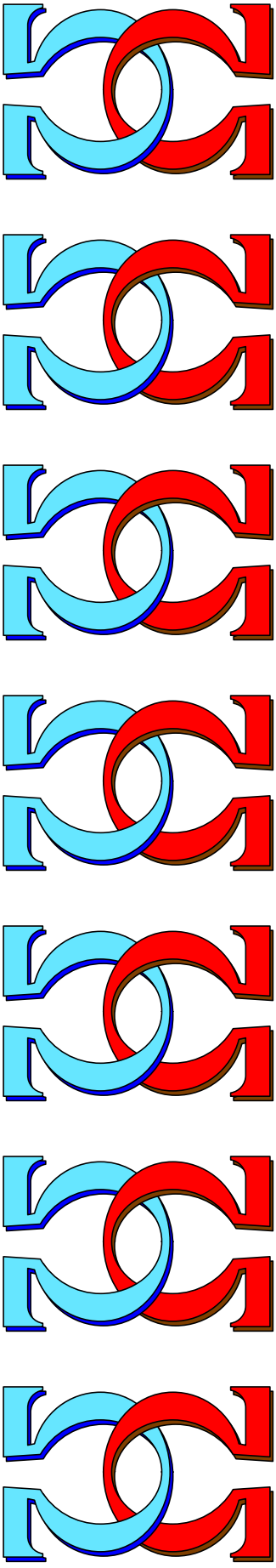

CDMTCS
Research
Report
Series

A T-decomposition algorithm with $O(n \log n)$ time and space complexity

Jia Yang and Ulrich Speidel
Department of Computer Science
The University of Auckland

CDMTCS-259
January 2005

Centre for Discrete Mathematics and
Theoretical Computer Science

A T-decomposition algorithm with $O(n \log n)$ time and space complexity

Jia Yang, Ulrich Speidel
Department of Computer Science
The University of Auckland

January 29, 2005

Abstract

T-decomposition is an algorithm that parses a finite string into a series of parameters for a recursive string construction algorithm. Initially developed for the communication of coding trees [18, 4], T-decomposition has since been studied within the context of information measures. This involves the parsing of potentially very large strings, which in turn requires algorithms with good time and space complexity. This paper presents a T-decomposition algorithm with $O(n \log n)$ time complexity.

1 Introduction

T-decomposition establishes a bijective relationship between T-code sets and finite strings. This was first recognized by Mark Titchener, who in 1993 proposed and implemented a *T-decomposition* algorithm [15]. The bijective relationship and the algorithm were later validated by Nicolescu and Titchener [11, 12].

T-decomposition has since been studied within the context of coding and information measures [19, 20, 21, 22, 2]. The authors' interest in T-decomposition lies in similarity measures [14, 27] and event detection. These two applications in particular require a fast T-decomposition algorithm that is able to process large strings with n symbols in as close to $O(n)$ time as possible.

Fundamentally speaking, T-decomposition involves the (repeated) decoding of a string into codewords of a certain type of variable length prefix code, called T-code [4]. The results of each parsing pass are then used to extract two paramaters – a codeword known as *T-prefix* and an integer called *T-expansion parameter* or *copy factor*. These are the used to extend ("T-augment") the code, which is then used for a new parsing pass. This is repeated until all T-prefixes and copy factors in the string have been identified [5].

Titchener's original algorithm is of $O(n^2)$ and experimentally shows a quadratic time behaviour for all but the most trivial strings. Simply speaking, this is caused by the fact that a string of length n may have to be parsed n times to complete the T-decomposition. In 1996, Titchener and Wackrow proposed a faster T-decomposition algorithm [18].

As each parsing pass only looks for one particular codeword p_m, Titchener and Wackrow's algorithm first compares the length of each codeword in the string with $|p_m|$. Only codewords of length $|p_m|$ are then subjected to full comparison with p_m. This allows all codewords with lengths other than $|p_m|$ to be skipped in the parsing pass because they cannot equal p_m. Nevertheless, this algorithm is $O(n^2)$ and in practice exhibits a quadratic time behaviour for most strings.
In 2003, the authors proposed a new algorithm [26]. It retains the length comparisons of Titchener and Wackrow's algorithm. However, in addition it sorts all codewords into linked lists according to their length. This avoids multiple length comparisons for the same codeword. While in practice, this algorithm exhibits a sub-quadratic behaviour for most strings, it is still an $O(n^2)$ algorithm. Our 2004 algorithm [28] also sorts codewords into linked lists, but uses a hash value rather than length as the sorting criterion to achieve a better spread across the linked lists. Since each list may still contain different codewords, individual codewords may still be parsed multiple times. While the algorithm exhibits a faster time performance than it predecessor, it is hard to prove that this is not still an $O(n^2)$ algorithm.
The present paper proposes an algorithm with a time and space complexity of $O(n \log n)$. In this paper, we will briefly describe T-codes and the principle of T-decomposition. We will then introduce our new algorithm and prove that its time-space complexity is $O(n \log n)$. Comparative experimental results will also be presented.

