heads of all the length lists. Finding the correct insertion point may thus require quite a number of comparisons if the length list is long.

Fortunately, there are two effects that work in our favour in practice. The first is that as tokens get longer, the linked lists necessarily get shorter as tokens are merged. The second is that most strings give rise to tokens with a variety of lengths, thus distributing the tokens across a relatively large number of linked lists.

As a result, the total number of comparisons required in our 2003 algorithm is thus generally lower than that required in Wackrow and Titchener's original algorithm. In practice, it runs significantly faster than `tcalc`. Its fundamental time complexity is nevertheless still $O(|x|^2 \log |x|)$, as may be readily shown using the same argument as for `tcalc`: There are $O(|x|)$ token elements at the start, each of which requires $O(\log |x|)$ memory and each may get visited up to $O(|x|)$ times. In practice, time behaviour is sub-quadratic in $|x|$ but still well above linear for most strings.

The amount of memory required per character in `tlist` is constant, but it is higher than that in `tcalc`, as one needs to accommodate four list references per token. More information about `tlist` may be found in [Yang and Speidel 2003b].

2 A faster algorithm

Experiments showed that our 2003 algorithm `tlist` spent a substantial part of its computational effort on two tasks:

1. traversing length lists to find the insert position for a newly merged token, and
2. comparing tokens of length $|p_i|$ with p_i even though the tokens were not equal to p_i.

This effect is particularly significant in two circumstances: during the early parsing passes of the T-decomposition and in the case of large alphabets. In these circumstances, tokens are generally still short and, in the case of large alphabets, diverse. That is, there are a lot of different tokens of the same length(s). There are only a few length lists, and they contain the majority of tokens. At this stage, the length lists are typically at their longest length during the entire process.

In summary, a large number of items in the length lists, especially coupled with a low "concentration" of p_i, can lead to significant delays in both insertion and actual parsing.

To become faster, we would need shorter length lists for faster insertion as well as fewer comparisons of unrelated codewords with p_i. However, neither the number of items in a length list nor its "concentration" of p_i depend on the algorithm, of course. Rather, they are a property of the string itself.

The solution proposed here is to abandon the length as the criterion for token membership of the "length lists" and replace it with a different criterion.

The new algorithm proposed in this paper, called `thash`, replaces the length criterion by a hash criterion. Rather than using an entry list, it uses a hash table to store what used to be the heads of the length lists. The former length lists now store tokens with equal hash value. This permits an increase in the number of lists, thereby reducing the average number of tokens per list.

The basic strategy of using a hash function in our new algorithm is as follows:

1. Hash each token into an integer (hash value).

2. Use hash values to assign a token to a linked list item, akin to the way the length of a token was used to assign the token to a linked list in the previous algorithm. Each of these linked lists (now called *hash lists*) links all tokens with a common unique hash value.

3. A hash table rather than an entry list is used to store the heads of the hash lists.

The main advantages of the new algorithm are:

1. If a suitable hash function is used, the different tokens can be distributed more evenly over the hash lists, leading to shorter lists with higher concentrations for p_i and thus fewer comparisons and faster insert times;

2. A hash list can be retrieved directly from the hash table (an array). In our previous algorithm, a search in the entry list was less efficient but unavoidable.

Our new algorithm works as follows. Assume $x \in A^+$. Let $f_{hash}(y)$ denote the hash value of a token y. Let $list_{hash}[m]$ denote the hash list corresponding to hash value m. Further, let $T_{hash}[m]$ denote the m'th entry of the hash table, which records the head of $list_{hash}[m]$. Thus, $list_{hash}[m]$ can be retrieved directly from the hash table.

1. Initialize T_{hash} with all the hash lists empty. Set a counter $i = 1$.

2. Parse x over A (i.e., each character of x is regarded as a token with length 1). Create the string list to record all initial tokens of x.

3. Determine p_i and k_i using the string list as before.
   - If $i = 1$: remove the corresponding k_i consecutive instances of p_i from the string list; for each token y that remains in the string list, compute the hash value $f_{hash}(y)$ and add a corresponding item to the end of $list_{hash}[f_{hash}(y)]$. There are now up to $\#A$ non-empty hash lists referenced from the hash table.
   - Otherwise: remove the corresponding k_i consecutive instances of p_i from their hash lists and the string list.

If the second-to-right token is also the leftmost token, the T-decomposition process is finished. Note that during the whole T-decomposition process, the corresponding item of the last character of x serves as the dummy tail of the string list. This item is only used to locate the penultimate token in the string list.

