Fast Low Memory T-Transform

String Complexity in Linear Time and Space
with Applications to Android App Store Security

by

Niko Rebenich
B.Eng, University of Victoria, 2007

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF APPLIED SCIENCE

in the Department of Electrical and Computer Engineering

Fast Low Memory T-Transform

String Complexity in Linear Time and Space
with Applications to Android App Store Security

by

Niko Rebenich
B.Eng, University of Victoria, 2007

Supervisory Committee

Dr. Stephen W. Neville, Supervisor
(Department of Electrical and Computer Engineering, University of Victoria)

Dr. T. Aaron Gulliver, Departmental Member
(Department of Electrical and Computer Engineering, University of Victoria)

Supervisory Committee

Dr. Stephen W. Neville, Supervisor
(Department of Electrical and Computer Engineering, University of Victoria)

Dr. T. Aaron Gulliver, Departmental Member
(Department of Electrical and Computer Engineering, University of Victoria)

ABSTRACT

This thesis presents *flott*, the *Fast Low Memory T-Transform*, the currently fastest and most memory efficient linear time and space algorithm available to compute the string complexity measure *T-complexity*. The *flott* algorithm uses 64.3% less memory and in our experiments runs asymptotically 20% faster than its predecessor. A full *C*-implementation is provided and published under the *Apache Licence 2.0*. From the *flott* algorithm two deterministic information measures are derived and applied to Android app store security. The derived measures are the *normalized T-complexity distance* and the *instantaneous T-complexity rate* which are used to detect, locate, and visualize unusual information changes in Android applications. The information measures introduced present a novel, scalable approach to assist with the detection of malware in app stores.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

First of all, I want to thank Mark Titchener for discovering T-codes. Without him this thesis never would have been written. Further, I would like to thank Ulrich Speidel, who I was fortunate enough to meet during his sabbatical at the University of Victoria. He patiently answered my questions about T-codes and inspired me to develop the T-transform algorithm presented in this thesis.

Moreover, I would like to express my gratitude to my supervisor Stephen Neville. It has been an honour to be one of his students. Besides his much appreciated time and feedback, Stephen gave me the financial and academic freedom that was necessary to make this thesis possible.

I am especially grateful to Aaron Gulliver. Aaron, thanks for always having an open door for me. I really enjoyed throwing back and forth ideas with you. Thank you for taking the time to help me make sense of my plots and letting me take over your whiteboard.

Thanks to Ian, Amber, Eamon, Caroline, and Thomas for being the good friends they are.

Lastly, I would like to thank my family away from home – Penny, Doug and Moe – thanks for being so wonderful. And most of all I would like to thank my parents Gisela and Peter, along with my brothers Jan and Till, for their continued love and encouragement. Mami, Papi, Jan, and Till thank you for always being there for me and supporting me in all my pursuits.

Those who say it can't be done are usually interrupted by others doing it.

James A. Baldwin

DEDICATION

For Mami and Papi.

Chapter 1

Introduction

With the creation of the Internet, the way we create, distribute, and access information has fundamentally changed. Practically anyone can obtain, modify, and publish arbitrary data from nearly anywhere in the world. The amount of new information on networks and computing devices keeps growing and creates new challenges for data mining researchers that try to analyze these data.

Moreover, the Internet has become the primary gateway through which users obtain software for their computing devices. With the explosion of available software it becomes harder and harder to protect users from criminals that try to deploy malicious software, herein referred to as *malware* or *malcode*, on user devices. A user device compromised by malware may leak private user information or may be hijacked to serve as a node in large on-line crime networks also referred to as *botnets*. Botnets may be used for financial gain, e.g. the distribution of spam, or denial of service attacks among other possible unlawful activities [1].

The recent introduction of mobile computing devices such as smartphones and tablets has complicated matters even more since cell-phone networks now have essentially become a part of the Internet. The big players in the smartphone and tablet markets are Apple and Google with their *iOS* [3] and *Android* [25] mobile operating systems respectively. The software applications for these platforms are commonly referred to as *apps* and are distributed via app stores over the internet. Not long after their introduction, mobile devices became a target for malware writers [44]. Modern app stores can hold several hundreds of thousands of apps [4] as practically anyone can offer applications for download. This makes the detection of malware within app stores a non-trivial problem as at this scale a human adjudication process becomes untenable. A prominent example of human adjudication

failing within the Apple app store is given in [40], where a researcher was able to get malware approved for distribution over the Apple app store. Malware in app stores may be identified as such eventually, but we inevitably have to take the risk into account that significant damage for users and service providers have occurred until such an assessment has been made. For a cellular carrier the financial and image loss due a network disruption caused by malcode can be devastating.

The traditional defence mechanism employed by anti-malware vendors is to first identify a software as malicious by disassembly or monitoring the application's activity on execution. Subsequently, a static or dynamic signature for that particular application is constructed and stored in a database. The signature database is then propagated to user devices on a subscription basis. This traditional defence model is likely to be modified, in one way or another, to work within the mobile device marketplace. It is, however, rather easy for malware writers to evade signatures by making simple modifications to small portions of their code. For the malware writer these simple modifications are very convenient since the malicious code does not have to be rewritten in order not to trigger known signatures.

As shown by Cohen [14], there unfortunately is no automated process that can accurately decide whether a given app is malicious or not. However, an approach that is not solely relying on information lossy signatures is likely to better identify malware than the traditional signature based approach alone. In this thesis we propose a pragmatic approach towards this goal by using information measures to detect unusual changes of apps within app stores. In particular, in this thesis we introduce a *deterministic complexity measure* based approach that: a) allows us to determine to what degree an app has changed from one release to the next b) allows us to detect the location of new and re-used (malicious) code within an app and c) is fast enough to scale to large app stores.

In this thesis we develop the *Fast Low Memory T-Transform (flott)*, which currently is the fastest and most memory efficient linear time and space algorithm to compute the *T-complexity* [78, 80] of a string. A full *C*-implementation of the algorithm is provided in [60] and Appendix A which is freely available as open-source under the *Apache Licence 2.0* [76]. The algorithm can be used to calculate a global distance in information content between two strings, and in addition, enables us to locate the position of unusual information changes between related strings via the *instantaneous T-complexity rate*.

This thesis is structured as follows:

- In Chapter 2, the different meanings of the term "complexity" as used in computer science contexts is explained and the T-complexity and *Lempel-Ziv complexity* [48] are introduced as examples of deterministic complexity measures.
- In Chapter 3, the T-transform algorithm is explained in detail and a full pseudo-code implementation of the algorithm is provided along with worked examples that illustrate how the algorithm achieves its linear runtime and memory usage.
- In Chapter 4, the performance of the *flott* algorithm is evaluated against its predecessor implementation and *flott's* memory requirements are compared with those of suffix trees that can be used to efficiently compute the *Lempel-Ziv complexity* of a string.
- In Chapter 5, the normalized T-complexity distance and instantaneous T-complexity rate are defined as global and local information measures which are subsequently used in a case study about Android applications.
- In Chapter 6, we conclude this thesis and suggest areas for future work.

Chapter 2

Complexity

Complexity is a term used quite frequently in the field of computer science, and its meaning is largely dependent on the context in which it is used. In this thesis we distinguish between the notions of *computational complexity* in time and space, *algorithmic complexity*, and *deterministic complexity* which we will discuss individually in the subsequent sections.

2.1 Computational Complexity

The computational complexity of an algorithm measures how efficiently the algorithm uses the available computing resources. In particular, we are interested in analyzing how much overall time and memory space is required by the algorithm to perform a task. Time and space complexity are usually a function of the length of the input data n. In the following let $f(n)$ be an exact time or memory requirement obtained for an algorithm implementation running on a particular computing hardware. We then say the algorithm has time (or space) complexity of the *order* g if there exist two positive constants c_1 and c_2 such that $f(n) \leqslant c_1\, g(n) + c_2$ for all allowed values of n. We write $O(g)$ and also refer to it as the *big* O notation characterizing the *asymptotic* time or space complexity. Essentially, this allows us to compare algorithms in terms of their relative overall performance irrespective of the particularities of the underlying hardware [74].

The function $g(n)$ is affected by the chosen model of computation. Hence, it is important to state what computing model was used when the computational complexity of an algorithm is evaluated. One of the simplest models of computa-

tion is the *Turing machine* model [33]. It is predominately used in a purely theoretical context and is less practical when examining an algorithm's time and space behaviour on modern day computers [74]. Therefore, throughout this thesis we adopt the *random-access machine* model with *uniform cost measure* [2, 16]. In this model we assume a finite program, a finite number of registers, and a finite $O(n)$ amount of uniquely addressable *words* or memory cells. In practice we differentiate between *uniform cost measure* and *logarithmic cost measure* when evaluating the computational complexity of algorithms on the *random-access machine*. The chosen *uniform cost measure* approach assumes that all elementary instructions such as arithmetic integer operations, integer comparisons, and read and write instructions on integers take constant, $O(1)$, time. Further, we assume that integers are stored in fixed sized words with a *word-size* ω logarithmic in n. In general, unless stated otherwise, we assume a word-size ω of 32 *bits* or 4 *bytes* as this has been established as a convenient choice. We note that in practice a *uniform cost measure* ultimately puts a cap on the maximum input length that a program can process. The maximum input length is then bounded by the chosen word-size. In order to allow for arbitrary sized inputs a *logarithmic cost measure* should be used. The *logarithmic cost measure* accounts for a cost in time and space proportional to the number of bits required to represent the integer that is subject to an elementary instruction [74]. Thus, an algorithm with order $O(n)$ time (or space) complexity under the *uniform cost measure* is of order $O(n \log n)$ time (or space) complexity under the *logarithmic cost measure*. The *uniform cost measure* was adopted in this thesis for straightforward comparison with the *big O* notation used in cited *suffix tree* literature.

Finally, when comparing algorithm performance, we compare the *worst case* time and space requirements unless explicitly stated otherwise.

2.2 Algorithmic Complexity

Algorithmic complexity has its roots in information theory and was pioneered by Andrei Nikolaevich Kolmogorov who proposed it as a measure of the information content of the individual string [43]. He defined the algorithmic complexity $K(x)$ of a string x as the size of the smallest possible algorithm which can execute on the universal turing machine and is able to reproduce just that string and halt [51, 17]. For this reason algorithmic complexity is also often referred to as *Kolmogorov com-*

plexity; however, Solomonoff [66] and Chaitin [9] have to be credited for the notion of algorithmic complexity as well, as both independently published similar works to that of Kolmogorov that arrived at essentially the same conclusions [50]. Determining the shortest program $K(x)$ is a known uncomputable problem [50, 17], and in terms of computational complexity the shortest program is by no means required to have the shortest space and/or time complexities. We may not be able to compute Kolmogorov complexity, however, in the next section we introduce "deterministic complexity", a complexity measure which may be viewed as a computable cousin of Kolmogorov complexity.

2.3 Deterministic Complexity

Deterministic complexity measures strive to measure the randomness of the individual string using a *deterministic finite automaton* (*DFA*). We define deterministic complexity as the algorithmic effort required by a string parsing algorithm to transform a string into a set of unique patterns [84]. The effort may be measured either as a function of the total number of parsing steps required by a *DFA* or as a function of the compressed string length in bits under an optimal pattern encoding scheme. It is tempting to view deterministic complexity as a computable "estimate" to Kolmogorov complexity, however, we herein strictly refrain from doing so, as the quality of such an estimate is not assessable.

There have been numerous approaches to use deterministic complexity measures for data mining purposes, most relevant for this thesis are the works by Vitányi, Li, Cilibrasi et al. [49, 11, 12, 10] in which the authors propose a Kolmogorov complexity based similarity metric, the *Normalized Information Distance (NID)*, and its deterministic cousin called the *Normalized Compression Distance (NCD)*. The *NCD* employs size-compacting industry standard string compressors such as the Lempel and Ziv factorization based compressor *LZ77* (*gzip*), a Burrows-Wheeler Transform based data compressor (*bzip2*), and more recently, prediction by partial matching (*PPM*) and Lempel Ziv Markov chain (*LZMA*) algorithms [12, 8, 26]. Conceptually, the *NCD* is computed as a ratio from two individually and jointly compressed strings. The idea is that, the more information the two strings share in common the smaller the size of their joint compression and, therefore, the lower their normalized distance and the closer their "relatedness". In order to properly relate every information pattern to every possible other pattern

inside a string conglomerate an optimal algorithm has to touch each string symbol at least once. Thus, a sequentially implemented solution for a deterministic complexity measure must have a lower bounded time complexity of $O(n)$.

The off-the-shelf compressors used in *NCD*, in one way or another, first transform information into a less redundant meta-representations which then are compacted in size using for example Huffman or arithmetic coding. However, a deterministic measure estimating the information content of a string does not necessarily require a size-compacting encoding phase as it is possible to obtain a complexity measure from the algorithmic effort required to construct the meta-representation alone. The *Lempel-Ziv complexity* [48], herein referred to as *LZ76*, and *T-complexity* [78, 80, 85], subject of this thesis, are two examples of such deterministic complexity measures. *LZ76*, is performing an exhaustive pattern search over its entire input which makes it well suited to estimate the complexity of sources with long memory. An *NID* and *LZ76* based approach was used in [58] for the construction of phylogenetic trees. However, a naïve *LZ76* implementation does not scale very well for large inputs because of its quadratic, $O(n^2)$, runtime. This is likely the reason why less resource demanding size-compacting compressors such as the *LZ77* compressor, *gzip*, and the block compressor *bzip2* are employed for the computation of the *NCD* in [12]. Both compressors have a runtime of the order $O(n \times m)$, where m is the window and block size used in *gzip* and *bzip2* respectively. Both compressors compress ergodic sources well [64]. However, ergodicity is an assumption that generally does not hold if the input is the concatenation of two long and related strings. A window or block based compressor is then producing correct results only if the joint length of the two strings fits within the block or window size of the compressor or if all repeated patterns fall within the chosen blocks or windows [8, 88]. Once the joined string exceeds the compressor's block or window, the compressor may not be able to relate shared information across frame boundaries as the compressor is literally throwing information away. Unfortunately, as soon as we require m to be of the same length as the joined strings all runtime advantages are lost, and we default back to an effective overall runtime of order $O(n^2)$. Similarly, the Markov model based compressors *PPM* and *LZMA* produce accurate results only if their dictionary size is left unrestricted [8, 26]. The accuracy of *PPM* and *LZMA* is paid for with poor runtime performance and, dependent on their respective Markov model implementations, exponential, $2^{O(n)}$, demands in memory space [13].

2.3.1 Lempel–Ziv Complexity

Lempel and Ziv *(LZ)* were among the first to assess the complexity of finite strings in terms of the number of "self-delimiting production" steps needed to reproduce a string x from a set of distinct string patterns [48]. Lempel and Ziv published a series of papers on interrelated string factoring algorithms which are listed in Table 2.1. A detailed explanation of the family of *LZ* algorithms is beyond the scope of this thesis, and the interested reader is referred to [63, 64] for a comprehensive overview.

In their first paper [48], published in 1976, the authors introduce a string production algorithm, herein referred to as *LZ76*, with time complexity of $O(n^2)$. In this early paper the authors seemed to have focused their attention primarily on the derivation of a deterministic complexity measure rather than its efficient implementation. Essentially, *LZ76* decomposes a string x in an exhaustive production history that allows the back referencing to any string position before the current parsing position. The number of *LZ76* production steps needed to reproduce the string x is then said to be the *LZ-complexity* of x.