2 Notation

Let $A = \{a_1, a_2, a_3, ..., a_{\#A-1} a_{\#A}\}$ be a finite alphabet, where $a_i, 1 \leq i \leq \#A$ is called a *symbol* or *character*. We use A^* to denote the set of all finite strings that can be generated by concatenations of characters from A. We have λ denote the empty string and let $A^+ = A^* \backslash \{\lambda\}$. For $x, y \in A^*$, we denote the concatenation of x and y as xy. We use x^k to denote the concatenation of k copies of x, such that $x^0 = \lambda$. The length of x is denoted as $|x|$.

3 T-codes

T-codes describe a family of variable-length code sets [18, 4]. A finite code set C is a T-code set iff:

- C is an alphabet, or
- C can be derived from a T-code set via a process kown as *T-augmentation.*

T-augmentation $C_{(p)}^{(k)}$ of a code set C is defined as follows:

$$C_{(p)}^{(k)} = \{x | x = p^{k'} y, where\ 0 \leq k' \leq k\ and\ y \in C \backslash \{y\}\} \cup \{p^{k+1}\}, \tag{1}$$

where $p \in C$ and $k \in IN^+$.
We say C' can be derived from C iff $\exists\ p \in C$ and $k \in IN^+$ such that $C' = C_{(p)}^{(k)}$. We call p the *T-prefix* and k the *T-expansion parameter* of that T-augmentation. A series of m successive T-augmentations of an alphabet A is denoted $A_{(p_1,p_2,...,p_m)}^{(k_1,k_2,...,k_m)}$.

Example: Let $A = \{0, 1\}$. Then

$$A_{(1)}^{(1)} = \{0, 10, 11\}$$

and

$$A_{(1,10)}^{(1,2)} = \{0, 11, 100, 1011, 10100, 101010, 101011\}.$$

3.1 The principle of T-decomposition

T-decomposition maps every finite string $x \in A^+$ to a unique T-code set C over A, such that for any $a \in A$, xa is one of the $\#A$ longest codewords in C. This mapping between x and C is unique and bijective [11, 12]. We will now show how to determine C from a given x.

1. Parse xa over A. Thus each symbol in xa is parsed into a codeword.
2. Let $m = 1$.
3. Identify p_m as the penultimate codeword in the parsing. Identify k_m as the length of the run of consecutive of copies of p_m in the parsing that ends in the penultimate codeword, ignoring the last codeword in the parsing of xa. If the run starts at the beginning of the string, go to step 6.
4. Parse xa left-to-right. Merge consecutive codewords from the previous parsing into a new codeword of the form $p_m^{k'_m} y$, where y is a codeword in the existing parsing and $1 \leq k'_m \leq k_m$, such that $y \neq p_m$ unless $k'_m = k_m$. This is equivalent to parsing xa over $A_{(p_1,p_2,\ldots,p_m)}^{(k_1,k_2,\ldots,k_m)}$.
5. Increment m and go to step 3.
6. End.

At the end of this T-decomposition process, $x = {p_m}^{k_m} {p_{m-1}}^{k_{m-1}} \ldots {p_2}^{k_2} {p_1}^{k_1}$. We obtain $C = A_{(p_1,p_2,\ldots,p_{m-1},p_m)}^{(k_1,k_2,\ldots,k_{m-1},k_m)}$.
As may be seen, C does not depend on a. We could replace a by any other symbol from A but would still get the same C. This reflects the fact that there are $\#A$ longest codewords in C, which differ only by the last symbol.
Consider the following example. Let $x = 1000010100$ and $A = \{0, 1\}$. The T-decomposition process that determines the corresponding T-code set C is demonstrated as follows. At each step, commas are used to indicate the boundaries between codewords.

1. Parse xa over S. Thus $xa = 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, a$.
2. The penultimate codeword in the current parsing is 0, thus $p_1 = 0$. The maximum length of the run of consecutive 0s that ends in the penultimate codeword is 2. Thus $k_1 = 2$. Note that a is not counted, even if $a = 0$.
3. Now parse xa left-to-right. Merge codewords if they are of the form of $p_1^{k'_1} q = 0^{k'_1} q$ with $q \neq 0$ unless $k'_1 = k_1$. Thus xa parses as $xa = 1, 000, 01, 01, 00a$.

4. Identify p_2 as the penultimate codeword 01. The maximum length of the run of consecutive 01s that ends in the penultimate codeword is 2, thus $k_2 = 2$.

5. Parse xa left-to-right and merge codewords of the form $p_2^{k'_2}q = (01)^{k'_2}q$, where $q \neq 01$ unless $k'_2 = k_2$. This results in $xa = 1,000,010100a$.

