An Improved T-Decomposition Algorithm

Jia Yang, Ulrich Günther
Department of Computer Science
University of Auckland
Auckland, New Zealand

CDMTCS-203
December 2002

Centre for Discrete Mathematics and Theoretical Computer Science
An Improved T-Decomposition Algorithm

Jia Yang, Ulrich Günther

December 5, 2002

Abstract

The bijective relationship between T-Code sets and finite strings discovered by Nicolescu and Titchener is of interest in coding, information measurement and pattern matching. The T-decomposition algorithm that accomplishes the mapping from the string to the T-code set was implemented in 1996 by Wackrow and Titchener. This paper introduces a new algorithm that permits faster and more efficient T-decomposition. Initial experimental results are also given.

1 Introduction

T-Codes [3, 4, 11] are a family of variable-length codes. It is possible to reconstruct any T-Code set from any one of its longest codewords, using an algorithm called T-decomposition. This algorithm was discovered empirically by Mark Titchener and later proved by Nicolescu [2, 8]. The algorithm has since led to the derivation of information and entropy measures [10, 7, 9, 13] that have been shown to be related to known entropies [14]. The authors’ particular interest is in the area of pattern matching and mutual information in strings, which utilizes these results. In order to be useful in practical applications, e.g., in search engines, we require an efficient T-decomposition algorithm.

In this paper, we will first describe the principle of the T-decomposition algorithm. We will then discuss the only previously existing implementation, followed by a description of our own approach and a presentation of its results. In the following, we will use the notation from [11].

Let $S$ be an alphabet and consider a string $x \in S^+$ and a letter $y \in S$. Then there exists a T-Code set $S_{\{p_1,p_2,\ldots,p_n\}}^{(k_1,k_2,\ldots,k_n)}$ such that $xy$ is one of its longest codewords and $S_{\{p_1,p_2,\ldots,p_n\}}^{(k_1,k_2,\ldots,k_n)}$ is independent of $y$, i.e., given $x$, all strings $xy$ are longest codewords in $S_{\{p_1,p_2,\ldots,p_n\}}^{(k_1,k_2,\ldots,k_n)}$.

$S_{\{p_1,p_2,\ldots,p_n\}}^{(k_1,k_2,\ldots,k_n)}$ may be derived from $xy$ as follows:

1. Let the current code set be $C = S$ and set the level counter $n = 0$.

2. Decode $xy$ over $C$. This yields $xy$ as series of codewords from $C$. If $xy$ decodes as a single codeword, stop.

3. Identify the second-to-last codeword in the decoding over $C$ and call it $p_{n+1}$.
4. The second-to-last codeword, \( p_{n+1} \), may be the last in a run of \( p_{n+1} \) in the decoding of \( xy \). Let \( k_{n+1} \) be the total length (in codewords) of this run.

5. T-augment \( C \) with \( p_{n+1} \) and \( k_{n+1} \) to yield \( C_{\left[k_{n+1}\right]}^{\left[p_{n+1}\right]} = S_{\left[p_{1}, p_{2}, \ldots, p_{n+1}\right]}^{\left[k_{1}, k_{2}, \ldots, k_{n+1}\right]} \).

6. Set \( n = n + 1 \) and \( C = S_{\left[p_{1}, p_{2}, \ldots, p_{n+1}\right]}^{\left[k_{1}, k_{2}, \ldots, k_{n+1}\right]} \). Continue at step 2.

Example: Let \( xy = 011111111010111101011 \) and let \( S \) be the binary alphabet \( S = \{0, 1\} \). We can reconstruct the T-Code set from this string as follows:

- First, we decode the string over \( S \). The direction of the decoding is left to right (i.e., the leftmost symbols/codewords are decoded first) and the result is (dots are used to indicate the boundaries between codewords):
  
  \[ 0.1.1.1.1.1.1.1.1.0.1.0.1.0.1.1.1.1.1.1.0.1.1 \]

  We find that the second-to-last codeword in this decoding is 1, so \( p_1 = 1 \). As there are no copies of 1 that immediately precede the second-to-last codeword, \( k_1 = 1 \).

- Now we decode the string again. This time we decode it over the T-code set \( S_{\left[1\right]}^{\left[1\right]} = S_{\left[1\right]}^{\left[1\right]} \). The result is:

  \[ 0.1.1.1.1.1.1.1.0.1.0.1.1.1.1.1.0.1.1 \]

  Note that all codeword boundaries that followed the T-prefix \( p_1 \) in the previous decoding have disappeared, except those after a run of \( k_1 + 1 = 2 \) copies of \( p_1 = 1 \). The second-to-last codeword in this decoding is 10, so \( p_2 = 10 \). As there are no copies of 10 that immediately precede the second-to-last codeword, \( k_2 = 1 \).