The two subsequent papers of Lempel and Ziv were less concerned about developing a deterministic string complexity measure but focused on the design of general purpose lossless data compression algorithms now commonly referred to as *LZ77* and *LZ78*. Both algorithms were not intended to serve as string complexity measures per se, however, the count of parsing steps needed to compress a string via either algorithm may used as an upper bound on the *LZ-complexity* of a string. Moreover, the compressed string length in bits may also be used as a deterministic complexity measure.

The *LZ77* algorithm [97], published in 1977, addressed the runtime performance issues of *LZ76* parsing by introducing a fixed size, $O(m)$, sliding window that restricts back referencing to any position within the window, resulting in an $O(m \times n)$ runtime complexity [97]. *LZ77* quickly became popular and was used in numerous, often commercial, compression schemes [63]. *LZ77* owes its popularity mainly to it's simplicity, speed, and constant memory requirements.

Finally *LZ78* was published in 1978 [98]. The *LZ78* algorithm uses an unrestricted size dictionary as part of its parsing algorithm which stores previously encountered string patterns. When the parsing begins, the dictionary is empty. From the current parsing position symbols are read and concatenated into a new

pattern. Reading continues until there is no match for the current pattern in the dictionary, and a new dictionary entry is formed with its last symbol being new. In contrast to *LZ76* which allows back referencing to any position in the string before the current parsing position, the dictionary method used in *LZ78* restricts the patterns in the dictionary to previously encountered parsing offsets in the string. Interestingly enough, *LZ78* was actually developed before *LZ77*, but to address the large memory requirements of an unrestricted dictionary, *LZ77* was published before *LZ78*.

		Complexity	
Algorithm	**Description**	**time**	**space**
LZ76 [48]	Straightforward implementation of the Lempel-Ziv factorization algorithm.	$O(n^2)$	$O(n)$
LZ77 [97]	Similar to *LZ76* but using a fixed sized sliding window.	$O(n \times m)$	$O(m)$
LZ78 [98]	Parsing algorithm with unrestricted dictionary size efficiently implemented using a *trie*.	$O(n)$	$O(n)$

Table 2.1: Computational complexity of Lempel-Ziv algorithm family.

The best-known version of *LZ78* is probably the Lempel-Ziv-Welch implementation (*LZW*) [91]. *LZW* is famous not only for its excellent performance in lossless image compression, but also for its patent [90] and the infringement lawsuits associated with the former. The aggressiveness with which the patent assignee, at the time Unisys Corporation, enforced their patents – but also the relatively high memory demands of the vanilla *LZ78* algorithm – resulted in a myriad of *LZ78* derivatives, often only differing in the way they implement or restrict their dictionaries [64]. *LZ78* based algorithms are among the most heavily patented algorithms, and until late 2004 when the last of Unisys' patents expired, the use of *LZW* was not without problems for many commercial applications. Today, derivative *LZ78* patents still have an impact on the use of some specific implementations.

The *LZ76* string parsing algorithm may be implemented in linear time and

space using a *suffix tree*. However, the construction of suffix trees in linear time and space is a non-trivial undertaking [56], and suffix trees have a reputation for high memory demands which presents a limiting factor for some applications operating on big data sets [45]. In part advances in computing and the availability of large and inexpensive memory have mitigated these issues. However, we are not aware of an efficient and at the same time "open" implementations of *LZ76* that can process large inputs. Most critical to the *LZ76* algorithm is an efficient implementation of back referencing to string patterns. Rodeh et al. [61] suggest a linear time and space *LZ76* implementation based on McCreight's linear time suffix tree construction algorithm provided in [55]. McCreight provides two approaches to linear time suffix tree construction. The most space efficient one possesses an alphabet size, $\|\mathbf{S}\|$, dependent runtime, $O(\|\mathbf{S}\| \times n)$, which makes the algorithm less practical for large alphabets. Realizing this McCreight suggests a modified algorithm making use of hashing to achieve a linear, alphabet independent runtime [55]. However, hashing based approaches come at the expense of additional memory usage and yield good performance only if hash collisions can be avoided which becomes hard for large inputs.

2.3.2 T-Complexity

T-complexity is a deterministic string complexity measure that is, just like *LZ76*, the result of a string factorization algorithm. For the purpose of this thesis this string factorization algorithm will be referred to as *T-transform*. In previous literature the term *T-decomposition* [85] was also used to described the T-transform process and both terms are used interchangeably in this thesis. The T-transform generates a set of coefficients (*copy factors*) by recursively filtering the information in a string with a set of string patterns also referred to as *copy patterns*.

Conceptually, the computation of T-complexity is similar to the computation of *LZ*-complexity in that both measures evaluate the effort to factor a string into a less redundant meta-representation of basic string patterns. However, the parsing mechanisms used in *LZ76* and the T-transform are quite different from one another. Further, the T-transform's notion of copy factors does not exist for the family of *LZ* factorization algorithms. Moreover, the T-transform is an *off-line* algorithm parsing strings from back to front. This means that the T-transform algorithm can only operate on finite strings. In contrast, the *LZ*-complexity of a string

may also be computed via a single pass implementation based on Ukkonen's *on-line* suffix tree construction algorithm [87, 29]. Since 2008 an on-line, suffix tree based, forward parsing algorithm developed by Hamano and Yamamoto is available for the computation of a T-complexity measure [32].

In this thesis we present the currently most efficient T-transform algorithm to compute the T-complexity of a string in linear time and space. T-complexity is presented as an efficient alternative to the *LZ*-complexity measure. With this thesis we provide an open-source implementation of the T-complexity measure as a viable alternative to *LZ*-complexity implementations, where the specific target application domain of the T-complexity is large-scale data sets.

2.4 Summary

This chapter has explained the meaning of the term "complexity" in the context of computational performance and information theory. In the next chapter we will focus our attention on practical implementations of the T-transform. Along with the several T-transform algorithms presented we provide the necessary background to understand the various T-transform paradigms.

Chapter 3

T-Transform Paradigms

T-complexity, and the T-transform algorithm for that matter, have their origin in coding theory, more precisely, they are a by-product of the construction process of *T-codes*. T-codes were proposed by Mark Titchener in 1984 as prefix-free variable length codes [77]. One and a half decades after Titchener's initial publication, T-codes have found various information theoretic applications ranging from string complexity measures [83, 85], similarity distances [95, 92], computer security applications [69, 20, 70], to the analysis of time series data [82].

Before we introduce the T-transform prototype we will provide the reader with the necessary background to understand the basic idea behind T-codes and their construction. We continue by introducing the complexity measure T-complexity and then discuss in detail the means by which the T-transform is implemented in linear time and space.

3.1 Basic Notation and Conventions

For consistency with prior literature, the notation presented here borrows largely from the works of Titchener, Speidel[1], Yang, and Eimann [84, 86, 27, 67, 92, 20]. At several occasions in this thesis, we provide *pseudo-code* for a hardware independent description of algorithms. For reader convenience, Table 3.1 provides a detailed reference of the operands and symbols used, and may be consulted when reading pseudo-code.

[1] nee Guenther (Günther)

3.1.1 Set Notation

Let $\mathbf{X}$ and $\mathbf{Y}$ denote two sets. Then the cardinality of $\mathbf{X}$, or the number of elements contained in $\mathbf{X}$, shall be defined as $\|\mathbf{X}\|$ and the cardinality of $\mathbf{Y}$ as $\|\mathbf{Y}\|$ respectively. Further, set subtraction is indicated by "$\setminus$" also commonly referred to as "backslash". Thus, the set $\mathbf{Z}$ defined as follows $\mathbf{Z} = \mathbf{X} \setminus \mathbf{Y}$ contains the elements of set $\mathbf{X}$ with the exclusion of any element contained in set $\mathbf{Y}$. $\mathbf{Z}$ may contain all or a partial number of elements from $\mathbf{X}$ or be the empty set $\varnothing$. $\mathbf{Z}$ is said to be a subset of $\mathbf{X}$ which is indicated as follows: $\mathbf{Z} \subseteq \mathbf{X}$. When $\mathbf{X}$ and $\mathbf{Y}$ share common elements their intersection, denoted by the $\cap$-symbol, is not empty, this fact is expressed as $\mathbf{X} \cap \mathbf{Y} \neq \varnothing$. Naturally, if the sets $\mathbf{X}$ and $\mathbf{Y}$ share common elements, then $\mathbf{Z}$ as defined above cannot contain the entirety of elements in $\mathbf{X}$; in this case $\mathbf{Z}$ is called a *proper subset* of $\mathbf{X}$ and is denoted by $\mathbf{Z} \subset \mathbf{X}$. Conversely, $\mathbf{X}$ is said to be a *superset* of $\mathbf{Z}$. The union of two sets is denoted by the symbol "$\cup$". If $\mathbf{W}$ is the union of the sets $\mathbf{X}$ and $\mathbf{Y}$ ($\mathbf{W} = \mathbf{X} \cup \mathbf{Y}$) then $\mathbf{W}$ combines the unique elements contained in both of the sets $\mathbf{X}$ and $\mathbf{Y}$. Finally, $\mathbb{N}$ denotes the set of all integers and $\mathbb{N}^+$ denotes the set of positive integers. Similarly $\mathbb{R}$ and $\mathbb{R}^+$ denote all real valued and all positive real valued numbers respectively.

3.1.2 Source Alphabets and Strings

Generally, in this thesis, the underlying assumption is that both string length and alphabet size are finite as otherwise the calculation of information measures on a random-access machine is not practical due to real world time and memory constraints. Expressed in set theory terminology one may see the alphabet as the set of characters from which strings are generated through concatenation by a information source. Traditionally, the *alphabet set* is denoted as $\mathbf{S} = \{a_1, a_2, a_3, \ldots, a_{m-1}, a_m\}$ and the individual characters, which are also referred to as *symbols*, are denoted by a_i where $1 \leqslant i \leqslant m = \|\mathbf{S}\|$.

The set of all finite strings over $\mathbf{S}$ is indicated by $\mathbf{S}^*$ and contains all possible concatenations of symbols from the set $\mathbf{S}$. Moreover, the alphabet itself is a subset of $\mathbf{S}^*$ if we consider $\mathbf{S}$ as the subset of all strings with size one.

Let x denote a string contained in $\mathbf{S}^*$ then the definition of x is given as $x = x_1 x_2 x_3 \ldots x_n$ where x_i are symbols in $\mathbf{S}$. The length of individual strings is denoted by the operator $|\cdot|$, and therefore, we have $|x| = n$. In analogy to the notion of the empty set, we define the empty string λ as a string with no symbols. The set

Operand / Symbol	Description
$\longleftarrow$	assignment (by value for primitive data types; by reference for composite data types)
@e	denotes the memory offset (distance) of the array element e away form the array's base address
$\lvert \cdot \rvert$	number of elements (fields) in a linear array (or composite data type)
$\lVert \cdot \rVert$	number of elements in a list (abstract data type describing an ordered set)
$[\,\cdot\,]$	array element located at the specified memory offset away form a base address
$[\,j,\, j{+}k{-}1\,]$	set of the k consecutive array elements located from memory offset of element j onwards
$\langle\,\cdot\,\rangle$	list element at an specified position in an the ordered set (list)
$\langle\,j,\, j{+}k{-}1\,\rangle$	set of the k consecutive list elements from position j onwards
r_{type}	type cast memory location r to the data type specified in the subscript
$.name$	data field at by "$name$" identified memory offset into a composite data type
:	operation executed on abstract data type(s)
`name` $(\,\cdot\,)$	function/method call

Table 3.1: Operand and symbol notation used in pseudo-code.

of non-empty strings is then defined as as $\mathbf{S}^{+}{=}\mathbf{S}^{*}\backslash\{\lambda\}$. Furthermore, if $x, y \in \mathbf{S}^{+}$ are two distinct strings then the length of the concatenation of x and y, denoted as xy, is $|xy| = |yx| = |x|+|y|$. Note however, that string concatenation is not commutative, that is, $xy \neq yx$. The shorthand notation to indicate the concatenation of k copies of the same string x is given by x^k, where k may assume any non-negative integer value including the special case $k = 0$ defined as the empty string $x^0 = \lambda$. Finally, we denote the i^{th} character in the string x either by x_i or as per common array index notation used in programming as $x[\,i\,]$. By convention we choose to index the elements (characters) in an array (string) from *left-to-right* with the first element or character identified with the numeral 1. That is, $x[\,1\,]$ refers to the first character in the string x. Similarly, we denote the substring starting at position i of length k by $x[\,i,\, i{+}k{-}1\,]$.

3.2 T-Augmentation

T-augmentation is the principal set operation used in the construction of T-codes. The T-augmentation operation is not limited to T-code sets but is applicable to any set of code words. T-codes belong to the category of prefix-free codes. Since the most primitive prefix-free code is an alphabet by itself, we define the most primitive T-code as just the alphabet, i.e. the most primitive binary T-code is $\mathbf{S} = \{0, 1\}$.

The T-augmentation procedure is subject to two parameters, p and k denoted as *copy pattern* and *copy factor* respectively. More specifically, we identify one code word from the T-code $\mathbf{S}$ as the copy pattern p; thereafter, by incrementally concatenating up to k copies of p the set of *T-augmentation prefixes* $\mathbf{\Lambda}_k(p)$ is formed. $\mathbf{\Lambda}_k(p)$ includes the empty string λ here indicated by p^0. The cardinality of $\mathbf{\Lambda}_k(p)$ is $k + 1$. The resulting T-augmentation prefix set is given by,

$$\mathbf{\Lambda}_k(p) = \bigcup_{i=0}^{k} p^i = \{p^0, p^1, \ldots, p^k\}, \tag{3.1}$$

where $k \in \mathbb{N}^{+}$. Ranked by element length, every member of $\mathbf{\Lambda}_k(p)$ is a prefix to every other member of higher rank. T-augmentation in which the copy factor is not restricted from above, i.e. $k \geqslant 1$, is referred to as *generalized T-augmentation* and similarly, a T-code produced this way is referred to as a *generalized T-code*. Note,

that unless stated otherwise, we assume that the terms T-code and T-augmentation refer to their generalized versions.

We define the T-augmentation function which transforms the T-code $\mathbf{S}$ into the T-augmented T-code $\mathbf{S}_{(p)}^{(k)}$ as follows,

$$\mathbf{S}_{(p)}^{(k)} = \bigcup_{i=0}^{k} p^i \mathbf{S} \setminus \mathbf{\Lambda}_k(p) \ . \tag{3.2}$$

In Equation 3.2 we prefix each of the elements in $\mathbf{S}$ one-by-one with all the elements in $\mathbf{\Lambda}_k(p)$; thereafter, we remove the elements $\mathbf{\Lambda}_k(p)$ themselves which then yields the prefix-free T-code $\mathbf{S}_{(p)}^{(k)}$. Generalizing this idea, we may construct a T-code of arbitrary size and composure of code words, by first selectively T-augmenting an initial alphabet $\mathbf{S}$ and subsequently iteratively T-augmenting the resulting codes up to any desired level. We write this iteration as,

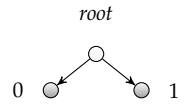
$$\mathbf{S}_{(p_1,p_2,\ldots,p_\alpha)}^{(k_1,k_2,\ldots,k_\alpha)} = \left[\ \cdots \ \left[\left[\mathbf{S}_{(p_1)}^{(k_1)} \right]_{(p_2)}^{(k_2)} \right] \ \cdots \ \right]_{(p_\alpha)}^{(k_\alpha)} \ , \tag{3.3}$$

and say that $\mathbf{S}_{(p_1,p_2,\ldots,p_\alpha)}^{(k_1,k_2,\ldots,k_\alpha)}$ denotes a T-code at *T-augmentation level* $\alpha \in \mathbb{N}^+$. We close this section by providing a short binary example demonstrating the construction of a simple T-code in two T-augmentation steps.