6. Determine p_3 as the penultimate codeword 000. Identify k_3 as the maximum length of the run of consecutive 000s that ends in the penultimate codeword, i.e., $k_3 = 1$.

7. Parsing xa left-to-right, merging codeword sequences of the form $000q$. This yields $xa = 1,000010100a$. Thus $p_4 = 1$ and $k_4 = 1$. Since the run of 1 from the penultimate codeword starts at the left end of the string, the process ends here.

Thus we get $C = S^{(2,2,1,1)}_{(0,01,000,1)}$, whose longest codewords are of the form xa.

4 T-decomposition with $O(n \log n)$ time-space complexity

In T-decomposition, xa has to be parsed a total of m times. If each symbol is accessed during each parsing pass, the number of access/comparison operations will be $O(n^2)$ as m is only limited by $n = |xa|$. On the other hand, it is also obvious that each symbol has to be accessed at least once, hence $O(n)$ is an obvious lower limit for the time complexity of any T-decomposition algorithm.
The only significant operations during each parsing pass are the mergers of the p_m with subsequent codewords. That is, in each pass, the algorithm primarily looks for occurrences of p_m in the codewords from the previous parsing pass of xa. The number of occurrences is generally small compared to the total number of codewords in the parsing. Our algorithm's goal is thus to avoid parsing other codewords than p_m in the respective parsing pass.

4.1 Data structures used in our algorithm

A sub-$O(n^2)$ algorithm requires data structures that permit the bypassing of irrelevant codewords during parsing. Like our 2003 and 2004 algorithms, our new approach uses a number of intertwined doubly linked lists to represent the string and its current parsing. Each element of these lists references exactly one specific occurrence of a codeword in the current parsing of xa. Each element is also a member of exactly two doubly linked lists, the *string list* and a *match list*.
The *string list* records all codeword instances in the current parsing of x in sequence. I.e., each element e of the string list contains direct references to the previous and next elements in the string list, $pred_s(e)$ and $succ_s(e)$, which correspond to the previous and next codewords in x, respectively. The dummy character a is not used in our algorithm. Simultaneously, each item in a string list is also part of another doubly linked list called a *match list*. Each match list links elements representing instances of identical codewords in the current parsing in the order in which they appear in xa. An element e representing a codeword y is linked to the elements representing the previous and next occurrences of

y via the references $pred_m(e)$ and $succ_m(e)$ respectively. All codewords are thus classified according to the match list that they belong to.
For codewords at the beginning or end of the respective list, $pred_s(e)$, $succ_s(e)$, $pred_m(e)$, and $succ_m(e)$ return a special reference (NULL), indicating that this is the first or last codeword in the list.
Each codeword element e representing an instance of the codeword y has an ID assigned to it that is unique for each match list. This unique ID, denoted $uid(e)$, serves as a reference back to the header of the corresponding match list for e. Instances of identical codewords thus share a common unique ID. We may thus write $uid(y)$ without ambiguity in a slight abuse of notation. Similarly, the match list header of the list containing e is denoted $match(uid(e))$ or, where $uid(e) = uid(y)$, as $match(uid(y))$, $match(e)$ or $match(y)$ etc. Similar abuse of notation may occur in other places where this does not gives rise to ambiguity.
Each match list header $match(y)$ contains a reference to the first element first($match(y)$) and to the last element last($match(y)$) in the match list. It also contains a *level visitation indicator* $level(match(y))$, which indicates the number i of the parsing pass during which codewords y were last absorbed by one or more preceding T-prefix(es) p_i into codewords of the form $p_i^j y$. When a match list header is first created, the associated match list is still empty. first($match(y)$) and last($match(y)$) are thus NULL references initially, and $level(match(y))$ is 0.
Last but not least, the match list header includes another reference, $higher(y)$. If not NULL, it references the match list for the codeword elements that represent $p_m y$.
We further define the absorption of an element $e' = succ_s(e)$ into e, denoted $absorb(e, e')$ as the following sequence of six operations:

$$\begin{aligned} pred_m(succ_m(e)) &:= pred_m(e) \\ succ_m(pred_m(e)) &:= succ_m(e) \\ pred_s(succ_s(e')) &:= e \\ succ_s(e) &:= succ_s(e') \\ pred_m(succ_m(e')) &:= pred_m(e') \\ succ_m(pred_m(e')) &:= succ_m(e') \end{aligned}$$

The first two operations remove e from its match list, the third and fourth operation remove e' from the string list, and the last two operations remove e' from its match list. We are thus left with an element e that is (temporarily) not in a match list, and an element e' that is neither in the string list nor in a match list.