4. Retrieve the hash list $list_{hash}[f_{hash}(p_i)]$ for the hash value of p_i, $f_{hash}(p_i)$.
   (a) If $list_{hash}[f_{hash}(p_i)]$ is empty, increment i and go to step 3; otherwise
   (b) traverse the hash list $list_{hash}[f_{hash}(p_i)]$ and compare each of its items with p_i. If it equals p_i, merge it with its successor in the string list to form token items of the form of ${p_i}^l q$ where $q \neq p_i$ if $1 \leq l \leq k_i$, or ${p_i}^{k_i+1}$ otherwise. These are the same rules as introduced in Section 1.1.

   Note that the comparisons are now carried out only between p_i and the tokens from $list_{hash}[f_{hash}(p_i)]$. During this parsing, a new token is created upon each merge, while those corresponding to the merged items disappear. The disappearing token items are deleted from the string list and the corresponding hash lists. The hash value of each new token is computed, and the new item for it is inserted into the corresponding hash list. When the last item of $list_{hash}[f_{hash}(p_i)]$ has been processed, increment i and go to step 3.

Figures 1 – 11 show the T-decomposition of 011110010110 using the new algorithm. Here we use a trivial hash function, the additive hash function [Jenkins 2004]. According to the additive hash function, the hash value of a string is the sum of this string's characters. For example, the hash value of 10001 is 2, while that of 111001 is 4.

3 Choice of hash function

While we have already discussed the advantages of using the hasing approach, there are also a number of problems. Calculating a hash value for a string is computationally more expensive than measuring its length, except for very trivial hash functions.

While it is convenient to use the additive hash function to demonstrate the T-decomposition process, this function does not meet our requirements in practice. A suitable hash function $f_{hash}(x)$ is important for our new algorithm as it determines how evenly different tokens are distributed across the hash lists. Of the different categories of hash functions, we are obviously only interested in string-hashing functions. Our hash function needs to meet two requirements:

– It needs to be efficient, such that the tokens of x can be hashed quickly (the number of tokens may be large). Sophisticated hash functions, such as those designed for cryptography, do not meet this requirement.

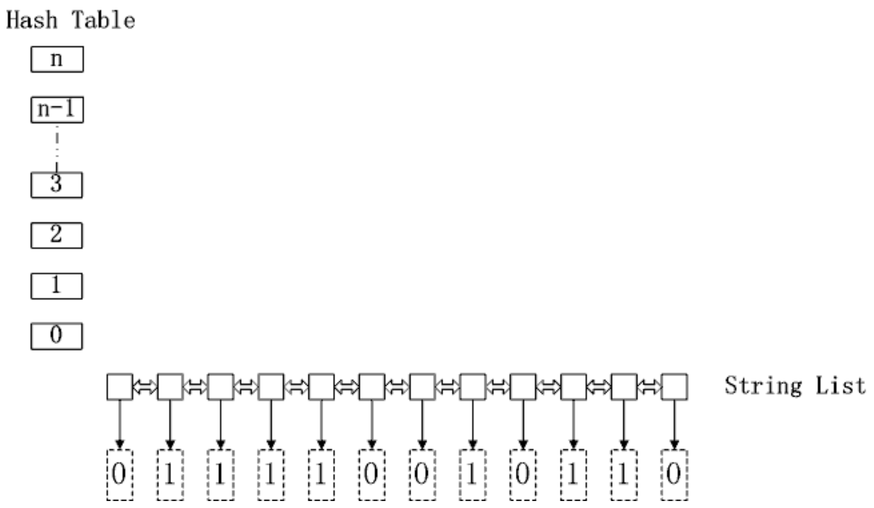

Figure 1: First parsing – initialize a hash table and set up a string list.