- Now we decode the string over \( S_{\left[p_1, p_2\right]}^{\left[1, 1\right]} = S_{\left[1, 1\right]}^{\left[1, 1\right]} \):

  \[ 0.1.1.1.1.0.1.0.1.0.1.1.1.1.0.1.1 \]

  Note again that all codeword boundaries following \( p_2 = 10 \) have disappeared, except the one after the run of \( k_2 + 1 = 2 \) copies of \( p_2 = 10 \) in the middle of the string. The second-to-last codeword in this decoding is 11, and there is one copy of 11 that immediately precedes the second-to-last codeword. So \( p_3 = 11 \) and \( k_3 = 2 \).

- Decoding the string over \( S_{\left[1, 1, 10, 11\right]}^{\left[1, 1\right]} \), we get: \( 0.1111111.11010.1011.1111011 \)

  Thus \( p_4 = 1011 \). Since there are no copies of 1011 that immediately precede the second-to-last codeword, \( k_4 = 1 \).

- Decode the string over \( S_{\left[1, 10, 11, 10, 11\right]}^{\left[1, 1, 2\right]} \), and the result is:

  \[ 0.11111.11101.010111111011 \]

  We now know from this decoding that \( p_5 = 11010 \) and \( k_5 = 1 \).

- Decode the string over \( S_{\left[1, 10, 11, 10, 110, 110\right]}^{\left[1, 1, 2, 1\right]} \) and get the result:

  \[ 0.111111111010101111111011 \]

  So \( p_6 = 1111111 \) and \( k_6 = 1 \).

- Decode the string over \( S_{\left[1, 10, 11, 10, 110, 110, 11, 1110, 111111\right]}^{\left[1, 1, 2, 1, 1\right]} \). The result is:

  \[ 0.1111111111010101111111011 \]

  i.e., \( p_7 = 0 \) and \( k_7 = 1 \).
• Thus decode the string over $S_{(1,1,2,1,1,1,1,1,1,1,1,0)}^{(1,1,2,1,1,1,1,1,1,1,1,1)}$. The result is a single codeword: 0111111110101111111011, which means that the T-decomposition has been completed. The T-Code set obtained, $S_{(1,1,2,1,1,1,1,1,1,1,1,1,1,0)}^{(1,1,2,1,1,1,1,1,1,1,1,1,1)}$ has the string $xy = 0111111110101111111011$ as one of its longest codewords.

Simple implementations of the above algorithm are inefficient for long strings. The reason for this lies in the fact that the T-decomposition algorithm described above involves the complete decoding of the entire string at each level of T-augmentation. However, for most long strings, many of the decoding passes are very similar. In fact, the only changes between two subsequent decoding passes are:

• The run of $p_{n+1}$ merges with the previously last codeword and decodes as a single codeword.

• Starting from the left end of the string, any run of up to $k_{n+1}$ copies of $p_{n+1}$ in the previous decoding merges with the subsequent codeword and decodes as a single codeword.

In large strings of a sufficiently random nature, the latter item is the exception rather than the rule. Apart from these changes, the two subsequent decodings are identical. This represents a lot of repetitive decoding of the same data with similar codes. Efficiency gains may thus be made if the number of duplicate decodings can be reduced.

2 Wackrow and Titchener’s Implementation

The above algorithm was first implemented by Wackrow under Titchener’s supervision as a C program called tcalc.c [15]. It exploits the fact that when the string is decoded over $S_{(k_1,k_2,...,k_{n+1})}^{(p_1,p_2,...,p_{n+1})}$, the only codeword boundaries that are removed from the decoding over $S_{(k_1,k_2,...,k_{n+1})}^{(p_1,p_2,...,p_{n+1})}$ are those that follow the T-prefix $p_{n+1}$. Even then, they are only removed if they do not follow a run of $k_{n+1} + 1$ copies of $p_{n+1}$.

Wackrow’s algorithm maintains a list of codeword boundary positions. During the decoding, the algorithm compares each codeword with the T-prefix $p_{n+1}$. This comparison is initially done by length only – if the codeword’s length does not equal $|p_{n+1}|$, no further comparison is undertaken. If the length of the codeword matches $|p_{n+1}|$, the codeword is compared to $p_{n+1}$ on a symbol-by-symbol basis. Furthermore, only codewords that precede the substring $p_{n+1}^{k_{n+1}} p_n^{k_n} \ldots p_1^{k_1} y$ at the end of $xy$ are decoded in each pass – the substring is implicit and yields no new information.