4.2 The algorithm

The parameters for the T-code set C for which xa is one of the longest codewords are determined as follows from x by using our new algorithm:

1. Parse x left-to-right over A. Thus each symbol in x is parsed into a codeword. Note that x rather than xa is parsed here as a carries no useful information. In this parsing pass, set up the string list that records all the codeword instances, which are just symbols at this stage. Simultaneously, create a match list for each

first occurrence of a distinct symbol a_i in the string. Subsequent occurrences of a_i are then added to this list. The IDs of the initial set of match lists are thus given by the indices i of the corresponding symbols $a_i \in A$. Store the unique ID of the respective match list in each codeword element and set $\mathit{level}(\mathit{match}(a_i)) := 0$.

2. Initialize two counters $m := 1$ and $\ell = \#A + 1$.

3. Identify the last element in the string list as p_m. Set $k_m = 1$. Continue to move along the string list towards the head of the list and increment k_m while the respective element has the same ID as the element identified as p_m. Stop once an element has a different ID or the beginning of the string list is reached. Remove the k_m elements representing copies of p_m from the string list and from $\mathit{match}(p_m)$. If the string list is now empty, finish.

4. Retrieve the match list header for p_m, $\mathit{match}(p_m)$ via its unique ID reference $\mathit{uid}(p_m)$ stored in the element representing p_m. If $\mathit{match}(p_m)$ represents an empty list, increment m and continue at step 3.

5. Let $e = \mathrm{first}(\mathit{match}(p_m))$. Initialize a counter $k'_m = 1$.

6. While $k'_m < k_m$ and $\mathit{uid}(e) = \mathit{uid}(\mathit{succ}_s(e))$, $\mathit{absorb}(e, \mathit{succ}_s(e))$ and increment k'_m. Note that each of the absorption operations in this while loop corresponds to the disappearance of one codeword boundary ("comma") in the parsing of xa.

7. Let $e' := \mathit{succ}_s(e)$ and let y denote the codeword referenced by e'. Then $\mathit{absorb}(e, \mathit{succ}_s(e))$. This operation also corresponds to the disappearance of one codeword boundary.

8. In the string list, the element e already represents a new codeword of the form $p_m^{k'_m} y$. However, it still carries the ID of p_m and does not belong to a match list at this point. This means that we need to give e a new ID and insert it into its corresponding new match list. For this purpose, retrieve $\mathit{uid}(e')$ and use it as a reference to look up the match list header $\mathit{match}(e')$.

9. If $\mathit{level}(e') \neq m$, continue at step 10, else continue at step 11.

10. Create new match lists $\mathit{match}(p_m^j y)$ for $1 \leq j \leq k'_m$. Associate each match list with a unique ID in the range of ℓ to $\ell + k'_m - 1$ by incrementing ℓ after each assignment. Set $\mathit{level}(\mathit{match}(p_m^j y)) := 0$ for each j. Set $\mathit{higher}(\mathit{match}(p_m^{j-1} y)) := \mathit{match}(p_m^j y)$ for each j. Note that this corresponds to a fixed number of operations for each of the k'_m absorptions above. Finally, append e to $\mathit{match}(p_m^{k'_m} y)$ and set $\mathit{level}(e') := m$. Then continue at step 12.

11. In this case ($\mathit{level}(e') = m$) there may be an existing match list that we can insert e into. We simply follow the header reference $h := \mathit{match}(e')$, which exists at this point. Initialize a counter $k''_m = 0$. While $k''_m < k'_m$ and $\mathit{higher}(h)$ exists, dereference $h := \mathit{higher}(h)$ and increment k''_m. If $k''_m = k'_m$, simply insert e at the end of h and assign $\mathit{uid}(e) := \mathit{uid}(h)$. Otherwise create new match lists $\mathit{match}(p_m^j y)$ for $k''_m < j \leq k'_m$, assign a unique ID to each new match list, starting with ℓ and incrementing