Example 3.2.1 (Binary T-Code). In the example presented here, consider the T-code $\mathbf{S}_{(1,0)}^{(3,1)}$ constructed from the binary alphabet $\mathbf{S} = \{0, 1\}$. The construction process of the T-code is illustrated using set notation and accompanied by figures showing their T-code trees at the individual T-augmentation levels $i = 0 \ldots 2$.
The base alphabet $\mathbf{S}$ from which $\mathbf{S}_{(1,0)}^{(3,1)}$ is constructed can be interpreted as a T-code at T-augmentation level zero. The binary T-code tree for $\mathbf{S}$ is depicted in Figure 3.1 (a), with the elements of $\mathbf{S}$ forming the leaf nodes of the tree. At T-augmentation level one the intermediate T-code set $\mathbf{S}_{(1)}^{(3)}$ and tree are given in Figure 3.1 (b). Note that, in set notation, the removal of the T-augmentation prefixes $\mathbf{\Lambda}_3(1)$ is illustrated by crossing those elements out. The second and last T-augmentation step yields the final T-code $\mathbf{S}_{(1,0)}^{(3,1)}$. Its code words and T-code tree are shown in Figure 3.1 (c).

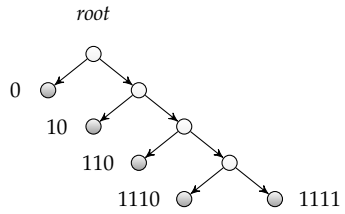
(a) level $i = 0$:

$$\mathbf{S} = \{0, 1\}$$



(b) level $i = 1$:

$$\mathbf{S}^{(3)}_{(1)} = \{ \quad 0, \quad \not{1}, \\ 10, \quad \not{1}\not{1}, \\ 110, \quad \not{1}\not{1}\not{1}, \\ 1110, \quad 1111 \quad \}$$



(c) level $i = 2$:

$$\mathbf{S}^{(3,1)}_{(1,0)} = \{ \quad \emptyset, \quad 00, \\ 10, \quad 010, \\ 110, \quad 0110, \\ 1110, \quad 01110, \\ 1111, \quad 01111 \quad \}$$

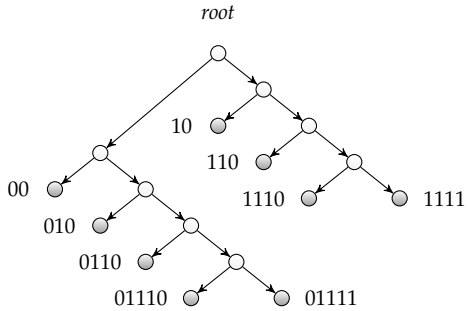


Figure 3.1: Example of the construction of the binary T-code $\mathbf{S}^{(3,1)}_{(1,0)}$ and its intermediate T-augmentation steps (a) – (c) .

3.3 T-Transform Prototype

In this section we will develop the prototype of the T-transform algorithm which allows us to construct a T-code from any arbitrary string. From the previous example we observe that $\mathbf{S}^{(k_1,\ldots,k_\alpha)}_{(p_1,\ldots,p_\alpha)} \subset \mathbf{S}^{(k_1,\ldots,k_{\alpha-1})*}_{(p_1,\ldots,p_{\alpha-1})} \subset \ldots \subset \mathbf{S}^*$, that is to say, a T-code at T-augmentation level α is a subset of all possible concatenations of the code words of the T-code at its previous T-augmentation level $\alpha - 1$ and so on. In general, the total number of the longest code words in $\mathbf{S}^{(k_1,\ldots,k_\alpha)}_{(p_1,\ldots,p_\alpha)}$ is equal to the cardinality of the underlying coding alphabet $\|\mathbf{S}\|$.

More specifically, the longest code words form the set

$$\mathbf{X} = \bigcup_{i=1}^{\|\mathbf{S}\|} xa_i, \tag{3.4}$$

with $a_i \in \mathbf{S}$, $1 \leqslant i \leqslant \|\,\mathbf{S}\,\|$ and the string x being the common prefix to all of the longest code words. For illustration consider once more Example 3.2.1. In the example the set $\mathbf{X}$ is given by,

$$\mathbf{X} = \bigcup_{i=1}^{2} xa_i \;=\; \{01110, 01111\}, \quad \text{with} \quad x = 0111 \;\text{ and }\; \mathbf{S} = \{0, 1\}\;.$$

Given one of its longest code words, it is possible to reconstruct the T-code by decomposing x into a combination of copy patterns and copy factors such that $xa_i \;=\; p_\alpha^{k_\alpha} p_{\alpha-1}^{k_{\alpha-1}} \cdots p_2^{k_2} p_1^{k_1} a_i$. Nicolescu et al. proved in [57] that the mapping between a T-code set and the set of its longest code words always exists and that it is unique. In other words, a special property of any T-code is that its construction process can be uniquely deduced from any of its longest code words. We denote the corresponding mapping function as the *T-transform*, $\kappa(x, \mathbf{S})$, given by,

$$\kappa : \mathbf{X} \;\leftrightarrow\; \mathbf{S}^{(k_1,\ldots,k_\alpha)}_{(p_1,\ldots,p_\alpha)}\,, \quad \text{where} \quad \mathbf{X} = \bigcup_{i=1}^{\|\mathbf{S}\|} xa_i, \quad x \in \mathbf{S}^*, \quad \text{and} \quad a_i \in \mathbf{S}\;. \tag{3.5}$$

The literal character a_i in the longest code words does not carry much significance other than being required to establish the prefix-freeness of a T-code. Thus, in subsequent treatment we often drop the subscript from the letter a while still letting it represent all possible choices. We may also omit the literal character altogether and assume it to be implicitly added to x.

3.3.1 Naïve T-Transform Algorithm

The T-transform function can be implemented as a recursive string decomposition algorithm. A naïve, iterative pseudo-code version is given in Figure 3.2. It accepts the input string x representing the prefix to any one of the longest code words $x \in \mathbf{S}^*$ and provides α-tuples for copy patterns $p = \{p_1, \ldots, p_\alpha\}$ and copy factors $k = \{k_1, \ldots, k_\alpha\}$ as output.

▶ **Algorithm** ▷ `T-Transform`

input : x `/* string` $x \in \mathbf{S}^*$ `*/`
output: α `/* number of t-augmentation steps */`
p `/* copy pattern` α`-tuple */`
k `/* copy factor` α`-tuple */`

initialization:
1 divide xa, $a \in \mathbf{S}$, into sequence single character tokens.
2 $i \longleftarrow 0$
3 **while** ***more than one token left*** **do**
4 $i \longleftarrow i + 1$
5 $p_i \longleftarrow$ *second-to-last* token.
6 $k_i \longleftarrow$ number of consecutive p_i to the left, counting p_i as the first copy.
7 scan tokens *left-to-right* and combine tokens into larger tokens of the form $p_i^{k'}q$ such that:
8 $(1 \leqslant k' \leqslant k_i \ \wedge \ q \neq p_i) \ \vee \ (k' = k_i \ \wedge \ q = p_i)$
9 **end**
10 $\alpha \longleftarrow i$

Figure 3.2: Pseudo-code listing of naïve T-transform algorithm.

The T-transform algorithm begins by splitting the string x in a list of single character substring patterns herein referred to as *tokens*. This initial state of the T-transform algorithm is also referred to as the T-transform at level $i = 0$. Next, the copy pattern of the first T-transform level p_1 is identified as the *second-to-last* token in the list. We then try to extend a chain of tokens identical to the copy pattern *to-the-left*. Counting the copy pattern as the first element, we establish the level $i = 1$ copy factor k_1 as the total number of elements in this chain. Now, moving to the beginning of the token list, we proceed by searching for tokens identical to the copy pattern *left-to-right*. Once such a token is found we merge at most k_1 consecutive copies of it with the immediately following token into a larger token $p_1^{k'}q$, $k' \leqslant k_1$. These composite tokens $p_i^{k'}q$ are referred to as *aggregate tokens* and $p_i^{k'}$ and q are called *aggregate prefix* and *aggregate suffix* respectively. We proceed with our search until we reach the end of the token list and repeat the overall process until at the end of a search no more tokens can be merged.

For illustration, Figure 3.3 shows the T-transform of some string at intermediate level i. The last element in the token list is $\hat{x}a$, and the prefix $\hat{x}$ to the literal character a is referred to as *T-handle*. With each T-transform level the length of the T-handle $|\hat{x}| = |p_i^{k_i}p_{i-1}^{k_{i-1}} \cdots p_1^{k_1}|$ grows until $\hat{x} = x$. The leftmost symbol in $\hat{x}$ marks the boundary between the mutually exclusive sets of all leftover aggregate tokens

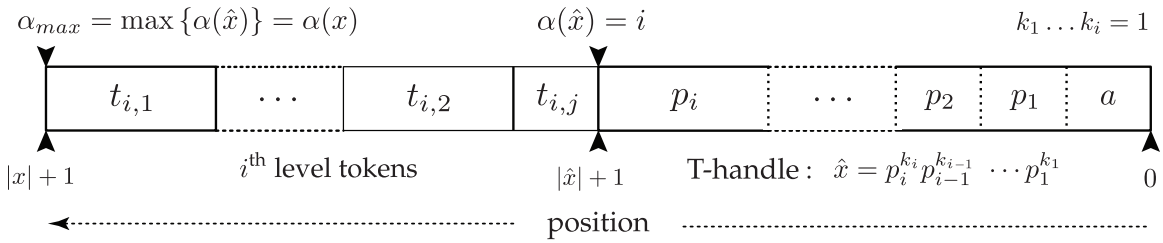


Figure 3.3: T-Transform at intermediate T-augmentation level i.

and the set of copy patterns at any given T-transform level $\alpha(\hat{x}) = i$. The cardinality of p and k grows with each additional decomposition level. The total number of levels, $\alpha_{max} = \alpha(x)$, of the T-transform is equal to the number of T-augmentation steps needed to construct a T-code in which x is the prefix to all its longest code words. As we will see shortly, the total number of T-transform levels, is related to how "T-complex" the information in x is.

To aid the better understanding of the T-transform algorithm, we will examine the worked example (Example 3.3.1) below.

Example 3.3.1 (T-Transform, Binary). In the following example let $\mathbf{S} = \{0, 1\}$ and let the binary string processed by the T-transform algorithm be defined as $x = 10110011011$. The decomposition of x starts by initializing the iteration counter i which counts the number of steps required to decompose x. Next, we add a terminating character $a \in \mathbf{S}$ to the string and divide it into single character tokens (the token boundaries are indicated by vertical lines) :

$$xa = \texttt{1} \mid \texttt{0} \mid \texttt{1} \mid \texttt{1} \mid \texttt{0} \mid \texttt{0} \mid \texttt{1} \mid \texttt{1} \mid \texttt{0} \mid \texttt{1} \mid \texttt{1} \mid a \; .$$

The first decomposition step determines the first item of the copy pattern α-tuple as the *second-to-last* token

$$p_1 = 1 \; .$$

The first item of the copy factor α-tuple is assigned to the total length of the sequence of consecutive p_1. We observe that the *second-to-last* token p_1 repeats once to the left. Thus, the total length of the sequence of consecutive p_1 is

$$k_1 = 2 \; .$$

Having established copy pattern and copy factor of the first decomposition step, we now scan the current string tokenization *left-to-right*. In each scan, as a general rule, if we encounter a token that is equal to the current copy pattern p_i we join at most k_i consecutive copies of it and merge these joined copies with the immediately following token into a new, larger token.

For the first parsing step we encounter an instance of $p_1 = 1$ at the first position of the current tokenization state. Hence, we merge the first and second token into the aggregate token $p^1q = 10$. As we continue to parse to the right two more aggregate tokens are generated by merging a chain of copy patterns with the immediate token after. Eventually, we reach the two copies of the copy pattern preceding the literal character a. Just as with previous matches of the copy pattern p_1 we join the two of them and merge them with the terminating character into the string $\hat{x}a = p_1^2a$. For this example the token boundaries after the first T-augmentation step are given by

$$xa = 1\ \ 0 \,|\, 1\ \ 1\ \ 0 \,|\, 0 \,|\, 1\ \ 1\ \ 0 \,|\, 1\ \ 1\ \ a\,.$$

Since there is more than one token left the loop goes into its second iteration with $i = 2$. Here the copy pattern is identified as $p_2 = 110$ with a copy factor of $k_2 = 1$. The subsequent *left-to-right* scan merges the two instances of p_2 with their immediately subsequent tokens resulting in the following token boundaries;

$$xa = 1\ \ 0 \,|\, 1\ \ 1\ \ 0\ \ 0 \,|\, 1\ \ 1\ \ 0\ \ 1\ \ 1\ \ a\,.$$

The algorithm continues with a third iteration of the loop using $p_3 = 1100$ and $k_3 = 1$ and yields

$$xa = 1\ \ 0 \,|\, 1\ \ 1\ \ 0\ \ 0\ \ 1\ \ 1\ \ 0\ \ 1\ \ 1\ \ a\,.$$

Finally, the fourth and last iteration eliminates the last token boundary using the copy pattern $p_4 = 10$ with $k_4 = 1$. At this point the T-handle $\hat{x}$ is an identical representation of all the information contained in x. Since no more tokens are left the algorithm terminates. Thus, the T-code for which xa is one of the longest code words required a total of $\alpha(x) = 4$ T-augmentation steps and is given by

$$\mathbf{S}^{(2,1,1,1)}_{(1,110,1100,10)}\,.$$

There are a total of $\|\mathbf{S}\|$ code words xa, in the T-code $\mathbf{S}^{(2,1,1,1)}_{(1,110,1100,10)}$ differing only in their last symbol a. From any of these longest code words we may deduce the above T-code by application of the T-transform algorithm.

3.4 T-Complexity

As we saw in the previous section the T-transform algorithm allows for the construction of a T-code from any of its longest code words $xa_i \in \mathbf{S}^+$ with $1 \leqslant i \leqslant \|\mathbf{S}\|$. Titchener observed in [79] that the algorithmic effort required to build such a T-code from the string $x \in \mathbf{S}^*$ could be used as a deterministic measure of how complex the information contained in that particular string is. We define the real valued T-complexity function $\zeta(x, \mathbf{S})$ as,

$$\zeta : \mathbf{X} \longrightarrow \mathbb{R}, \quad \text{where} \quad \mathbf{X} = \bigcup_{i=1}^{\|\mathbf{S}\|} xa_i, \quad x \in \mathbf{S}^*, \quad \text{and} \quad a_i \in \mathbf{S} \tag{3.6}$$

and, omitting the details of its derivation, compute it as the log-weighted sum of copy factors given by

$$C_T(x) \;=\; \zeta(x, \mathbf{S}) \;=\; \sum_{i=1}^{\alpha(x)} \log_2(k_i + 1)\ . \tag{3.7}$$

Visually the T-complexity of x may be looked at as a measure of how densely populated the T-code decoding tree for x is. In this decoding tree the set $\mathbf{X} = \bigcup_{i=1}^{\|\mathbf{S}\|} xa_i$ represents the set of all paths to the T-code's longest code words. As a thorough discussion of T-code decoding trees is beyond the scope of this thesis; more details about them and how they relate to the definition of T-complexity are provided in [27] to the interested reader.