Note that in each decoding pass, each of the codewords preceding $p_{n+1}^{k_{n+1}} p_n^{k_n} \ldots p_1^{k_1} y$ is checked. For most long strings, the number of codewords that need to be checked in each pass thus decreases only slowly.

3 Our implementation

The number of the codewords that need to be compared with the respective T-prefix in each pass is a key factor that affects the efficiency of the algorithm.
In our algorithm, we also maintain a list of codeword boundary positions as a doubly linked list, which we call the *string list* as it lists all codewords in the sequence in which they appear in the string. Each item in this linked list records the starting position of a codeword in the string, its length, and points at both the preceding and the following codeword in the string. In this respect, the basic strategy of our algorithm is similar to that of Wackrow and Titchener. However, their implementation stores the boundary positions in an array rather than a linked list.

Apart from the string list, we also maintain an additional group of doubly linked lists, which we call *length lists*. Each item from the aforementioned doubly linked list also features two pointers that additionally embed the item into one of these length lists. Each of these length lists contains the linked list items corresponding to codewords of a common length. That is, the length list for \( L \) links all items that correspond to codewords of length \( L \). We thus have two means of navigating through the string: We can either proceed from one codeword to the adjacent next or previous codeword, or we can jump from that codeword to the next (or previous) codeword with the same length.

The advantage of using the length lists is that we need not compare each codeword in the string with the T-prefix \( p_{n+1} \). Instead, we only look at those codewords that have the same length as \( p_{n+1} \). These are available in the length list for \( L = |p_{n+1}| \).

The length lists are dynamically created on demand – if our string never contains any codewords of length \( L \), then no length list is created for \( L \).

The decoding algorithm for the T-decomposition then works as follows: In the first decoding pass, we decode the string over \( S \). Each symbol in the string is a codeword and the length of each codeword in this pass is 1. We thus create a length list for length \( L = 1 \) and add items for all these codewords to it. At the same time, we create the string list. We then proceed as follows:

1. Before each subsequent decoding pass, we identify the respective T-prefix and T-expansion parameter by following the string list backwards, one codeword from its end. This identifies the T-prefix. Subsequent probing steps in the list compare the respective next codeword to the left with the T-prefix. If they match, the T-expansion parameter is incremented, if not, the last copy of the T-prefix to the left becomes the new end of the string and the T-expansion parameter's value becomes final. Once the leftmost T-prefix copy coincides with the start of the string, we are finished. If this is not the case, we need at least one more decoding pass. In this pass, codewords with different length may appear as the boundaries after copies of T-prefixes are removed.

2. We identify the length of the T-prefix and select the appropriate length list. If it does not exist, then there are no codeword boundaries to remove and we can proceed with the next T-augmentation level. If it does exist, we follow the length list from its first element to its end, checking each element as to whether the codeword it represents matches the current T-prefix. If it does not match, we proceed with the next entry in the length list.

3. If a match is detected, then we must merge (concatenate) the current codeword with the subsequent codeword in the string list. For this purpose, we retain the
item corresponding to the current codeword and update its length to reflect the merger. Furthermore, we have to:

- Remove the current and the subsequent codeword from their respective length lists, as the length of the merged codeword is different.
- Remove the subsequent codeword from the string list, as it ceases to exist as a separate codeword.
- If the removed codeword also matches the T-prefix, we must allow for the merger with the codeword following it in the string (up to $k_{n+1}$ merge operations are possible).
- The resulting codeword must be inserted into the string list and into its appropriate length list. This length list is created if required. If the length list already exists, a pointer to the last codeword inserted into it aids the fast location of the insert point.

The operation then resumes by checking the next codeword in the length list for $|p_{n+1}|$ until the end of the list is reached. After this, the algorithm selects the next T-prefix and T-expansion parameter and returns to step 1.

4 Comparison

The algorithm described in the last section was implemented as a C program tlist.c [16] based on the user interface of tcalc.c and was tested in comparison with tcalc.c. The testing was performed using ASCII files with characters 0 and 1 and a calibrated average T-entropy [12]. Perhaps the most reassuring outcome of this testing is the confirmation that tcalc seems to work correctly for long strings. Both programs obtain identical results for long strings and we thus believe that both of them work correctly. Our initial tests also show that tlist executes several times faster than tcalc for all strings in the test. The following tables show an initial comparison of execution times for various strings and string lengths, obtained using the UNIX time command on a Redhat 8.0 Linux PC.

The first table shows a comparison of the T-decomposition of 2 million symbol strings with different average T-entropy values.

The second table shows the execution times for maximum T-entropy (“random”) strings of various lengths. It should be noted here that all of these strings were generated as the n-bit suffixes of the same 2000000 bit string. Strings of lengths below 400000 bits parsed to fast to produce meaningful times.