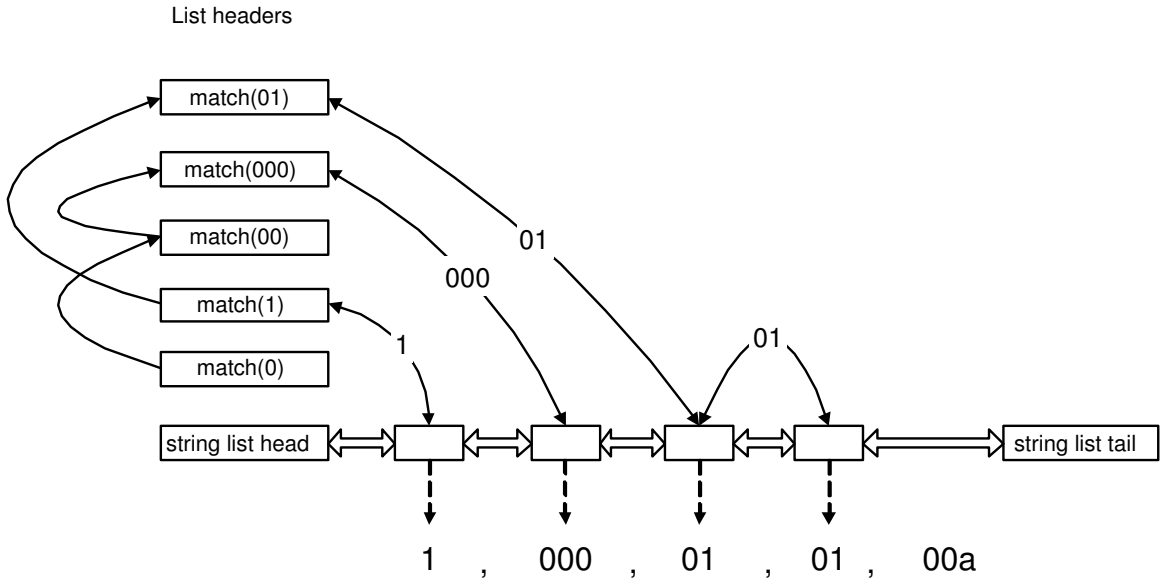

Figure 1: String list and match lists after parsing $1000010100a$ with $p_1 = 0$ and $k_1 = 2$. The boxes to the left depict the match list headers in order of creation, starting with *match*(0) at the bottom. The single arrows left of the match list headers correspond to the *higher*() references for the lists. The horizontal boxes correspond to the codeword elements, linked by the string list (thick double arrows) and the respective match lists (curved and labelled double arrows). For orientation, the corresponding codewords in the string are pointed out at the bottom. Note that the match list *match*(0) contains no codewords as it represents p_1.

ℓ after each assignment. We also set $higher(match(p_m^j y)) := match(p_m^{j+1} y)$ for each $k''_m \leq j < k'_m$ and $level(match(p_m^j y)) := m$ for each $k''_m \leq j \leq k'_m$. Note that this also corresponds to at most a fixed number of operations for each of the k'_m absorptions above. We then insert e into $match(p_m^{k'_m} y)$.

12. If $match(p_m)$ is not empty, continue at step 5, otherwise increment m and continue at step 3.

Figure 1 depicts the configuration of the string list, the match lists, and the *higher*() references for the match list headers after the first parsing pass with $p_1 = 0$ and $k_1 = 2$.

5 Complexity Analysis

We will first consider the memory requirements for the algorithm. The algorithm's significant memory requirements occur in two areas: codeword elements in the intertwined doubly linked lists and list headers. After the first sorting pass in which the elements are created, there are $n = |x|$ such elements. No codeword elements are newly created after the first pass as existing ones are reused. As a result, no additional memory requirement for codeword elements arises from subsequent parsing. Each codeword element requires $O(\log n)$ memory as it needs to reference:

- up to four codeword elements at any position in the lists. Each of these references requires $O(\log n)$ memory, and
- its match list, which requires $O(\log n)$ memory for the unique ID.

The total memory requirement for the codeword elements is thus $O(n \log n)$.
The second major memory requirement arises from the match list headers. Match list headers may get created on two occasions:

- during the first parsing pass of x, up to $\#A \leq n$ list headers may be created, and
- whenever a codeword element absorbs subsequent codeword elements in the string list.

If a sequence of absorptions creates a codeword $p_m^j y$, then this requires j absorptions and up to j new list headers may be created. In total, we cannot create more than n list headers this way as each absorption results in the removal of one codeword boundary from x and the number of codeword boundaries in x is $n-1$. Each list header needs to reference one other list header and up to two arbitrary codeword elements (head and tail of the list). The memory requirement per list header is $O(\log n)$. Across all list headers it is thus $O(n \log n)$.
All other memory requirements are of lower order. The total space complexity of the algorithm is thus $O(n \log n)$.
The time complexity of the algorithm's first parsing pass is $O(n \log n)$ as n elements of size $O(\log n)$ are created and appended to the string list and to the end of their respective match list. A match list also takes $O(\log n)$ to create if it does not already exist.
As for the algorithm's time complexity after the first parsing pass, retrieval of $pred_s(e)$, $pred_m(e)$, $succ_s(e)$, $succ_m(e)$, $level(e)$, and $uid(e)$, as well as $match(e)$ (for elements e that are already in the correct match list) are all $O(\log n)$. This arises from the need to read references sized $O(\log n)$. This also applies to $absorb(e, succ_s(e))$, which is simply a combination of a fixed number of the previous operations.
The number of these operations during the algorithm is $O(n)$ – at least in contexts other than the lookup or creation of list headers for codewords newly created through absorption. This arises from the fact that each removal of a particular codeword boundary in step 6 or 7 is always associated with at most a constant number ($O(1)$) of these operations. The total cost in these contexts is thus $O(n \log n)$ as there are at most $n-1$ codeword boundaries to be removed.
As mentioned above, the two exceptions to the rule are the lookup and the creation of match list headers for codewords that have been newly created through absorption. These initially require retrieval of the reference $higher(h)$, which is $O(\log n)$. Additionally, they may require repeated retrieval of $higher(h)$ or repeated creation of new match list headers in steps 10 or 11. Note however that each creation of a new header and each retrieval of $higher(h)$ is uniquely associated with exactly one absorption operation – both are counted using the same k'_m. Since there cannot be more than n absorptions in total, and the cost of an individual creation operation or retrieval of $higher(h)$ is $O(\log n)$, the total cost for these operations across the entire algorithm is also $O(n \log n)$.
It should be noted that the $\log n$ component of the time complexity arises exclusively from the handling of $O(\log n)$-sized references. In an implementation with fixed reference