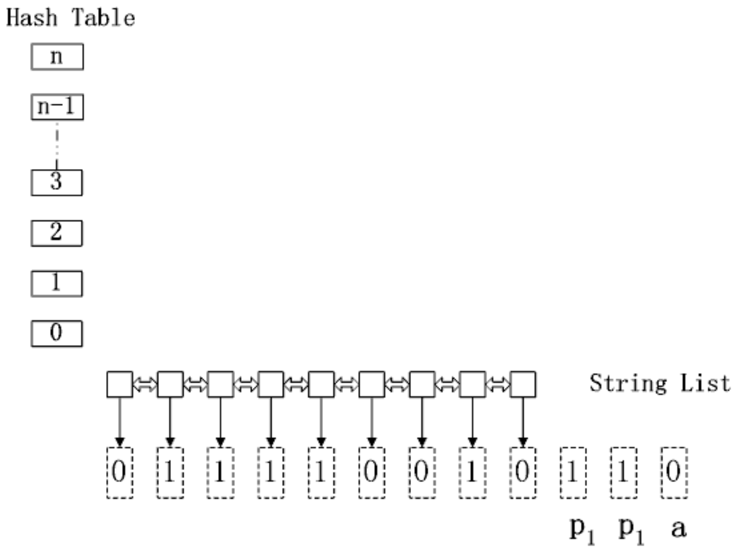

Figure 2: First parsing – identify literal character and first T-prefix copies.

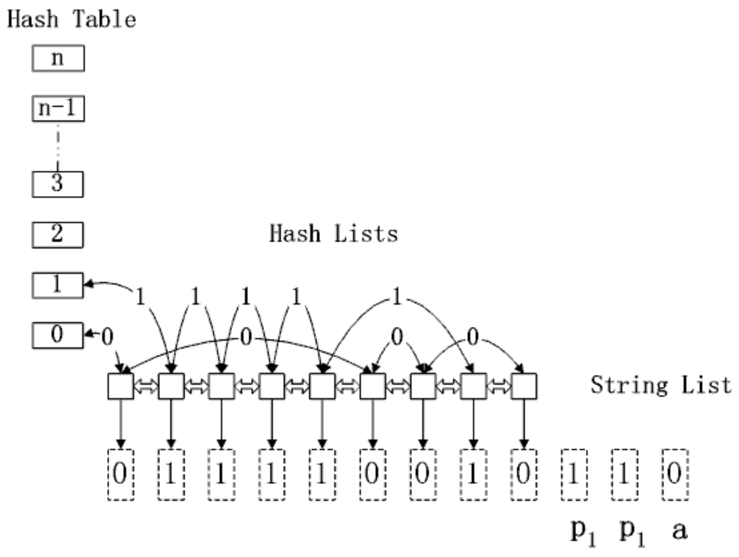

Figure 3: First parsing – compute the hash values of all the tokens, and put them into the corresponding hash lists.

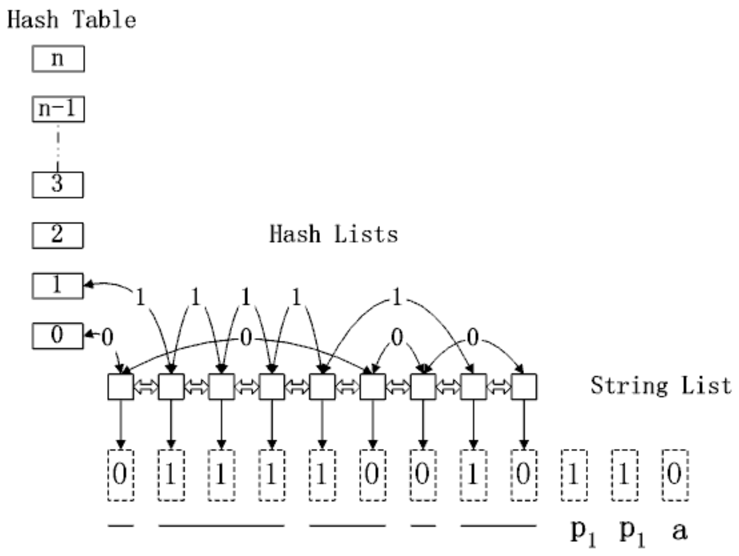

Figure 4: Second parsing on remainder of string.

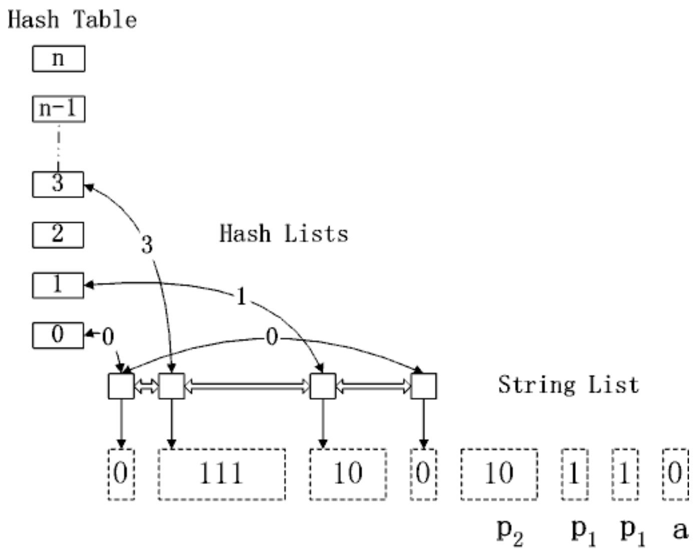

Figure 5: Second parsing – merge tokens following T-prefix copies.