The data – while still based on a few measurements only – shows tlist outperforming tcalc in all cases. The highest gains are obtained for strings with medium T-entropy, but the gains are still significant even for random-looking strings. It also seems as if the behaviour of tlist under string extension is slightly more benign than that of tcalc, i.e., tlist scales slightly better.

However, some of the gains of tlist disappear when strings with larger alphabets are used. In this case, the codewords tend to stay short for longer as there are now many more combinations for short codewords available. In this case, the cost of the larger data
<table>
<thead>
<tr>
<th>Filename</th>
<th>tlist [s]</th>
<th>tcalc [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>lgst3.573550</td>
<td>3.5</td>
<td>7.6</td>
</tr>
<tr>
<td>lgst3.586787</td>
<td>4.5</td>
<td>25</td>
</tr>
<tr>
<td>lgst3.611055</td>
<td>6.7</td>
<td>48</td>
</tr>
<tr>
<td>lgst3.651050</td>
<td>11</td>
<td>76</td>
</tr>
<tr>
<td>lgst3.687660</td>
<td>9.8</td>
<td>118</td>
</tr>
<tr>
<td>lgst3.766200</td>
<td>14</td>
<td>151</td>
</tr>
<tr>
<td>lgst3.907580</td>
<td>19.5</td>
<td>200</td>
</tr>
<tr>
<td>lgst3.925405</td>
<td>43.7</td>
<td>213</td>
</tr>
<tr>
<td>lgst3.971029</td>
<td>33</td>
<td>230</td>
</tr>
<tr>
<td>lgst4.000000</td>
<td>45</td>
<td>239</td>
</tr>
</tbody>
</table>

Table 1: Execution time comparison between tcalc and tlist

<table>
<thead>
<tr>
<th>Length in ASCII bytes (0 or 1)</th>
<th>tlist [s]</th>
<th>tcalc [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>400000</td>
<td>3.0</td>
<td>10.8</td>
</tr>
<tr>
<td>500000</td>
<td>4.5</td>
<td>17</td>
</tr>
<tr>
<td>600000</td>
<td>6.0</td>
<td>23.5</td>
</tr>
<tr>
<td>700000</td>
<td>7.6</td>
<td>32</td>
</tr>
<tr>
<td>800000</td>
<td>9.5</td>
<td>41</td>
</tr>
<tr>
<td>900000</td>
<td>12</td>
<td>51</td>
</tr>
<tr>
<td>100000</td>
<td>14</td>
<td>63</td>
</tr>
<tr>
<td>110000</td>
<td>16</td>
<td>76</td>
</tr>
<tr>
<td>120000</td>
<td>19</td>
<td>89</td>
</tr>
<tr>
<td>130000</td>
<td>21</td>
<td>104</td>
</tr>
<tr>
<td>140000</td>
<td>25</td>
<td>121</td>
</tr>
<tr>
<td>150000</td>
<td>27</td>
<td>138</td>
</tr>
<tr>
<td>160000</td>
<td>31</td>
<td>156</td>
</tr>
<tr>
<td>170000</td>
<td>34</td>
<td>175</td>
</tr>
<tr>
<td>180000</td>
<td>38</td>
<td>195</td>
</tr>
<tr>
<td>190000</td>
<td>41</td>
<td>217</td>
</tr>
<tr>
<td>200000</td>
<td>45</td>
<td>239</td>
</tr>
</tbody>
</table>

Table 2: Execution time by string length for tcalc and tlist
structure overhead of \texttt{tlist} outweighs the small benefit that one can derive from the length lists. A test with a pseudo-random binary file (alphabet size 256) confirmed this.

5 \textbf{Conclusion}

The new T-decomposition algorithm presented in this paper appears to confirm the correctness of the implementation by Wackrow and Titchener. While the basic strategy was derived from that of Wackrow and Titchener and the user interface routines are to a large extent borrowed from \texttt{tcalc}, the core T-decomposition routines required a completely independent implementation.

As predicted by the design, the tests carried out to date show that the performance of the T-decomposition algorithm presented here is better than that of the existing implementation. The presently available data suggests that this improvement should permit the analysis of much longer strings than at present.

It is conceivable that the algorithm may be further improved. One possibility for improvement may lie in the optimization of the way it compares codewords with the respective T-prefix. E.g., it would be conceivable to split the length lists further into lists whose members feature, e.g., a particular suffix or prefix.

We would like to thank Mark Titchener for his constructive comments and for suggesting the use of his calibrated files for the comparative testing. \texttt{tcalc} and \texttt{tlist} are available under the GNU GPL.

\textbf{References}


[16] J. Yang and U. Guenther: tlist.c, written in C, available on request from the authors, under the GNU GPL