size (and hence an implied limit on n), the behaviour of the algorithm is linear in time. This is usually the case for implementations on a conventional computer.

6 Comparison

A complexity analysis of an algorithm may say little about the performance of an actual implementation using practical data. Comparisons with algorithms for which complexity analysis is difficult thus require a comparative experiment. This applies in particular to comparison with our 2004 algorithm, which uses hashing to create what in hindsight may be called "impure" match lists.
In this section we present a comparative study of the time efficiencies of the new algorithm and the other three algorithms previously discussed in this paper. Four C programs were used for this purpose. They were

1. **ftd** (fast T-decomposition): our implementation of the new algorithm.
2. **thash**: our implementation of our 2004 algorithm.
3. **tlist**: our implementation of our 2003 algorithm.
4. **tcalc**: Wackrow and Titchener's implementation of their algorithm.

The comparison was performed on a Redhat 8.0 Linux PC. The Unix *time* command was used to measure the execution times of these programs.
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 the *logistic map* [24]. This comparison is also displayed in Figure 2.
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, the files consist of "0" and "1" characters.
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) obtained from the logistic map. This comparison is also shown in Figure 3.
We also used three English texts in our comparison, shown in Table 3. These plain text files were downloaded from Project Gutenberg [6]. They are:

- *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

Figures 4 – 6 show the experimental results of comparing **ftd** and **thash** based on strings of different lengths. These figures confirm that the time complexity of **thash** exceeds

File index	File name	**ftd** [s]	**thash** [s]	**tlist** [s]	**tcalc** [s]
1	lgst3.573550	0.41	1.24	2.90	6.51
2	lgst3.586787	0.42	0.93	3.67	21.75
3	lgst3.611055	0.42	0.70	5.72	41.32
4	lgst3.651050	0.47	0.70	9.23	65.78
5	lgst3.687660	0.48	0.71	8.06	100.45
6	lgst3.766200	0.41	0.69	12.34	130.43
7	lgst3.907580	0.61	0.92	16.62	159.62
8	lgst3.925405	0.60	0.91	23.13	182.80
9	lgst3.971029	0.61	0.96	28.38	192.81
10	lgst4.000000	0.62	0.91	38.91	205.24

Table 1: Execution time comparison for **ftd**, **thash**, **tlist** and **tcalc**. All strings are $2,000,000$ bits long.

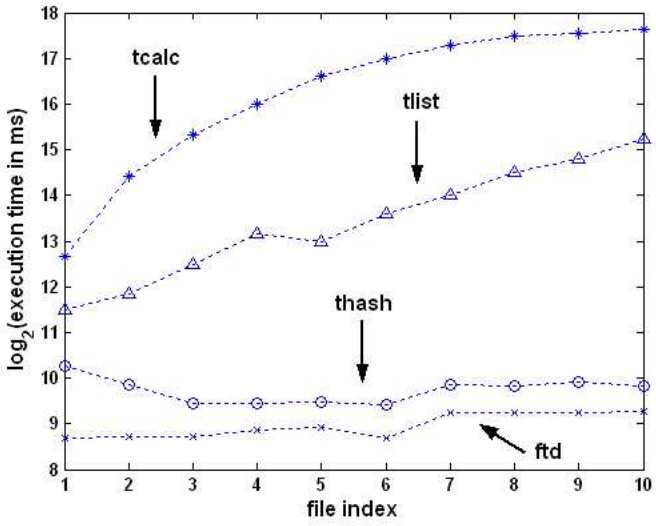

Figure 2: Execution time comparison for **ftd**, **thash**, **tlist** and **tcalc**. The data is taken from Table 1.