Having introduced the prototype of the T-transform algorithm along with the deterministic complexity measure T-complexity, the remaining sections of this chapter provide an overview of the improvements made to the algorithm. We then conclude this chapter by introducing the proposed implementation which is currently the most efficient linear time and space implementation available.

3.5 T-Transform Algorithm Evolution

The naïve T-transform algorithm presented in Figure 3.2 is not a very efficient way to decompose a string into its copy pattern and copy factor representation. If we consider a string in which each character is unique the naïve implementation takes $O(n^2)$ time. This may easily be seen if we assume a string in which every character is unique. In each *left-to right* copy pattern matching pass i we have to compare $n-i$ characters yielding an over all runtime of $\sum_{i=1}^{\alpha(x)}(n-i) \leqslant n \times (n-1)/2 = O(n^2)$ [92]. Several efforts have been made to improve the runtime of the T-transform algorithm; a brief history of improvements is laid out in Table 3.2.

		Complexity	
Algorithm	**Description**	**time**	**space**
tcalc [89], *tlist* [93]	Character-by-character and length based token comparisons.	$O(n^2)$	$O(n)$
thash [94]	Hash function based token comparisons – average time complexity: $O(n)$.	$O(n^2)$	$O(n)$
ftd [96], *flott*	Unique integer based, constant time token comparisons.	$O(n)$	$O(n)$

Table 3.2: Computational complexity of T-transform algorithms.

Realizing that character-by character comparisons are the main bottleneck in any T-transform algorithm, Wackrow and Titchener were able to improve the naïve implementation in *tcalc* (1995) [89] by making note of the individual token length. In each pattern matching search pass they compare the token length with the copy pattern length first, and thus, were able to bypass a significant amount of character-by-character comparisons. The next three versions of T-transform algorithms were developed by Speidel and Yang. Their 2003 implementation, *tlist* [93], creates linked lists for tokens of the same length. Thus, in each parsing pass only the elements in the copy pattern corresponding length list have to be examined. However, as for the naïve implementation, the overall worst case runtime of *tcalc* and *tlist* is still $O(n^2)$ [92]. In 2005 Speidel and Yang published *thash* [94], the first algorithm which could achieve an average runtime of $O(n)$. The *thash* al-

gorithm stores tokens according to a computed hash value in doubly liked lists. The performance of the *thash* algorithm largely depends on the chosen hash function. Unfortunately, for the worst case of a hash function that assigns all tokens the same hash value the overall runtime has a $O(n^2)$ bound. Yang and Speidel addressed the shortcomings of *thash* in their 2005 paper [96] which introduced the *Fast T-Decomposition (ftd)* algorithm, the first true $O(n)$ time and space T-transform implementation. This thesis presents an improvement to the *ftd* algorithm, the *Fast Low Memory T-Transform (flott)*. The *flott* and *ftd* implementation are similar to one another in that both algorithms assign unique integer identifiers to tokens of the same kind. However, *flott* does so in a much more memory efficient way and has the added benefit of a slight improvement to the average time needed to assign token identifiers.

Before we discuss the *flott* algorithm in detail in Section 3.7, we introduce the data structures used in the *ftd* algorithm and provide a worked example of their usage to aid in the understanding of the *flott* algorithm.

3.6 Fast T-Decomposition *(ftd)*

This section will provide an overview of the *ftd* algorithm without going into as much detail as pseudo-code. Instead we introduce the data structures required to achieve linear time and space complexity and illustrate their use in an example. See [96, 92] for a more in-depth discussion of the *ftd* algorithm.

We now introduce the data structures needed to achieve linear time and space complexity in the *ftd* algorithm. We differentiate between three main data types used. With slight variations in their realization these data types are common to both the *ftd* and *flott* algorithms. The data type classes are:

Primitive data type: A data type with a one-to-one correspondence to a random-access machine's memory entity. Data types that we regard herein as such are: an **integer**, a **decimal** value, a **character** value and a **reference** pointing to the address of some memory content.

Composite data type: A data type that unites a fixed number of primitive data types in a single entity. We may access the individual data fields either by their numerical index or by their predefined label. An example of a composite data type is a ***string*** of length n defined as **character [**n**]**.

Abstract data type: A data type that is primarily defined by the operations it can carry out on a single or a collection of instances of primitive or composite data types. An example of an abstract data type is a linked ***list*** allowing to manipulate its list elements through a set of list operations.

Above, we have printed primitive data types in **bold** to distinguish them from composite and abstract data types which we printed in ***bold italics***. For the remainder of this thesis we shall adopt this typographic convention in data type diagrams, symbols, and pseudo-code. The next section will examine the specifics of *ftd's* doubly linked lists data structures.

3.6.1 Token and Match List Data Structures

The central data types used in *ftd* and *flott* are doubly linked list storing string tokens in form of a composite data type. Figure 3.4 shows the abstract data type diagram for the doubly linked list.

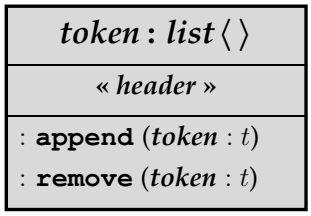


Figure 3.4: Abstract ***list*** data type diagram of *ftd*.

We assume that the reader has basic knowledge of doubly linked lists and list operations as per the discussions contained in [65]. The doubly linked lists used in this thesis maintain a header data structure, which is the composite data type through which the list elements are accessed. The list header stores an integer index, or in *ftd's* case a reference, to head and tail of the list. In addition, we make use of "append" and "remove" as the sole functional list operations. We make note of the fact that both functional operations require $O(1)$ time to execute. Equally, accessing head, tail, or tokens directly adjacent to one another takes constant time.

The *ftd* algorithm compares, and groups tokens according to their unique integer identifier *(uid)*. This has the advantage that token comparisons are reduced to simple integer comparisons that are carried out in constant time on the adopted random-access machine model with *uniform cost measure*. The idea of classifying

tokens according to an integer identifier is akin to the idea of hash values in *thash*. However, in contrast to *thash*, the *ftd* algorithm provides a mechanism for assigning token identifiers resistant to collision.

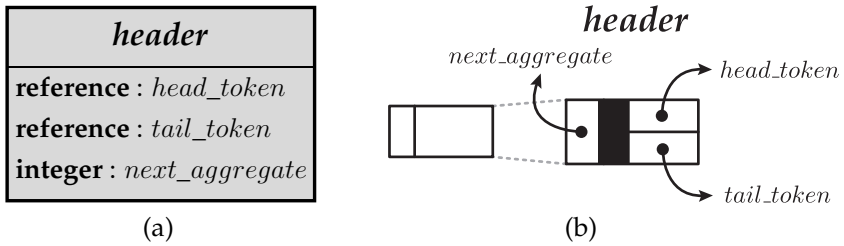


Figure 3.5: Match list ***header*** data type diagram (a) and symbol (b) of *ftd*.

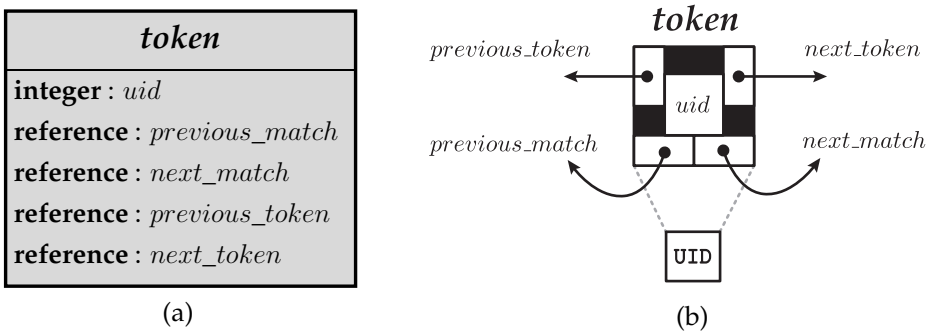


Figure 3.6: ***token*** data type diagram (a) and symbol (b) of *ftd*.

Similar to previous T-transform implementations, the *ftd* algorithm starts out with the input string split into single character tokens. Figure 3.6 shows the token data structure and symbol notation used. The algorithm stores tokens in a doubly linked token list. Each token, represented by its uid, is also part of a doubly linked match list which links all tokens with the same uid. The match list header used is shown in Figure 3.5 and includes an additional integer field whose purpose is discussed later on. Keeping match lists allows us to eliminate the expensive *left-to-right* search pass for the copy pattern in each transform level which was necessary in the naïve T-transform algorithm of Section 3.3.1. In the *ftd* algorithm copy pattern matches are all located in the same match list making it possible to skip over all non-matching tokens. The number of required match lists grows with the number of newly generated aggregate tokens. In the next section we will explain in detail how the *ftd* algorithm manages its match lists and assigns $uids$ to new aggregate tokens.

3.6.2 Unique Identifier Assignment

We have not yet explained the motivation behind the additional "$next\_aggregate$" integer field in the header structure of Figure 3.5. This field plays a principal role in *ftd's* unique identifier assignment mechanism which is the subject of this section.

The *ftd* algorithm stores the headers for all match lists in a preallocated array. Naturally the question arises of how large this array must be. To answer this question, consider the *ftd* algorithm in its initial state. We require at least $\|\mathbf{S}\|$ $uids$ to represent the alphabet from which the string is composed. Over the course of all subsequent T-transform levels a string of length n cannot generate more than $n-1$ aggregate tokens [92]. Hence, an array of size $(n-1)+\|\mathbf{S}\|$ is sufficient.

We start assigning $uids$ to the initial single character tokens by defining a one-to-one mapping function on the alphabet $\mathbf{S}$, assigning every individual alphabet symbol to an integer in the range from 1 to $\|\mathbf{S}\|$. This integer value becomes the uid for the token. Simultaneously this uid serves as an offset into the preallocated array storing the header for the match list to which the token belongs. The assignment of $uids$ for subsequently generated aggregate tokens is more complex and is illustrated in the following example.

Example 3.6.1 (Unique Identifier Assignment in *ftd*, Binary). This example illustrates how the *ftd* algorithm assigns $uids$ to new aggregate tokens, and how these are placed in the appropriate match lists. We use the binary alphabet $\mathbf{S} = \{0, 1\}$, and the string subject to decomposition is given as $x = 1011101101111$. The T-code in which x forms the prefix to the longest code words is $\mathbf{S}^{(4,1,1,1)}_{(1,110,1110,10)}$. We define the ordinal number of $a \in \mathbf{S}$ as the one-to-one integer mapping given by

$$\operatorname{ord}(a) = \begin{cases} 1 & \text{if } a = 0 \\ 2 & \text{if } a = 1 \end{cases}. \tag{3.8}$$

Figure 3.7 sketches the *ftd* algorithm in its initial state showing only the first six tokens in x. The first and second entries of the match list header array are occupied, linking the tokens of one alphabet symbol each. Note that even though not explicitly drawn, the match list headers are assumed to maintain a reference to the tail of their match list.

The copy pattern for the first level of the T-transform is determined as $p_1 = 1$ with a copy factor of $k_1 = 4$ and uid of 2. Figure 3.8 shows how the first aggregate

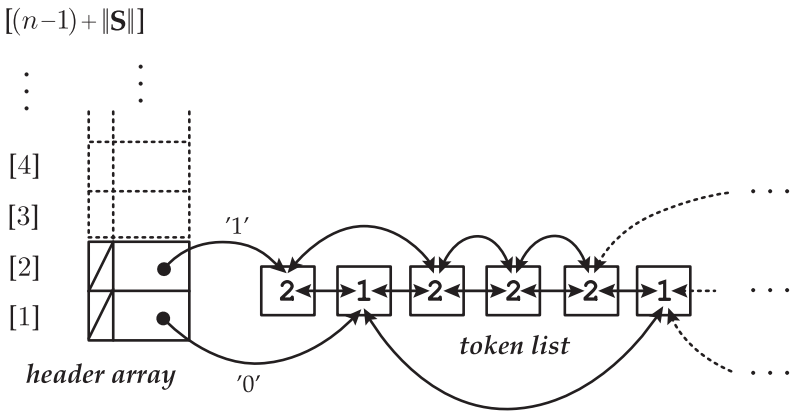


Figure 3.7: Initialization state of *ftd*.

token is formed. The aggregate prefix p_1 and suffix q token are located in the first and second position of the token list. We merge both into the new aggregate token $g_1 = p_1 q = 10$ and assign it the uid 3, which is the index to the next free slot in the header array. This free slot becomes the new home for the aggregate token's match list header. Finally, we connect the match list of the former aggregate suffix to the one of the new aggregate token via the $next\_aggregate$ field.

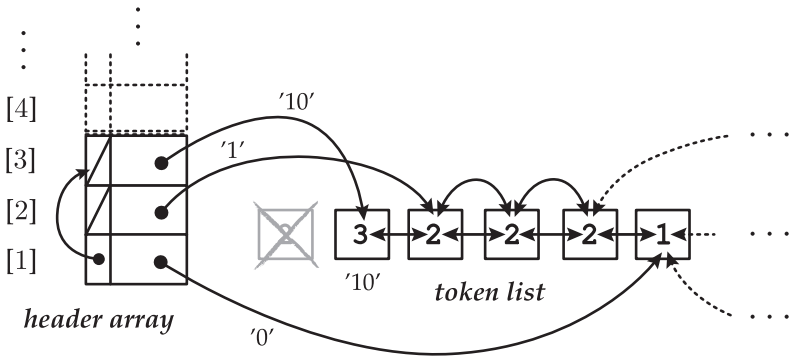


Figure 3.8: Creation of aggregate token $g_1 = 10$ in *ftd*.

The formation of the second aggregate token is depicted in Figure 3.9. Here the aggregate prefix $p_1^{k'}$ is made up from $k' = 3$ consecutive copies of the copy pattern. The aggregate suffix q has a uid of 1, the same as the previous aggregate suffix, suggesting that we might have generated an aggregate token of the same kind before. Thus, we try following at most $k' = 3$ consecutive $next\_aggregate$ links to determine the aggregate tokens's uid. However, we get stopped short after just following one link leading us to the match list header of uid number 3. We then extend the $next\_aggregate$ link by "daisy-chaining" the next two free header array

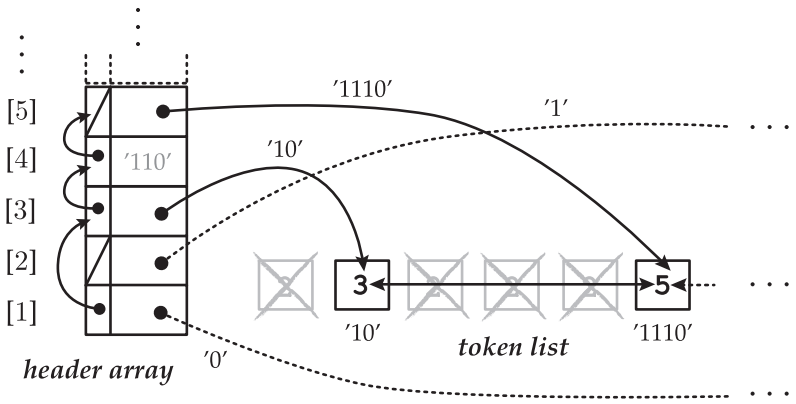


Figure 3.9: Creation of aggregate token $g_2 = 1110$ in *ftd*.

slots as additional nodes. Thereafter, we merge aggregate prefix and suffix into the aggregate token $g_2 = 1110$ and assign it the uid of 5. Thus, when we subsequently encounter the aggregate token $g_3 = 110$ we automatically find its match list and uid using the same process. The uid assignment mechanism is carried out in the same fashion for all remaining aggregate tokens which concludes this example.