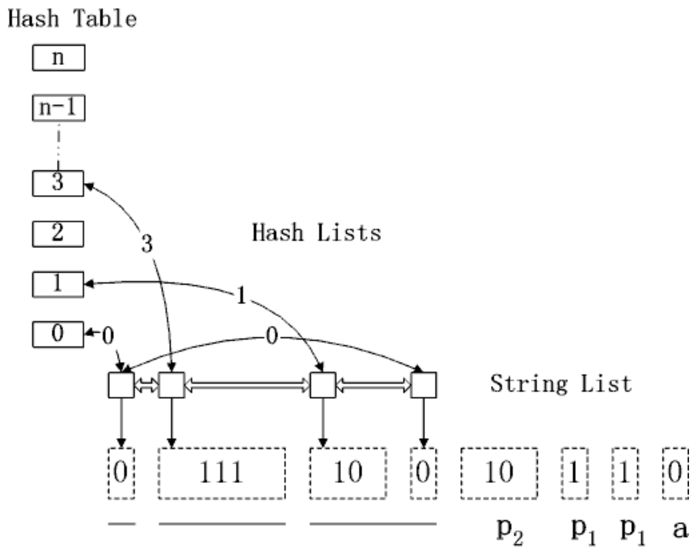

Figure 6: Third parsing on remainder of string.

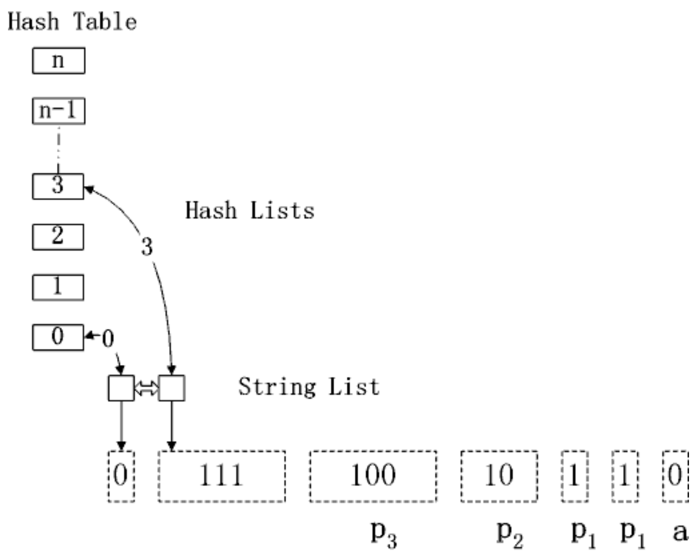

Figure 7: Third parsing – merge tokens following T-prefix copy.

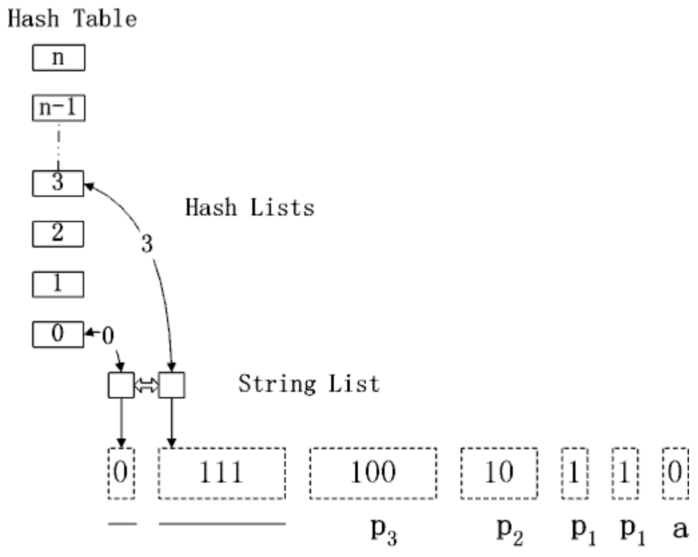

Figure 8: Fourth parsing on remainder of string.