Length (characters)	**ftd** [s]	**thash** [s]	**tlist** [s]	**tcalc** [s]
100,000	0.03	0.05	0.28	0.66
200,000	0.05	0.08	0.85	2.56
300,000	0.10	0.11	1.61	5.47
400,000	0.12	0.18	2.59	9.54
500,000	0.15	0.21	3.83	14.62
600,000	0.18	0.24	5.23	20.55
700,000	0.22	0.31	6.60	27.76
800,000	0.25	0.35	8.27	35.45
900,000	0.30	0.40	10.15	44.55
1,000,000	0.34	0.43	12.20	54.41
1,100,000	0.37	0.48	14.07	65.31
1,200,000	0.37	0.55	16.30	77.60
1,300,000	0.41	0.55	18.74	89.95
1,400,000	0.46	0.58	21.41	103.04
1,500,000	0.47	0.67	23.71	118.27
1,600,000	0.50	0.71	26.45	134.00
1,700,000	0.56	0.75	29.46	150.40
1,800,000	0.56	0.83	32.76	167.78
1,900,000	0.58	0.87	35.63	187.33
2,000,000	0.62	0.90	38.99	205.19

Table 2: Execution time by string length for **ftd**, **thash**, **tlist** and **tcalc**

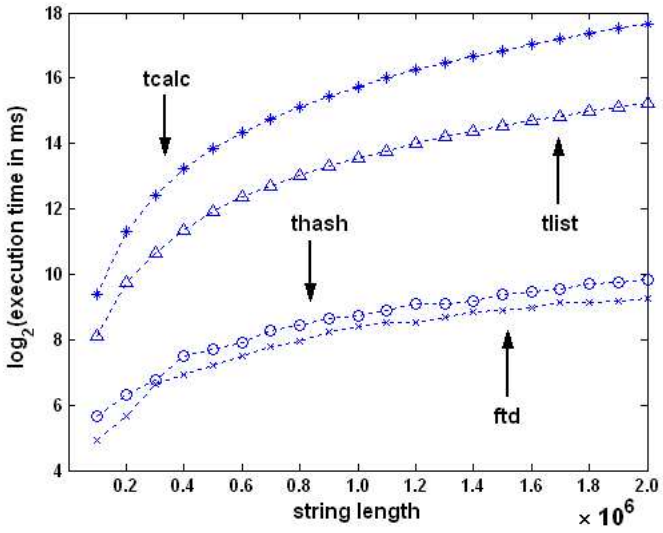

Figure 3: Execution time by string length for **ftd**, **thash**, **tlist** and **tcalc**. The data is taken from Table 2.

File name	**ftd** [s]	**thash** [s]	**tlist** [s]	**tcalc** [s]
Mansfield Park (905,074 bytes)	0.49	0.70	94.35	114.85
Ulysses (1,560,001 bytes)	0.83	1.22	343.7	394.62
The King James Bible (4,445,260 bytes)	2.26	3.25	1020.94	1956.57

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

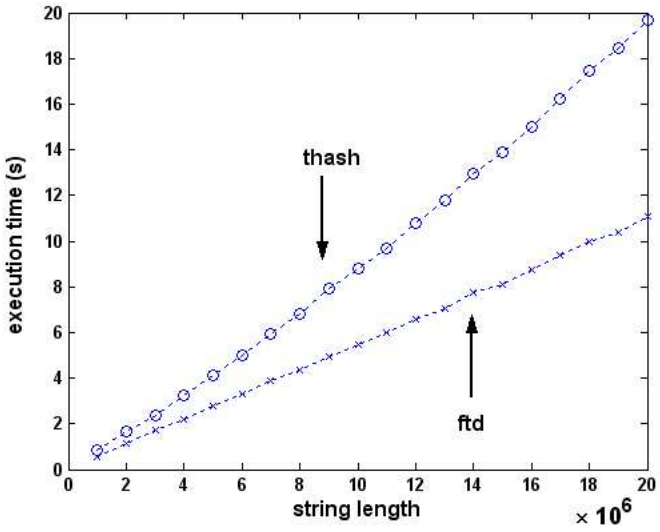

Figure 4: Execution time by string length for **ftd** and **thash**. The strings were the n-character prefixes of a string that consists of the first 20,000,000 digits of π.

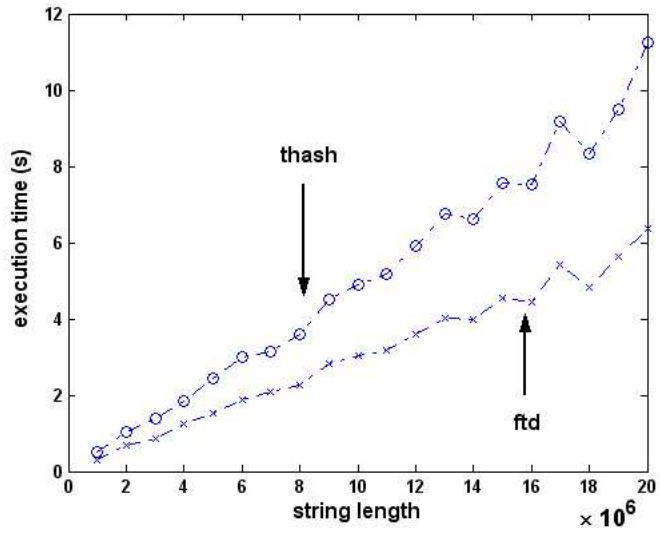

Figure 5: Execution time by string length for **ftd** and **thash**. The strings were the n-character prefixes of a 20,000,000-character string generated via the logistic map using the same parameters as lgst4.000000. These strings are assumed to be pseudo-random.