3.6.3 Time and Space Complexity

The *ftd* algorithm requires $O(n)$ space to store the token list and $O(n + \|\mathbf{S}\| - 1)$ space to store the header array. Hence, the overall space complexity is of order $O(n)$. More specifically, from the token and match list header data structure diagrams in Figures 3.5 and 3.6, we see that 2 integers and 6 references need to be stored per input character. On a 32-bit random-access machine, on which both integers and references are 4 byte wide, we therefore require $32n$ bytes of memory. In contrast, on a 64-bit random-access machine references double in size which raises the overall memory requirements to $56n$ bytes.

The overall time complexity of the *ftd* algorithm is composed from the time it takes to parse the input in its initial single character token state and the time required for the subsequent aggregate token construction. The initial parsing pass takes $O(n)$ time. Consider the formation of an arbitrary aggregate token $p^{k'}q$. The assignment of its uid takes $O(k')$ time; however, with each formed aggregate token the effective input size shrinks by k' tokens as well resulting in $O(n)$ time to generate all $uids$. No more than $n - 1$ aggregate tokens are generated which means that we require no more than $O(n)$ time to generate all aggregate tokens. Thus, we

conclude that the overall time complexity of the *ftd* algorithm is of order $O(n)$.

In this section we gave a brief overview of the *ftd* algorithm. In light of the high memory requirements of the algorithm on a 64-bit random-access machine, the next section introduces the much more memory efficient *flott* implementation.

3.7 Fast Low Memory T-Transform *(flott)*

This section outlines the differences between *flott's* and *ftd's* data structures, provides a full pseudo-code description for the *flott* algorithm, and sheds light on the key observations that decrease *flott's* overall memory usage. Subsequently, we explain *flott's* uid assignment procedure with the aid of an example.

The *flott* and *ftd* algorithm are very similar in that both algorithms share the notion of token list and match lists. However, *flott* provides a much more memory conservative overall implementation by eliminating the need for a match list header array.

Comparing the token data structure of *ftd* and *flott* (see Figures 3.6 and 3.10) we notice that the references in *ftd's* token data structure have been replaced by integers. Thus, a token is represented by a 5 integer tuple: 1 integer is used for the uid and 4 integers are used to identify the token's neighbours in it's token and match list. Essentially, the 4 token linking integers serve as offsets to blocks of $5 \times 4 = 20$ bytes of memory within a single consecutive memory allocation. As a consequence of not using architecture dependent machine references in its data structures, the overall memory usage in *flott* remains at $20n$ bytes on a 64-bit random-access machine whereas the the memory consumption of of *ftd* increases from $32n$ to $56n$ bytes.

The overall space complexity of *flott* is further reduced by recognizing that memory, in contrast to time, can be reused. Consider once more Example 3.6.1. In Figure 3.9 the *ftd* algorithm generates the aggregate token $p_1^3 q = 1110$. Once the aggregate token has formed, the memory previously occupied by the aggregate prefix remains unused for the remainder of the algorithm. The *flott* algorithm exploits this fact by simply reusing the memory of aggregate prefixes to store the headers of aggregate match lists.

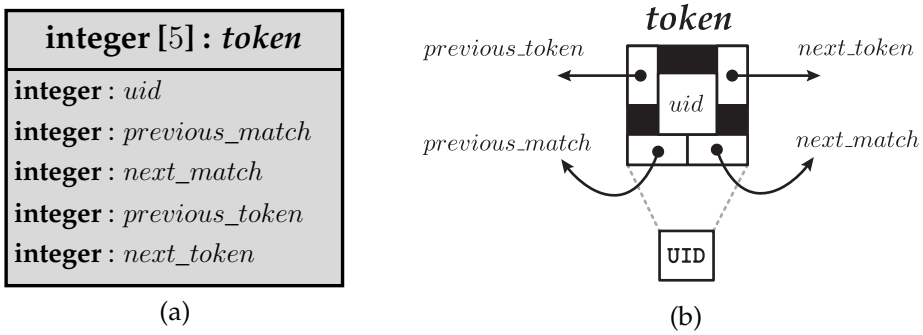


Figure 3.10: ***token*** data type diagram (a) and symbol (b) of *flott*.

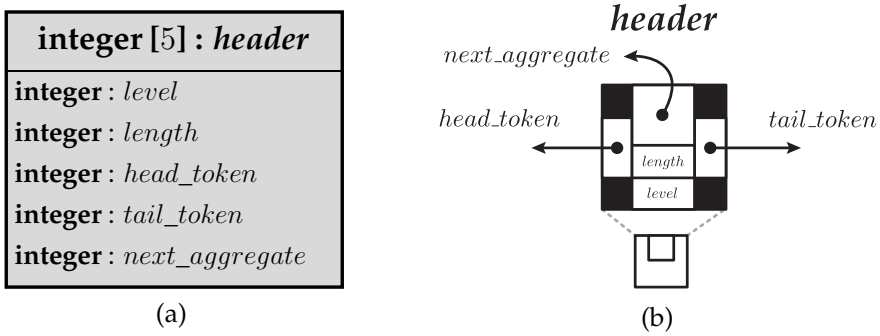


Figure 3.11: Match list ***header*** data type diagram (a) and symbol (b) of *flott*.

For this purpose the data structures of token and match list header need to have the same memory footprint. Their data type diagrams and symbols are drawn in Figures 3.10 and 3.11. Match list header and token data structure are in essence simple five element integer arrays. The match list header introduces two additional fields to keep track of the list's length, and the transform level at which the header was last visited. As we will see shortly, reusing the aggregate prefix memory allows us to eliminate the match list header array previously used in *ftd*, reducing the overall space complexity of *flott* to $20n$ bytes.

The *flott* algorithm is illustrated as an abstract data type in Figure 3.12. The algorithm is operating on a matrix of integers stored in a two dimensional integer array. Each column in the matrix represents a 5 integer tuple which can represent either a token or a match list header. The number of columns contained in the matrix is the sum of the length of the input string n and the cardinality of the underlying alphabet $\|\mathbf{S}\|$, i.e. $M = n + \|\mathbf{S}\|$.

flott
character [n] : $string$
integer [$5 \times M$] : $memory$
integer : $levels$ ***result***$_A$ [$self.levels$] **/** ***result***$_B\langle self.levels\rangle$: $result$
: **integer** : **`ccr`** (***token*** : t) : **integer** : **`ccl`** (***token*** : t) : **integer** : **`ord`** (**character** : a) : ***header***, **integer** [$5 \times M$] : **`init`** (**character** [n] : x **,** **integer** : n) : **`aggregate`** (**integer** : i, v', w **,** ***token*** : q)

Figure 3.12: Abstract data type of *flott* algorithm.

Subroutine	Description
`ccr`, **`ccl`**	Given a token, this subroutine counts the consecutive tokens in a run to the right: **`ccr`** (to the left: **`ccl`**).
`ord`	Given a character $a \in \mathbf{S}$ as input, this subroutine returns the character's ordinal number such that $\mathrm{ord}\,(a) \in \{1, \ldots, \|\mathbf{S}\|\}$.
`init`	Given a string x, this subroutine initializes the initial token list and match lists. The **`init`** subroutine returns a header for the token list and an array of $5 \times (n + \|\mathbf{S}\|)$ integers storing the entirety of tokens and match lists (see Figure 3.14 for the pseudo-code listing).
`aggregate`	Given the aggregate suffix q, the length of the aggregate prefix v', the length of the aggregate suffix w, and the current T-transform level i, this subroutine creates a new aggregate token along with its proper uid (see Figure 3.15 for the pseudo-code listing).

Table 3.3: Descriptions of subroutines used in *flott*.

The *flott* algorithm can provide information about generated copy patterns and copy factors as output. For this purpose either a linked list based or an array based result data type can be used. The two possible result data types are discussed in more detail in Section 3.7.1. The *flott* algorithm makes use of a number of subroutines whose functionality is briefly outlined in Table 3.3.

A full pseudo-code implementation of *flott* is given in Figure 3.13. Before studying the provided pseudo-code and *flott's* uid assignment procedure which is discussed in Example 3.7.1, we remind the reader that a detailed pseudo-code notation explantation is given in Table 3.1.

▶ **Algorithm** ▷ `flott`

input : x /* string $x \in \mathbf{S}^*$ */
output: $levels$ /* t-transform level count: α */
p /* copy pattern α-tuple */
k /* copy factor α-tuple */
c /* t-complexity */

initialization:

1 $n \longleftarrow |x|$ /* token boundary count */
2 $c \longleftarrow 0$ /* initialize t-complexity value */
3 $i \longleftarrow 1$ /* initialize t-transform level counter */

/* initialize memory of level zero tokens / match lists headers (m) and token list header ($tlist$) */
4 $tlist,\ m \longleftarrow$ **init** (x, n)
5 **while** $n > 0$ **do**
6 $p[\,i\,] \longleftarrow tlist\,\langle\, n\,\rangle$ /* i^{th} copy pattern token */
7 $k[\,i\,] \longleftarrow$ **ccl** $(p[\,i\,])$ /* i^{th} copy factor */
8 $v \longleftarrow (@p[\,i\,]) - p[\,i\,].previous\_token$ /* copy pattern length */
/* remove chain of copy patterns from their lists */
★A ▷ 9 **foreach** $t \in tlist\,\langle n-k[\,i\,]+1,\ n\rangle$ **do**
10 $tlist,\ m_{header}[\,p[\,i\,].uid\,]$: **remove** (t)
11 **end**
12 $n \longleftarrow n - k[\,i\,]$ /* update token boundary count */
13 $c \longleftarrow c + \log_2(k[\,i\,]+1)$ /* update t-complexity */
/* create i^{th}-level aggregate tokens by iterating through the elements of the match list of $p[i]$ */
★B ▷ 14 **forall the** $tlist\,\langle j\rangle \in m_{header}[\,p[\,i\,].uid\,]\langle 1\rangle$ **do**
15 $k' \longleftarrow \min($ **ccr** $(tlist\,\langle j\rangle),\ s)$
16 $q \longleftarrow tlist\,\langle j + k'\rangle$ /* aggregate suffix token */
17 $w \longleftarrow (@q) - q.previous\_token$ /* aggregate suffix length */
★C ▷ 18 **foreach** $t \in tlist\,\langle j,\ j+k'-1\rangle$ **do**
19 $tlist,\ m_{header}[\,p[\,i\,].uid\,]$: **remove** (t)
/* reclaim token memory for match list use */
20 $m_{header}[@t].length \longleftarrow 0$
21 $m_{header}[@t].level \longleftarrow i$
22 **end**
23 $n \longleftarrow n - k'$ /* update token boundary count */
/* determine uid & match list of new aggregate token */
24 **aggregate** $(\,i,\ k' \times v,\ w,\ q)$
25 **end**
26 $i \longleftarrow i + 1$ /* increment iteration counter */
27 **end**
28 $levels \longleftarrow i - 1$

Figure 3.13: Pseudo-code listing of *flott* algorithm.

```
▶ Function ▷ flott : init()

input : x                                          /* string x ∈ S* */
        n                                          /* string length */
output: tlist, m        /* token list and allocated memory array */
initialization:
1   M ⟵ n + ‖S‖         /* number of token cells to be allocated */
    /* all-zero integer array allocation                          */
2   m ⟵ integer[5×M] ⟵ {0, ...}
3 for j ⟵ 1 to n do
4 |   t ⟵ m_token[ j ]
5 |   t.uid ⟵ n + ord(x[ j ])
6 |   tlist, m_header[ t.uid ] : append(t)
7 end
```

Figure 3.14: Pseudo-code listing of *flott* init function.

```
▶ Method ▷ flott : aggregate()

input : i                                      /* t-transform level */
        v'                                /* length of aggregate prefix */
        w                                 /* length of aggregate suffix */
        q                                     /* aggregate suffix token */
initialization:
1   h ⟵ m_header[ q.uid ]                              /* match list of q */
2 m_header[ q.uid ] : remove(q)   /* remove q from its old match list */
3 while true do
4 |   if h.level ≠ i then /* create new aggregate token uid */
5 |   |   h.level ⟵ i
6 |   |   h.next_aggregate ⟵ (@q)
7 |   |   q.uid ⟵ (@q) − w
8 |   |   m_header[ q.uid ] : append(q)
9 |   |   break
10|   else /* look up aggregate token uid */
11|   |   g ⟵ m_token[ h.next_aggregate ]
12|   |   u ⟵ g.previous_token + v'
13|   |   if g.uid ⩾ u then /* aggregate token uid found */
14|   |   |   g.uid ⟵ u
15|   |   |   m_header[ u ] : append(q)
16|   |   |   break
17|   |   end
18|   |   h ⟵ m_header[ g.uid ]
19|   end
20 end
```

Figure 3.15: Pseudo-code listing of *flott* aggregation method.

Example 3.7.1 (Unique Identifier Assignment in *flott*, Binary). This example is intended to demonstrate how the *flott* algorithm generates $uids$ and places newly generated aggregate tokens in their appropriate match lists. Once again we consider the binary string $x = 1011101101111$, assembled from the alphabet $\mathbf{S} = \{0, 1\}$, and prefix to the longest code words in the T-code $\mathbf{S}^{(4,1,1,1)}_{(1,110,1110,10)}$. To aid readability, we will only indicate the match lists' tail links – the links connecting the match lists' heads are implied.

The *flott* algorithm pseudo-code in Figure 3.13 starts by making a call to the *init* function (Figure 3.14) which returns the token list header and an integer matrix of size $5 \times M$, where $M = n + \|\mathbf{S}\| = 15$. Dividing the matrix in 5×1 columns, the first n columns represent the single character tokens doubly linked to one another using their column offsets. The remaining $\|\,\mathbf{S}\,\|$ columns are initialized with the alphabet's match list headers. As a general rule, a token's uid is always assigned to the column offset at which its match list header is located. Because the initial match lists are located at offsets $n + 1$ and larger, the initial tokens receive $uids$ of the form $n + \mathrm{ord}\,(a)$, where $a \in \mathbf{S}$ and the ordinal function, $\mathrm{ord}\,(a)$, was previously defined in Equation 3.8.

From the pseudo-code in Figure 3.13 we can see that the first copy pattern of the T-transform is established as $p_1 = 1$ and has a length of $v = 1$. The subsequent call to *ccr* yields a copy factor of $k_1 = 4$. Loop $\star_{\mathbf{A}}$ removes the tokens in the chain of copy patterns from the end of their respective token and match lists. We briefly mention here that the *ccr* function can easily be absorbed into loop $\star_{\mathbf{A}}$ and is just shown as a separate entity to support readability. In a similar way the *ccl* function may be absorbed into loop $\star_{\mathbf{C}}$. Figure 3.16 depicts *flott's* initial parsing state after having merged the copy pattern run into the T-handle $\hat{x} = p_1^{k_1} = 1111$.

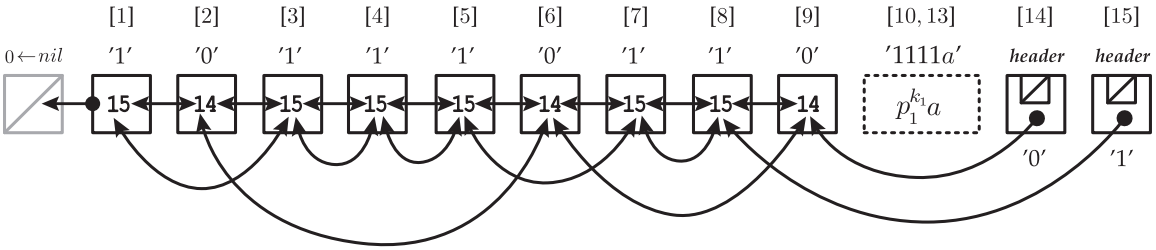


Figure 3.16: Initialization state of *flott*.