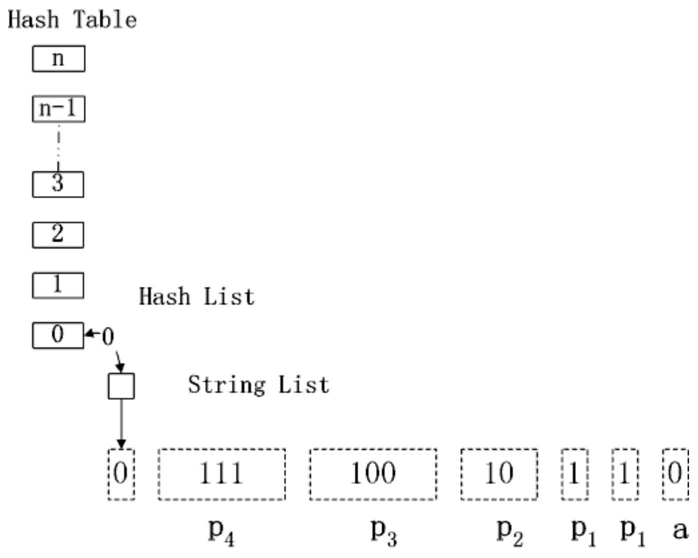

Figure 9: Fourth parsing – identify the fourth T-prefix.

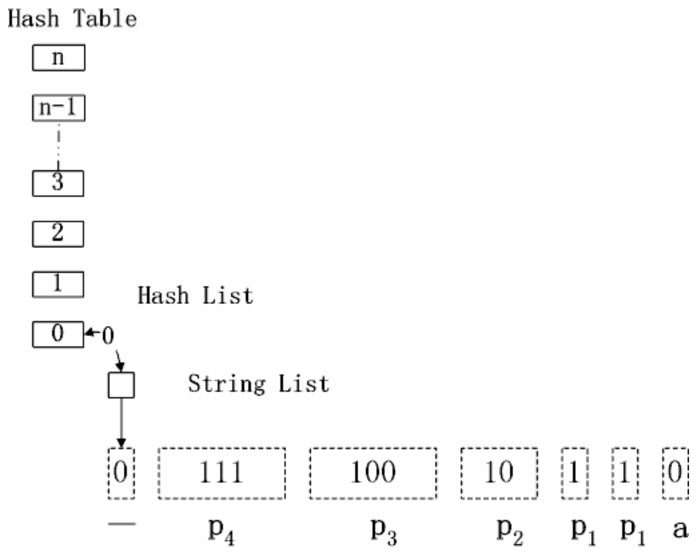

Figure 10: Fifth parsing on remainder of string.

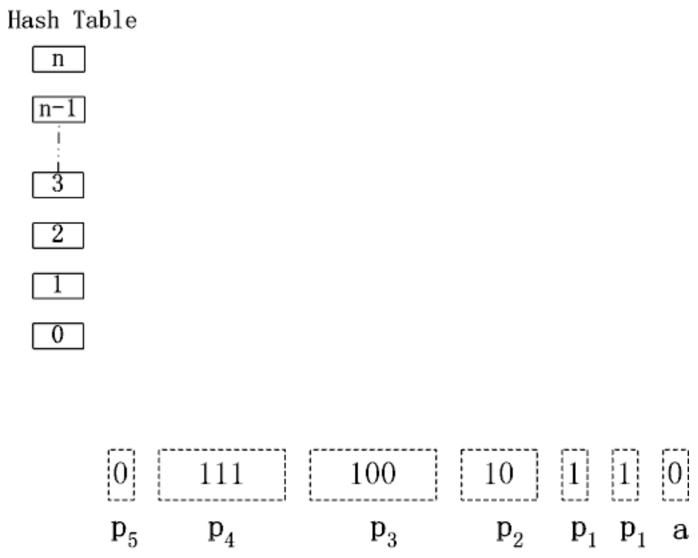

Figure 11: Fifth parsing – identify the last T-prefix.

– To reduce the overall number of comparisons, the hash values for different tokens should result in an even distribution (at least in an ideal situation). We thus need a hash function with few collisions.

Thus what we need is a fast hash function with few collisions. As these two goals are somewhat incompatible, a compromise had to be found.