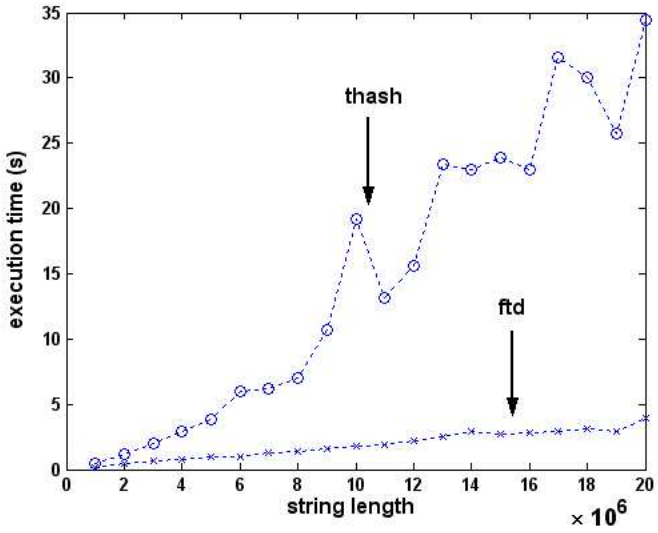

Figure 6: Execution time by string length for **ftd** and **thash**. The strings were the n-character prefixes of a 20,000,000-character string generated via the logistic map using the same parameters as lgst3.573550. The degrees of randomness of these strings are expected to be low.

that of **ftd**. The digits of π used in our experiment (Figure 4) were downloaded from the *HighPi* Web site [1].
The experimental results show that our new algorithm is generally faster than the previous algorithms.

7 Conclusion

The time efficiency of a T-decomposition algorithm is important if it is to be used in a real-time scenario or for the analysis of larger data sets. Like its immediate predecessors, the new algorithm's space complexity is $O(n \log n)$. This is the same as that of Titchener and Wackrow's algorithm. However, the time complexity of our new algorithm is also $O(n \log n)$, while those of previous T-decomposition algorithms are larger and can generally only be shown to be $O(n^2)$. Experimental results also demonstrate that our new algorithm is generally faster than the previous algorithms.
This new algorithm makes T-decomposition processing for large strings more feasible, and opens up T-decomposition-based analysis for real time applications. The implementation, **ftd**, is available from the authors.
We would like to thank Mark Titchener for his constructive comments and for making his calibrated files available for comparative testing.

References

[1] `http://highpi.4t.com/index.html`, also known as *HighPi*.

[2] 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.

[3] 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`.

[4] 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`.

[5] U. Guenther: *T-Complexity and T-Information Theory – an Executive Summary*. CDMTCS Report 149, Centre for Discrete Mathematics and Theoretical Computer Science, The University of Auckland, February 2001. `http://www.tcs.auckland.ac.nz/CDMTCS/researchreports/149ulrich.pdf`.

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

[7] G.R. Higgie:*Analysis of the families of variable-length self-synchronizing codes called T-Codes*, PhD thesis, The University of Auckland,1991.

[8] `http://www.student.carleton.edu/~holschuj/`.

[9] `http://burtleburtle.net/bob/hash/doobs.html`.

[10] `http://burtleburtle.net/bob/hash/perfect.html`.

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

[12] 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.

[13] `http://www.partow.net/programming/hashfunctions`.

[14] 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.

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

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

[17] M.R. Titchener and J.J. Hunter:*Synchronization Process for the Variable-Length T-codes*. IEE Proceedings — Computers and Digital Techniques, 1985, 133(1), pp. 54–64.

[18] 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.

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

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

[21] 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.

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

[23] 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.

[24] Mathworld: `http://mathworld.wolfram.com/LogisticMap.html`

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

[26] 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.

[27] 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.

[28] Jia Yang, Ulrich Speidel: *A fast T-decomposition algorithm*, submitted to Journal of Universal Computer Science.

[29] Jia Yang, Ulrich Speidel: *A T-decomposition algorithm with $O(n \log n)$ time and space complexity*. CDMTCS Report 259, Centre for Discrete Mathematics and Theoretical Computer Science, The University of Auckland, February 2005. `http://www.tcs.auckland.ac.nz/CDMTCS/researchreports/259ulrich.pdf`.

[30] F. Zolghadr, B. Honary and M. Darnell: *Statistical real-time channel evaluation (SRTCE) technique using variable-length T-codes*, IEE Proceedings – Communications, speech and vision, 136(4), August 1989, pp. 259-266.