The remaining instances of the copy pattern are linked via the match list header at offset 15. All tokens in this match list will be part of aggregate tokens

at the end of the first T-transform level. Loop $\star_{\mathbf{B}}$ begins by locating the first copy pattern match located at offset 1. A call to *ccr* gives us that the aggregate prefix copy factor $k' = 1$. Before we merge aggregate prefix and suffix into the first aggregate token $g_1 = p_1^{k'}q = 10$, we record the aggregate suffix length in symbols $w = 1$ and initialize the aggregate prefix for use as a match list header. The new aggregate token needs a uid and match list which is determined by a call to the *aggregate* method (see Figure 3.15). Since this is the first time we form an aggregate token with the suffix $q = 0$, the suffix's match list has not been visited yet, and we have to generate its uid and match list new. As shown in Figure 3.17, the aggregate

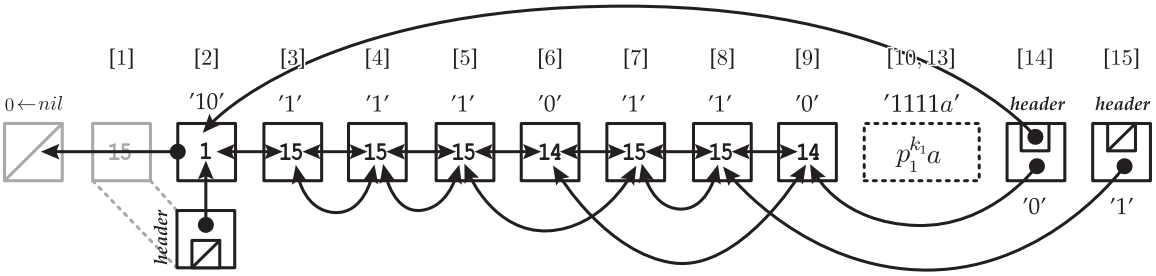


Figure 3.17: Creation of aggregate token $g_1 = 10$ in *flott*.

token is reusing the memory of the former aggregate prefix for its match list. It's uid, equal to it's match list header offset, is determined by simply subtracting the length of the aggregate suffix from its own offset as follows,

$$\begin{aligned} g_1.uid &= (@q) - w \\ &= 2 - 1 = 1\,. \end{aligned}$$

The aggregation process is completed by updating the aggregate suffix's match list $level$ field to the current T-transform level and letting the $next\_aggregate$ field point to the newly generated aggregate token.

The second aggregate token $g_2 = 1110$ is formed in Figure 3.18. The aggregate prefix concatenates $k' = 3$ copies of the copy pattern and the aggregate suffix was encountered previously. Thus, the aggregate suffix match list header has been visited before and we may already have a uid and match list for g_2. We follow the $next\_aggregate$ link in the aggregate's suffix match list leading us to the previously generated aggregate token g_1. We try to locate the match list for g_3 by computing a

preliminary uid as follows,

$$\begin{aligned} g_2.uid &= g_1.previous\_token + k' \times v \\ &= 0 + 3 \times 1 \; = \; 3 \; . \end{aligned}$$

Since the uid evaluated for g_2 exceeds the one assigned to g_1, we know that the match list header for g_2 is not located in any of the match list slots that were initialized for reuse during the creation of g_1. Therefore, we switch from the match list of the aggregate suffix to the match list of g_1. This is the first time the match list header of g_1 is visited during this T-transform level which means we have to generate uid and match list new. In analogy to the previous aggregate token a valid uid for g_2 is determined as

$$\begin{aligned} g_2.uid &= (@q) - w \\ &= \quad 6 - 1 \; = \; 5 \; . \end{aligned}$$

Once again, the aggregation process is completed by updating the aggregate suffix's $level$ field in the match list header with the current T-transform level and letting the $next\_aggregate$ field point to the newly generated aggregate token.

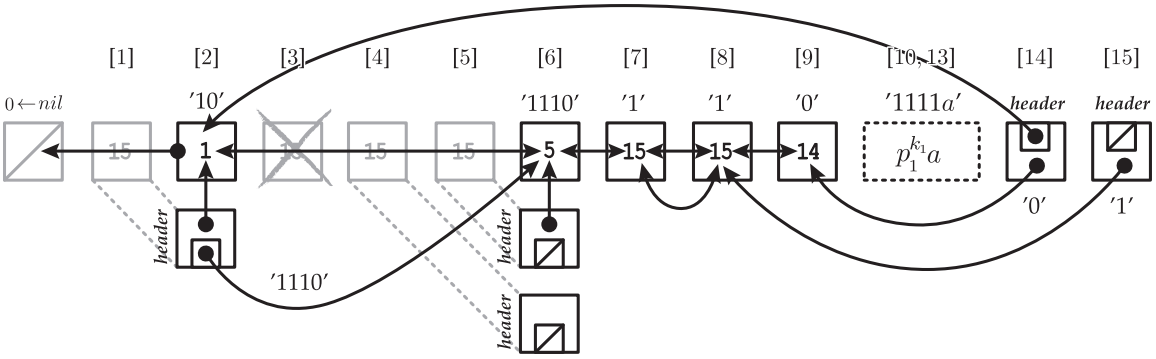


Figure 3.18: Creation of aggregate token $g_2 = 1110$ in *flott*.

The last aggregate token created in the first T-transform level is $g_3 = 110$ as shown in Figure 3.19. The aggregate suffix is once more $q = 0$. We reach g_1 via the $next\_aggregate$ link and compute a preliminary aggregate uid as

$$\begin{aligned} g_3.uid &= g_1.previous\_token + k' \times v \\ &= 0 + 2 \times 1 \; = \; 2 \; . \end{aligned}$$

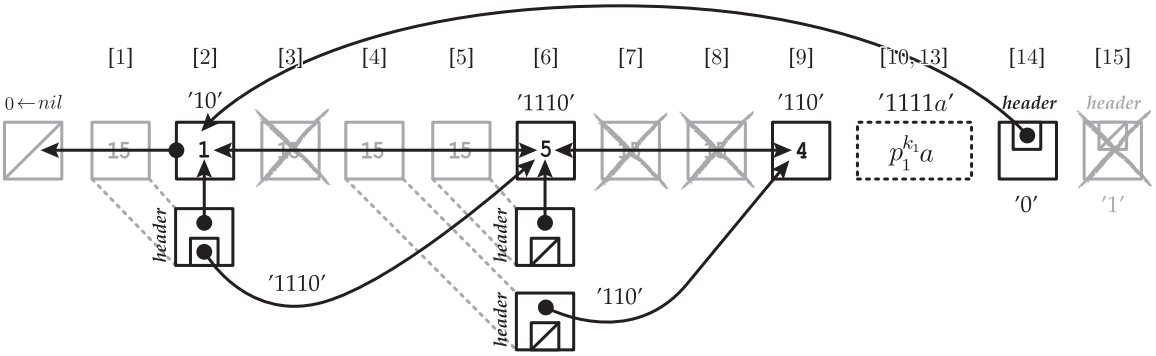


Figure 3.19: Creation of aggregate token $g_3 = 110$ in *flott*.

The determined uid is larger than the uid of g_1 forwarding us to g_2 via the $next\_aggregate$ link in the match list header of g_1. Using the uid of g_2 we evaluate the new preliminary aggregate uid to

$$\begin{aligned} g_3.uid &= g_2.previous\_token + k' \times v \\ &= 2 + 2 \times 1 \;=\; 4\,. \end{aligned}$$

In this case the aggregate uid is valid because it is smaller than the one of g_2 allowing us to append the aggregate token to the match list located at offset 4. At this point all matches of the copy pattern p_1 have been merged into aggregate tokens and the first level of the T-transform is completed.

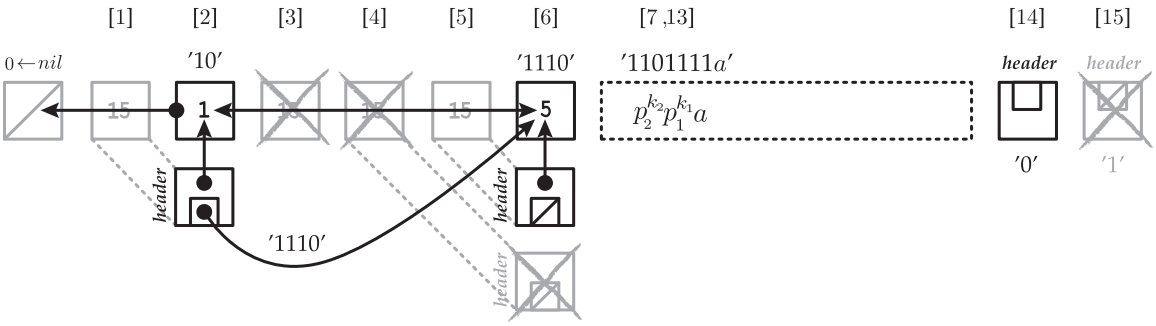
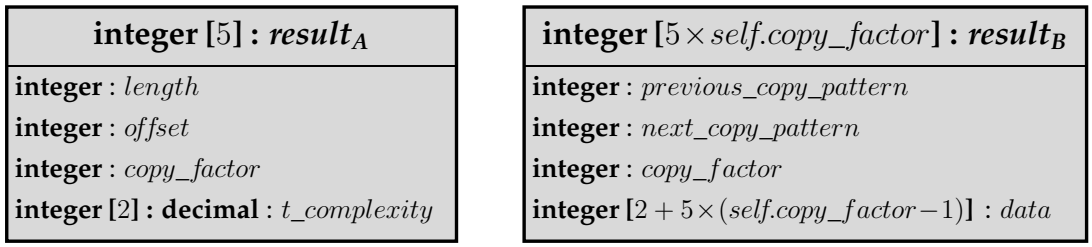


Figure 3.20: Parsing state after completion of the second *flott* level.

In this particular example the remaining T-transform levels do not generate additional aggregate tokens but only increase the length of the T-handle incrementally. The result for the second T-transform level is shown in Figure 3.20. Concluding this example we note that in general aggregate tokens may be generated at any level of the T-transform using the same procedures as outlined above.

3.7.1 Collecting T-Transform Results

As a T-transform result we often would like to record the copy factor along with the length and start offset of the copy pattern in the input for each T-transform level i. Further, it is often useful to record the rate of change of the T-complexity value over the string decomposition process. This rate is herein referred to as the *instantaneous T-complexity rate* and is discussed in more detail in Chapter 5.



(a) (b)

Figure 3.21: T-transform result data types in *flott*: (a) array, and (b) linked list.

Fortunately, we do not require any additional memory to store the copy pattern, copy factor, and T-complexity values for each T-transform level in *flott*. Consider once again Figure 3.20 in Example 3.7.1. The incrementally growing number of memory slots that spatially represent the tokens of the T-handle in the token list are not accessed or needed in any way by the *flott* algorithm. Thus, we may use this memory to store T-transform output. Figure 3.21 shows two examples of possible data types for T-transform results.

3.7.2 Time and Space Complexity

The *flott* algorithm requires $O(n + \|\mathbf{S}\|)$ space to store the token and match lists. Hence, the overall space complexity is order $O(n)$. More specifically, when using 4 byte long integers we need $20n$ bytes of memory on both, 32-bit and 64-bit random-access machines.

The overall time complexity of the *flott* algorithm is composed from the time it takes to parse the input in its initial single character token state and the time required for the subsequent aggregate token construction. As for the *ftd* algorithm the initial parsing pass takes $O(n)$ time in *flott*. The assignment of a single uid for the aggregate token $p^{k'}q$ takes $O(1)$ in the best case and $O(k')$ time in the worst case. With each formed aggregate token the effective input size shrinks by k' giving $O(n)$

time to establish all $uids$. The total number of aggregate tokens does not exceed $n - 1$ yielding an overall runtime of $O(n)$.

3.8 Summary

In this chapter we introduced the prototype of the T-transform, a string decomposition algorithm that can be used to measure the deterministic complexity of a string. The first naïve T-transform algorithm presented at the beginning of the chapter had a runtime of $O(n^2)$ and was subsequently improved to the $O(n)$ *ftd* algorithm by Speidel and Yang. Finally, we discussed the proposed implementation, *flott*, and outlined its efficient memory reuse strategy in detail. In the next chapter we will provide a comparative analysis of *ftd* and *flott* along with a brief comparison to some well established suffix tree implementations.

Chapter 4

Comparative Analysis

Having developed a more memory efficient $O(n)$ T-transform algorithm in the previous chapter, the performance of the *C*-implementations of *ftd* and *flott* are compared in Section 4.1 of this chapter. Furthermore, we juxtapose the memory requirements of *ftd* and *flott* with well established suffix tree implementations in Section 4.2.

4.1 T-Transform Benchmark Evaluation

This section compares the runtime of *ftd* and *flott* for two types of input sources on commercially available hardware. One of the input sources was derived from the *Enron* email data set [15] and represents a textual data source with a high degree of redundancy while the other is representing a maximum entropy source and was obtained from a *Quantis*™ random number generator [37]. The experiments of this section assume byte-wise string decomposition, i.e. $\| \mathbf{S} \| = 256$, and were carried out on an Intel Xeon 3.0 *gigahertz* (GHz) processor with 6 *megabytes* (MB) cache and 16 *gigabytes* (GB) of main memory. Both algorithms were compiled with *gcc* using identical code optimization settings (*-O3*).

Figure 4.1 and Figure 4.2 show the runtime measurements for string lengths of up to 256 MB obtained from a Quantis random number generator and Enron data set respectively. Both, *ftd* and *flott* display a linear runtime behaviour in practice which is what we expect. Interestingly enough, in our experiments, the percentage of *flott's* runtime improvement over *ftd* is influenced by the string length and shows an improvement in runtime of approximately 20% asymptotically.

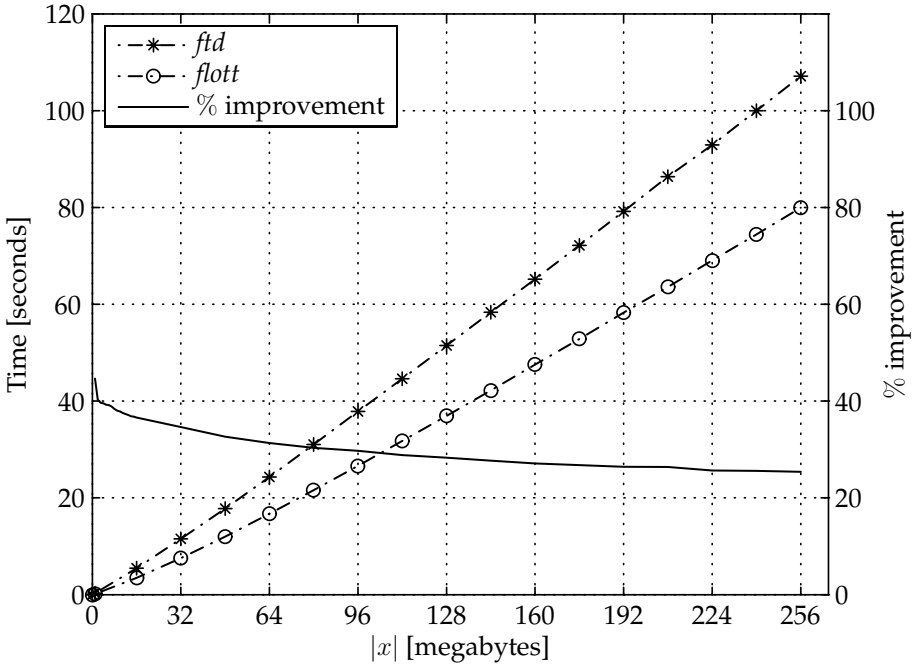


Figure 4.1: Runtime of *ftd* and *flott* – Quantis random number generator.