We tried a number of hash functions (some of them may be found under [Jenkins 2004]), which included the additive hash function, the rotating hash function, P.J. Weinberger's hash function, Pearson's Hash function and the cyclic redundancy checksum (CRC) function.

After comparing performance, we chose the *SDBM* hash function for the implementation of our algorithm. SDBM hashing is used in the open source SDBM project [Partow 2004].

The choice of hash function makes it rather difficult to derive a reliable upper bound for the time complexity of the algorithm presented here. If we use the length of a token as its hash value, the same bound as for `tlist` applies – in this case, the two algorithms are virtually identical. However, for more complex hash functions, the time complexity may exceed $O(|x|^2 \log |x|)$ because of the extra costs associated with hash value computation. On the other hand, one would expect more complex hashes to produce fewer collsions and hence lead to faster parsing. As the next section shows, this somewhat complex picture looks very much favourable in practical situations, though.

`File index`	`File name`	`thash` [s]	`tlist` [s]	`tcalc` [s]
1	lgst3.573550	1.2	2.9	6.50
2	lgst3.586787	0.8	3.7	21.7
3	lgst3.611055	0.7	5.7	41.3
4	lgst3.651050	0.7	9.2	65.8
5	lgst3.687660	0.7	8.1	100.5
6	lgst3.766200	0.7	12.3	130.4
7	lgst3.907580	0.9	16.6	159.6
8	lgst3.925405	0.9	23.1	182.8
9	lgst3.971029	0.9	28.4	192.8
10	lgst4.000000	0.9	38.9	205.2

Table 1: Execution time comparison for `thash`, `tcalc` and `tlist`. All the strings are 2000000 bits long.

4 Comparison

This section studies the comparative performance of the three algorithms previously discussed in this paper. Three C implementations were used for this purpose: `tcalc` by Wackrow and Titchener, `tlist`, and `thash`, our implementation of our new algorithm.

The comparison was performed on a Redhat 8.0 Linux PC, using the Unix *time* command to measure the execution times.

Table 1 shows a comparison based on two-million-character strings with various degrees of pseudo-randomness. The strings used for comparison were generated via a bipartition of the *logistic map* [Weisstein 2004] and were generously supplied to us by Mark Titchener. This comparison is also displayed in Figure 12.

The strings are stored in files whose file names indicate their degree of pseudo-randomness. The larger the number in the file name, the more "random" the string (file) is presumed to be. The highest Kolmogorov-Sinai (Pesin) entropy is found in the file "lgst4.000000". All files were based on a binary alphabet $\{0, 1\}$. In other words these files consist of "0" and "1" characters, which were processed as a single bit each.

Table 2 shows the execution times for these pseudo-random strings of different lengths. All these strings were generated as the n-character suffixes of the same pseudo random 2000000-character string (lgst4.000000) generated from the logistic map. This comparison is also displayed in Figure 13.

Besides these binary alphabet-based files, we also used three English texts in our comparison. The result is shown in Table 3. These plain text files were downloaded from Project Gutenberg [Gutenberg 2004]. They are:

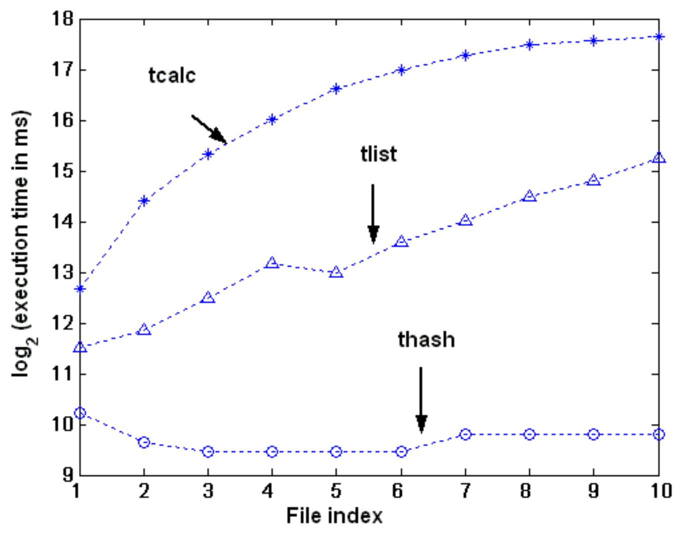

Figure 12: Execution time comparison for `thash`, `tcalc` and `tlist`. The data are from Table 1.