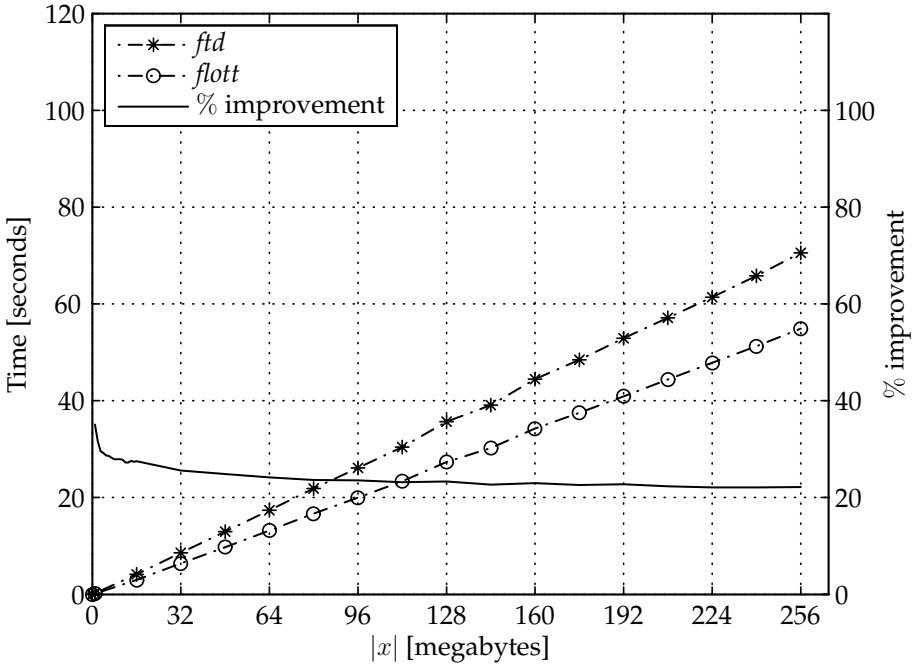


Figure 4.2: Runtime of *ftd* and *flott* – Enron email data set.

The reason why *flott* has a higher speed-up for shorter strings is explained as follows. The *ftd* algorithm loads the input string in its entirety to memory before it initializes its data structures and starts decomposing the string. In contrast, *flott* initializes its data structures directly from the storage device without buffering the input. While the *flott* implementation can buffer the input on request, for the computation of a string's T-complexity alone there is no need for the input to be retained in memory. Moreover, since *flott* does not make use of a header array, it needs to allocate and initialize much less memory than *ftd*. These fixed up-front costs become less and less significant with growing input size since both algorithms then allocate and manipulate gigabytes of memory impacting performance due to cache misses.

Furthermore, in order to calculate the T-complexity of a string we need to evaluate the binary logarithm at each of the T-transform levels. Since we are only ever evaluating the logarithm for integer values the call to the math library in *ftd* has been replaced with a logarithm look-up table in *flott* resulting in some overall performance gain. Finally, the compiler generated assembly code of *flott* was hand inspected to assure that it would make the most efficient use of pointer arithmetic which contributed positively to the overall performance of *flott*.

To explain the overall lower processing times in Figure 4.2 versus those in Figure 4.1 we note that in the Quantis random number generator strings (Quantis strings) we are not likely to encounter a large number of very long copy patterns and expect the copy factors to be $p_i = 1$ most of the time. Thus, it will require a higher number T-transform levels to decompose Quantis strings than strings from the Enron data set (Enron strings). Hence, for the Quantis strings more uids and match lists have to be generated than for Enron strings which explains the overall lower slopes in Figure 4.2.

4.2 Juxtaposition with Suffix Trees

As already mentioned in Chapter 2 the *LZ*-complexity of a string may be calculated in linear time and space using a suffix tree. Essentially, *LZ*-complexity is a measure of the effort needed to construct such a tree. Due to their reputation of requiring vast amounts of memory and their non-trivial linear time implementations suffix trees have seen limited use [45]. A naïve implementation of a suffix tree for example requires $40n$ bytes of memory in the worst case [56].

Algorithm	Maximum input length $n^{\dagger}$	Complexity	
		time	space‡
flott	$(2^{32}-1) - \|\mathbf{S}\| = 4,294,967,039$	$O(n)$	$20n$
ftd [96]	$(2^{32}-1) - \|\mathbf{S}\| = 4,294,967,039$	$O(n)$	$56n$
McCreight [55]	$(2^{32}-1)/29 = 148,102,320$	$O(\|\mathbf{S}\| \times n)$	$28n$
ILLI [45]	$(2^{27}-1) = 134,217,727$	$O(\|\mathbf{S}\| \times n)$	$20n$
IHTI [45]	$(2^{27}-1) = 134,217,727$	$O(n)$	$24n$

† 64-bit random-access machine, using 32-bit wide integers for referencing, $\|\mathbf{S}\| = 256$

‡ space complexity in bytes

Table 4.1: Comparison of T-transform and suffix tree implementations.

Table 4.1 shows how *ftd* and *flott* compare to well established suffix tree construction algorithms and is based on the data provided in [45]. An early implementation by McCreight [55] reduced the worst case memory requirements for suffix trees to $28n$ bytes. Two decades later Kurtz conceptualized and implemented two improved versions of McCreight's suffix tree construction algorithm in [45] further reduceding memory consumption. Kurtz refers to his improved implementations as *ILLI* (improved linked list implementation) and *IHTL* (improved hash table implementation). *ILLI* requires $20n$ bytes of memory in the worst case and runs in $O(\|\mathbf{S}\| \times n)$ time. Given that *ILLI* becomes less practical for larger integer alphabets the *IHTL* implementation makes use of hashing to achieve $O(n)$ runtime. However, *IHTL* achieves this improvement at the expense of $4n$ bytes of additional memory usage.

The results provided in Table 4.1 are for a 64-bit random-access machine using 32-bit integer referencing. We see that *flott* requires $20n$ bytes (64.3%) less memory than *ftd* and is on par with Kurtz's most memory efficient suffix tree implementation. However, we need to mention that the implementation of Kurtz makes use of variable size linked list nodes, referred to as "long" and "short" nodes, which may reduce the average space consumption to less than the reported worst case. At this point neither *ftd* nor *flott* make use of variable size linked list nodes, but such an implementation is theoretically possible and remains the subject of future work.

The maximum input size is an other important factor to consider when comparing algorithms on the *uniform cost measure* random-access machine model. From Table 4.1 it is evident that *ftd* and *flott* allow for much larger inputs than the suffix tree implementations listed. Given enough main memory, on a 64-bit random-access machine *ftd* and *flott* allow the size of the input strings to approach 4 GB. In contrast, the input of the suffix tree implementations is limited to around 150 MB. It is worth noting that on a 32-bit random-access machine the input to *ftd* and *flott* is limited by the total addressable memory. We assume that using *Physical Address Extension (PAE)* an application can access 4 GB of memory in the flat memory model. This limits *ftd's* and *flott's* effective input on a 32-bit machine to less than 125 MB and less than 200 MB respectively. As a final point we note that on a 32-bit machine the memory savings of *flott* are only 37.5% ($12n$ bytes) due to the then smaller 32 bit wide references in *ftd's* data structures.

We close this section by remarking on more recent works that propose suffix arrays, pioneered by Manber and Myer [53], as a more space efficient alternative to suffix trees. Suffix arrays may be implemented in linear time and space [42, 41], and are a very active area of research. They have the potential to reduce the worst case memory consumption to a third of that of *flott* and *ILLI* [34, 42, 41, 39]. However, suffix arrays achieve their lower space complexity at the cost of increasing the processing time needed for sorting routines that assure optimal memory usage. How the overall performance of a suffix array based *LZ*-complexity measure compares to *flott* has not yet been investigated due to the lack of a reference open-source suffix array based *LZ*-complexity implementation.

4.3 Summary

In this chapter we evaluated the performance of the *C*-implementations of *ftd* and *flott* algorithms. In our experiments *flott* was shown to be 20% faster than *ftd* while at the same time consuming only 35.7% of *ftd's* memory on a 64-bit machine. Finally, we compared *flott* against the well established suffix tree implementations of McCreight and Kurtz. *flott's* runtime is alphabet independent and the algorithm allows for inputs close to 4 GB in size while its memory consumption is on par with that of Kurtz's most efficient suffix tree implementation. This concludes the current chapter. The next chapter will derive T-transform based information measures and show how to enhance computer security on mobile devices.

Chapter 5

T-Transform Applications

In this chapter we present some real world examples for practical uses of T-codes and the T-transform. While this thesis mainly focuses on computer security related applications in the mobile device market place, the T-transform has numerous other areas of applications some of which are listed below:

Nonlinear dynamics / entropy estimation:

Ebeling et al. compared partition-based Shannon entropy, Kolmogorov–Sinai entropy, and *T-entropy* (a deterministic T-transform derived measure estimating the average information content in a string) for nonlinear maps in [19]. Along the same line, Speidel et al. compare *LZ*-complexity, Shannon n-block entropies, and T-entropy against the Kolmogorov–Sinai entropy of the logistic map in [73]. Finally, in [81] Titchener uses T-entropy to evaluate standard industry compressors on their performance.

String similarity detection:

Yan and Speidel examined in [95] how similarity measures based on *ftd* and the *LZ* family of parsing algorithms compare against Hamming and Levenshtein distances.

Time series signal processing:

A medical application of the T-transform is given in [82] where Titchener examines time series data of electroencephalogram and electrooculogram signals with partition based T-entropies to derive sleep state indicators. Further,

Hughes et al. make use of T-entropy in a chemical engineering setting by analyzing time series signals to diagnose and control aluminium production [35].

Computer security:

Aside from the computer security applications presented in the remainder of this chapter T-code based information measures have found applications in a computer security context. Eimann et al. have successfully used T-transform based measures to detect network events such as denial of service attacks *(DDoS attacks)* [69, 20]. Further, T-codes have found applications in a secure authentication protocol proposed by Speidel et al. in [70].

Randomness tests:

A T-complexity based randomness test aimed to replace the *LZ*-complexity based test in the *NIST* test suite is proposed by Hamano et al. in [32]. Another randomness test approach based on T-codes is proposed by Speidel in [68].

Lossless data compression:

A general purpose lossless data compressor based on T-codes is proposed by Hamano et al. in [31].

Communication theory:

Recently, Speidel et al. proposed a T-code based multi-carrier error correcting code in [72].

In this thesis we add to the computer security related applications of the T-transform by providing examples of how the T-transform may be deployed to identify unusual changes in the code base of mobile device applications *(apps)* created for Google's *Android* platform [25]. While we limit our attention to Android apps, the techniques presented in this chapter are generally applicable to other mobile app platforms as well.

Before exploring the Android case study that is provided in Section 5.3 we define the normalized information distance and its T-complexity based deterministic version in Section 5.1. Further, Section 5.2 introduces the *instantaneous T-complexity rate* which allows us to qualitatively profile the spatial location of information changes between two strings.

5.1 Normalized Information Distance

Given two strings x and y we would like to have a metric allowing us to numerically quantify how close they are related to one another. Ideally such a metric, or information distance, should account for differences in string length and provide a normalized output, i.e. $d(x, y) \in [0, 1]$, for the similarity between x and y. There are many possible ways in which such a distance might be expressed; in [50] Li et al. define it in terms of Kolmogorov complexity and refer to it as the normalized information distance given by

$$d_{NID}(x, y) = \frac{\max\{K(x \mid y),\ K(y \mid x)\}}{\max\{K(x),\ K(y)\}} \tag{5.1}$$

$$\approx \frac{K(xy) - \min\{K(x),\ K(y)\}}{\max\{K(x),\ K(y)\}}\ . \tag{5.2}$$

$K(x \mid y)$ in Equation 5.1 is the conditional Kolmogorov complexity which is defined as the shortest length program that, given y as an input, produces x as its output.

The numerator in Equation 5.1 addresses the possibility of x and y containing vastly different amounts of information. That is, if x happens to be much smaller than y it will likely not require as much algorithmic effort to describe x in terms of y than the other way around. Hence, we choose to evaluate the $\max\{K(x \mid y),\ K(y \mid x)\}$ to ensure that we account for the amount of extra information by which one string dominates the other in terms of contributed information content. The denominator in Equation 5.1 makes sure that this amount is normalized by the information contained in the individually more complex string.

For $x = y$ the normalized information distance yields $d_{NID}(x, y) = 0$ indicating maximum similarity and $d_{NID}(x, y) = 1$ indicating maximum dissimilarity. Equation 5.1 is proven to be a metric, i.e. the distance is non-negative, symmetric, and satisfies the triangle inequality [49, 12]. Li et al. provide Equation 5.2 in [49] as an approximation to 5.1 which makes the assumption that $K(x \mid y) \approx K(xy) - K(y)$ and $K(xy) \approx K(yx)$.

5.1.1 Normalized Compression Distance

As already outlined in Section 2.2 we cannot compute the Kolmogorov complexity of a string, and thus, the derivation of a deterministic version of the normalized information distance is inherently problematic. Indeed, Terwijn et al. prove in [75] that Equation 5.1 cannot be approximated from above or below to any computable precision. Nevertheless, the computable normalized compression distance defined as,

$$d_{NCD}(x,y) = \frac{Z(xy) - \min\{Z(x),\ Z(y)\}}{\max\{Z(x),\ Z(y)\}}, \tag{5.3}$$

can yield good results in practice [49, 11, 12]. In Equation 5.3 the term $Z(x)$ denotes the compressed size of the string x. As already mentioned in Section 2.3, the quality of the normalized compression distance is ultimately tied to window, block, or dictionary size of the compressor used [8, 88].

5.1.2 Normalized T-Complexity Distance

Similarly to the normalized compression distance in the previous section we define the normalized T-complexity distance as

$$d_{NTC}(x,y) = \frac{\max\{C_T(x\,|\,y),\ C_T(y\,|\,x)\}}{\max\{C_T(x),\ C_T(y)\}}\ . \tag{5.4}$$

$C_T(x\,|\,y)$ is the *conditional T-complexity* which we define as the additional decomposition effort on x required after application of all copy factor/copy pattern combinations encountered during the decomposition of y.

The normalized T-complexity distance is symmetric, but we do not require it to satisfy the triangle inequality, and thus, we do not claim for $d_{NTC}(x,y)$ to be a metric. The normalized T-complexity distance may be defined in a computationally more efficient manner as,

$$d_{NTC}(x,y) = \frac{C_T(xy) - \min\{C_T(x),\ C_T(y)\}}{\max\{C_T(x),\ C_T(y)\}}\ , \tag{5.5}$$

which assumes that $C_T(x\,|\,y) \approx C_T(xy) - C_T(y)$ and $C_T(xy) \approx C_T(yx)$. Even though Equation 5.5 is computationally easier to evaluate, all distances presented in this thesis have been computed using Equation 5.4. Example 5.1.1 illustrates how the conditional T-complexity required by Equation 5.4 is evaluated in practice.