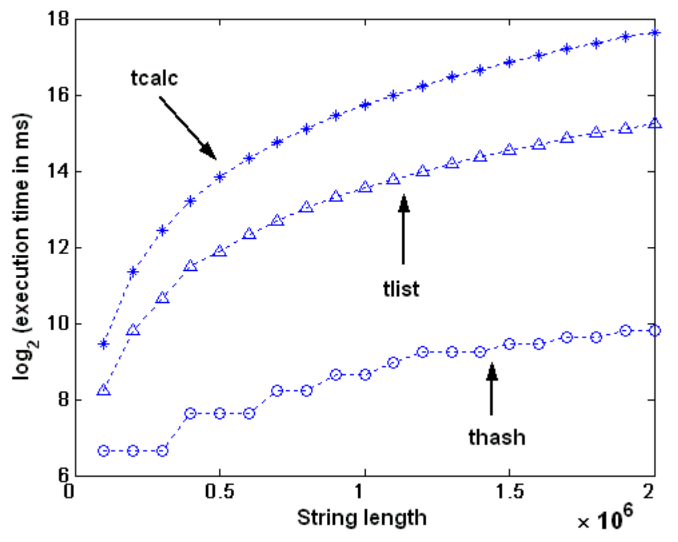

Figure 13: Execution time by string length for `thash`, `tcalc` and `tlist`. The data are from Table 2.

Length (characters)	thash [s]	tlist [s]	tcalc [s]
100,000	0.1	0.3	0.7
200,000	0.1	0.9	2.6
300,000	0.1	1.6	5.5
400,000	0.2	2.9	9.5
500,000	0.2	3.8	14.6
600,000	0.2	5.2	20.6
700,000	0.3	6.6	27.8
800,000	0.3	8.3	35.5
900,000	0.4	10.2	44.6
1,000,000	0.4	12.2	54.4
1,100,000	0.5	14.1	65.3
1,200,000	0.6	16.3	77.6
1,300,000	0.6	18.7	90.0
1,400,000	0.6	21.4	103.0
1,500,000	0.7	23.7	118.3
1,600,000	0.7	26.5	134.0
1,700,000	0.8	29.5	150.4
1,800,000	0.8	32.8	167.8
1,900,000	0.9	35.6	187.3
2,000,000	0.9	38.9	205.2

Table 2: Execution time by string length for `thash`, `tcalc` and `tlist`.

File name	thash [s]	tlist [s]	tcalc [s]
Mansfield Park (905,074 bytes)	0.7	94.3	114.9
Ulysses (1,560,001 bytes)	1.3	343.7	394.6
The King James Bible (4,445,260 bytes)	3.4	1020.9	1956.5

Table 3: Execution time comparison for `thash`, `tcalc` and `tlist`.

- *Mansfield Park* by Jane Austen, 905074 bytes plain text
- *Ulysses* by James Joyce, 1560001 bytes plain text
- *The King James Bible*, 4445260 bytes plain text

Our comparison showed that our new algorithm performs much faster than the previous algorithm in every aspect.

5 Conclusion

An efficient T-decomposition algorithm is desirable because it permits the analysis of large strings. However, real-time analysis and the analysis of very large strings have in the past been hampered by well-above-linear execution times. In all our experiments to date, `thash` executes with almost linear time for most strings of practical lengths. T-decomposition processing for large data sets is thus feasible. The new algorithm also opens the path for T-decomposition to be applied in real time data processing situations. The concept of de-crowding the string/hash lists will be further pursued in an upcoming paper by the authors [Yang and Speidel 2005]. It proposes a replacement for `thash` which can be shown to execute in $O(|x| \log |x|)$ and achieves a further significant practical speed-up compared to `thash`.

Acknowledgements

We would like to thank Mark Titchener for his constructive comments and for suggesting the use of his calibrated files for the comparative testing. We are also indebted to the anonymous referees for their extensive constructive comments, and to Cris Calude for saving us from the pitfalls of modern e-mail traffic.

References

[Ebeling et al. 2001] W. Ebeling, R. Steuer, and M. R. Titchener: *Partition-Based Entropies of Deterministic and Stochastic Maps*, *Stochastics and Dynamics*, 1(1), p. 45., March 2001.

[Guenther et al. 1997] U. Guenther, P. Hertling, R. Nicolescu, and M. R. Titchener: *Representing Variable-Length Codes in Fixed-Length T-Depletion Format in Encoders and Decoders*, Journal of Universal Computer Science, 3(11), November 1997, pp. 1207–1225. `http://www.iicm.edu/jucs_3_11`.

[Guenther 1998] U. Guenther: *Robust Source Coding with Generalized T-Codes*. PhD Thesis, The University of Auckland, 1998. `http://www.tcs.auckland.ac.nz/~ulrich/phd.pdf`.

[Gutenberg 2004] Project Gutenberg, http://www.gutenberg.net/.

[Jenkins 2004] Bob Jenkins' Web site: http://burtleburtle.net/bob/hash/doobs.html.

[Nicolescu 1995] R. Nicolescu: *Uniqueness Theorems for T-Codes*. Technical Report. Tamaki Report Series no.9, The University of Auckland, 1995.

[Nicolescu and Titchener 1998] R. Nicolescu and M. R. Titchener, *Uniqueness Theorems for T-Codes*, Romanian Journal of Information Science and Technology, 1(3), March 1998, pp. 243–258.

[Partow 2004] Arash Partow's Web site: http://www.partow.net/programming/hashfunctions.

[Speidel 2000] Ulrich Speidel: *Similarity Searches Using a Recursive String Parsing Algorithm*, Supplemental Papers for the 2nd International Conference on Unconventional Models of Computation, UMC2K, Brussels, December 13 - 16, 2000, page 54.

[Titchener 1993] M. R. Titchener: *Unequivocal Dodes: String Complexity and Compressibility* (Tamaki T-code project series), *Technique report, Computer Science Dept., The University of Auckland*, August, 1993.

[Titchener and Wackrow 1995] M. R. Titchener and S. Wackrow: *T-CODE Software Documentation* (Tamaki T-code project series), *Technique report, Computer Science Dept., The University of Auckland*, August, 1995.

[Titchener 1996] M. R. Titchener: *Generalized T-Codes: an Extended Construction Algorithm for Self-Synchronizing Variable-Length Codes*, IEE Proceedings – Computers and Digital Techniques, 143(3), June 1996, pp. 122-128.

[Titchener 1998a] M. R. Titchener, *Deterministic computation of string complexity, information and entropy*, *International Symposium on Information Theory*, August 16-21, 1998, MIT, Boston.

[Titchener 1998b] M. R. Titchener: *A Deterministic Theory of Complexity, Information and Entropy*, *IEEE Information Theory Workshop*, February 1998, San Diego.

[Titchener 1998c] M. R. Titchener, *A novel deterministic approach to evaluating the entropy of language texts*, *Third International Conference on Information Theoretic Approaches to Logic, Language and Computation*, June 16-19, 1998, Hsi-tou, Taiwan.

[Titchener 2000] M. R. Titchener: *A measure of Information*, IEEE Data Compression Conference, Snowbird, Utah, March 2000.

[Titchener et. al. 2005] M. R. Titchener and A. Gulliver and R. Nicolescu and U. Speidel and L. Staiger: *Deterministic Complexity and Entropy*, FUINE 64(1-4)1-482(2005), IOS Press.

[Wackrow and Titchener 1998] S. Wackrow and M. R. Titchener (with some minor additions by U. Guenther): tcalc.c, written in C, available from `http://tcode.tcs.auckland.ac.nz/~mark/`, under the GNU GPL.

[Weisstein 2004] Eric Weisstein's Web site: http://mathworld.wolfram.com/LogisticMap.html, also known as *mathworld*.

[Yang and Speidel 2003a] Jia Yang and Ulrich Speidel: tlist.c, written in C, available on request from the authors, under the GNU GPL.

[Yang and Speidel 2003b] Jia Yang, Ulrich Speidel: *An Improved T-decomposition Algorithm*, 4th International Conference on Information, Communications & Signal Processing, Fourth IEEE Pacific-Rim Conference On Multimedia, Singapore, December 2003. Proceedings. Vol.3, pp. 1551 - 1555.

[Yang and Speidel 2003c] Jia Yang, Ulrich Speidel: *T-information: A New Measure for Similarity Comparison*, DMTCS 2003, December 2003 (Dijon, France, July 2003). Supplemental papers, pp. 29-39.

[Yang and Speidel 2005] Jia Yang, Ulrich Speidel: *A T-decomposition algorithm with $O(n \log n)$ time and space complexity*, to appear in Proceedings of the 2005 IEEE International Symposium on Information Theory (ISIT2005), Adelaide, Australia, September 2005.