Example 5.1.1 (Conditional T-Complexity, ASCII strings). Let $x, y \in \mathbf{S}^+$ with $x =$ `'Hi␣Grandma'` and $y =$ `'Hi␣Granny'`. Then the conditional T-complexity $C_T(x \,|\, y)$ is given as the additional decomposition effort on x after application of all "knowledge" gained from the decomposition of y.

To compute $C_T(x \,|\, y)$ we define a *stop symbol* **$** such that **$** $\notin$ **S**. We then form and decompose the string concatenation x**$**$y \in \dot{\mathbf{S}}^+$, where $\dot{\mathbf{S}} = \{\mathbf{S}, \mathbf{\$}\}$. The T-transform algorithm decomposes x**$**$y$ in 11 steps. A list of copy patterns along with their respective copy factors is given in Table 5.1. The stop symbol between the strings x and y prevents the creation of a copy pattern containing information from both x and y.

			p_i: copy pattern		
	i: **level**	k_i: **copy factor**	**offset**	**length**	**value**
$C_T(y) = 8.58$	1	1	20	1	`y`
	2	2	19	1	`n`
	3	1	17	1	`a`
	4	1	16	1	`r`
	5	1	15	1	`G`
	6	1	14	1	`␣`
	7	1	13	1	`i`
	8	1	12	1	`H`
	9	1	10	2	`a`**`$`**
$C_T(x \,|\, y) = 3.0$	10	1	9	1	`m`
	11	1	1	8	`Hi␣Grand`

Table 5.1: T-transform of string x**$**$y =$ `'Hi␣Grandma`**`$`**`Hi␣Granny'`.

From Table 5.1 we evaluate the conditional T-complexity of x given y as the log-weighted sum of copy factors associated with copy patterns that contain information belonging to x as

$$C_T(x \,|\, y) = \sum_{i=9}^{11} \log_2(k_i + 1) = 3.0 \; .$$

Furthermore, as a by-product of the decomposition of x**$**$y$ we obtain the T-complexity of y as the log-weighted sum of copy factors associated with copy

patterns that contain information belonging to y as

$$C_T(y) = \sum_{i=1}^{8} \log_2(k_i + 1) = 8.58 \; .$$

In the same way we obtain $C_T(y \,|\, x) = 2.0$ and $C_T(x) = 9.0$ by decomposing $y\texttt{\$}x$ as shown in Table 5.2. We note that the decomposition step associated with the stop symbol in Table 5.2 is not accounted for because the information contained in this copy pattern does neither belong to x nor to y.

			p_i: **copy pattern**		
	i: **level**	k_i: **copy factor**	**offset**	**length**	**value**
$C_T(x) = 9.0$	1	1	20	1	`a`
	2	1	19	1	`m`
	3	1	18	1	`d`
	4	1	16	2	`an`
	5	1	15	1	`r`
	6	1	14	1	`G`
	7	1	13	1	`␣`
	8	1	12	1	`i`
	9	1	11	1	`H`
	10	1	10	1	**`$`**
$C_T(y \,|\, x) = 2.0$	11	1	9	1	`y`
	12	1	1	8	`Hi␣Grann`

Table 5.2: T-transform of string $y\texttt{\$}x =$ `'Hi␣Granny`**`$`**`Hi␣Grandma'`.

We conclude this example by using Equation 5.4 to compute the normalized T-complexity distance between x and y as

$$d_{NTC}(x, y) = \frac{\max\{C_T(x \,|\, y),\, C_T(y \,|\, x)\}}{\max\{C_T(x),\, C_T(y)\}} = \frac{3.0}{9.0} = 0.34 \; .$$

5.2 Instantaneous T-Complexity Rate

The normalized T-complexity distance, can be viewed as a *global* measure of similarity between the strings x and y. A distance value $d_{NTC}(x, y) > 0$ indicates some degree of dissimilarity and implies that $\max\{C_T(x \mid y), C_T(y \mid x)\} > 0$. In two different but closely related strings x and y, we often find that both strings share a set of common substrings that make up the majority of the information contained in either string. It is of value to be able to identify the spatial location of "information changes" in either string given the respective other. The quest for such a *local* information measure leads us to the definition of the instantaneous T-complexity rate which was already briefly mentioned in Section 3.7.1.

The instantaneous T-complexity rate is a qualitative indicator of "high" and "low" complexity regions in a string. It can be used to visualize how information is distributed over the length of a string. More precisely, in each T-transform level we calculate the change in T-complexity and average this quantity over the amount by which the T-handle has increased in length. The definition of the instantaneous T-complexity rate is then given as follows,

$$\Delta C_{T_i} = \frac{C_T(p_i^{k_i} \cdots p_1^{k_1}) - C_T(p_{i-1}^{k_{i-1}} \cdots p_1^{k_1})}{k_i \times |p_i|} = \frac{\log_2(k_i + 1)}{k_i \times |p_i|} \,. \tag{5.6}$$

However, it is necessary to point out here that the use of the instantaneous T-complexity rate is not without problems. The T-complexity $C_T(x)$ of a string x does not increase linearly with string length n, but analogous to *LZ*-complexity [48] asymptotically approaches an $O(n/\log\, n)$ bound from below [71]. There have been non-trivial efforts to devise a linearized deterministic complexity measure from the T-complexity of a string in [85]. This linearized string complexity measure is also referred to as *T-information*. The derivation of T-information and the associated instantaneous T-information rate are beyond the scope of this thesis. Moreover, for the Android case study presented in the following sections, the instantaneous T-complexity rate provides qualitatively good results while requiring less computational effort than an instantaneous T-information rate based approach.

In the following we examine yet another example to illustrate the use of the instantaneous T-complexity rate as qualitative indicator for information re-use in a string.

Example 5.2.1 (Instantaneous T-Complexity Rate, ASCII string). Usually we are concerned with examining the instantaneous T-complexity rate of two concatenated strings only. However, for illustration purposes we consider in this example the *ASCII* string $x = uvw$, which is generated from the concatenation of the individual strings $u =$ `'Hi␣Grandma'`, $v =$ `'␣la␣la␣la␣la␣la␣la␣'`, and $w =$ `'Hi␣Granny'`. Evidently the strings u and w are similar to one another as they share a common subsequence. The string v, also referred to as *tandem repeat* in molecular biology, is longer than u and v; however, it contains a periodic pattern.

The T-transform algorithm requires twelve steps to decompose x. Table 5.3 lists the copy patterns along with their respective copy factors. The reader may have noticed that tandem repeats manifest themselves in large copy factors, i.e. $k_{10} = 6$, while information that is spatially disjoined but reused within the string manifests itself in long copy patterns, i.e. $p_{12} =$ `'Hi␣Grand'`. Both, long copy patterns and large copy factors help to reduce the overall number of levels required by the T-transform to decompose the string x.

		p_i: copy pattern		
i: **level**	k_i: **copy factor**	**offset**	**length**	**value**
1	1	38	1	`y`
2	2	37	1	`n`
3	1	35	1	`a`
4	1	34	1	`r`
5	1	33	1	`G`
6	1	32	1	`␣`
7	1	31	1	`i`
8	1	30	1	`H`
9	1	28	2	`a␣`
10	6	25	3	`a␣l`
11	1	9	1	`m`
12	1	1	8	`Hi␣Grand`

Table 5.3: T-transform of string x.

Figure 5.1 shows the instantaneous T-complexity rate of the string x plotted over the strings length. In the plot the tandem repeat $p_{10} =$ `'a␣l'` with period 6 along with the area of shared information $p_{12} =$ `'Hi␣Grand'` stand out by causing a drop in the instantaneous T-complexity rate.

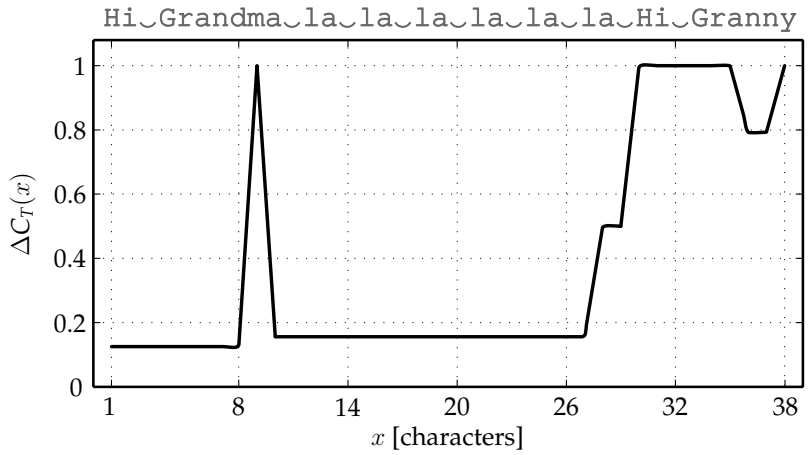


Figure 5.1: Instantaneous T-complexity rate of string x.

This concludes Example 5.2.1. In the next section we devote our attention to a real world case study in which the normalized T-complexity distance and instantaneous T-complexity rate are applied to Android applications for mobile devices.

5.3 Android Market Applications

The Android platform is an open-source operating system running *Linux* at its core and provides a software development kit for mobile *Java* apps. Due to its openness, the Android platform was able to rapidly gain mobile device market share. Not long after their introduction, mobile devices have been targeted by malware [44, 36, 40]. The detection of malware within app stores is a non-trivial task as practically anyone can offer applications for download resulting in modern app stores that hold several hundreds of thousands of apps [4]. The discovery of malware within app stores has become more frequent [52, 30, 6] and with the growing size of app stores the detection problem has become a daunting task. In this thesis we will provide some real world examples that show how the linear time and space *flott* algorithm may provide a fast and scaleable solution to detect unusual changes in the app code from one release to the next.

5.3.1 Preprocessing

As already mentioned, the analysis in the following case study focuses on Android application binaries. From one binary release to the next the registers used

in the compiled Java binaries may change and pose a source of noise. We would like to eliminate this source of noise for the purpose of calculating the normalized T-complexity distance between releases. To do so we developed a customized version of the open-source Android disassembler *baksmali* [23]. Our modified *baksmali* version strips all registers from the disassembled Java byte code while at the same time retaining machine language instructions *(opcodes)* and constant memory assignments such as numeric and string constants. However, we do not retain class and method names which are recorded verbosely in the output of the official *baksmali* distribution. In our customized *baksmali* version we replace class and method names by a 4 byte wide hash value. An in such a way modified *baksmali* disassembler produces an output file that is roughly the same size as the original Java binary.

5.3.2 Global App Evolution Tracking

In this section we use the normalized T-complexity distance to extract information from a history of binary releases of an Android application. In particular, in this case study we explore the Android app *"foursquared"* [47], a client for the popular social networking platform *Foursquare* [21]. The development of the *foursquared* app was started in early 2009 by Joe LaPenna as an open-source application for Android devices, and its development was continued by Mark Wyszomierski among others until late 2010. As shown in Figure 5.2 we collected 41 binary releases from

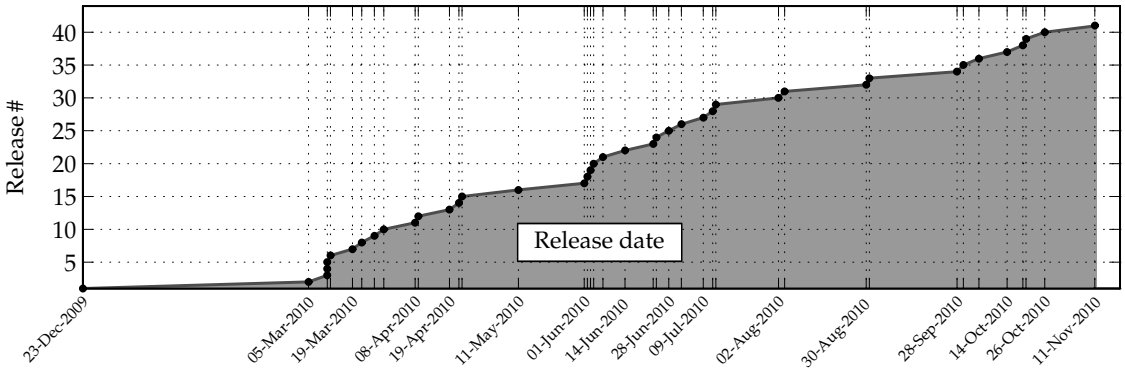


Figure 5.2: Timeline of collected binary *foursquared* releases.

the project over a one year period from December 2009 to November 2010 at which time the project was abandoned by LaPenna and Wyszomierski who now work for

Google and Foursquare, respectively. We note that in Figure 5.2 the time between *foursquared* release dates rarely exceeds one month.

To get a sense of how the development process of the *foursquared* app progressed we calculate the normalized T-complexity distance matrix which is plotted in Figure 5.3. The plot shows the evolutionary distance of every *foursquared* release to every other release.

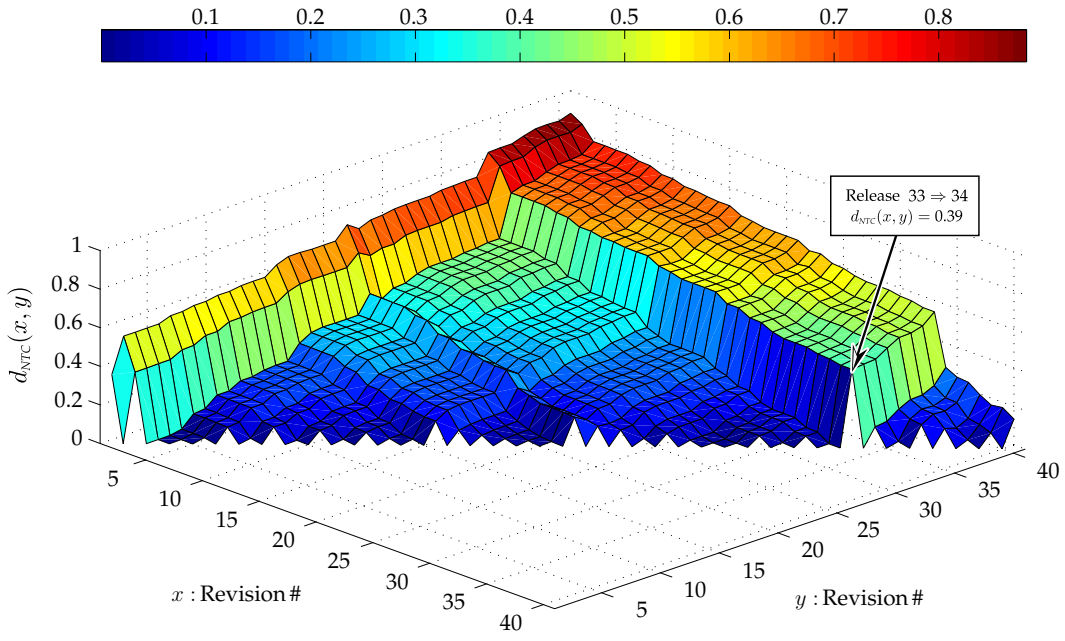


Figure 5.3: Normalized T-complexity distance matrix of *foursquared* app releases.

Since the normalized T-complexity distance is symmetric, i.e. $d_{NTC}(x, y) = d_{NTC}(y, x)$, it is sufficient to compute only the upper or lower half of the distance matrix. The plot clearly shows that the further apart the releases are with respect to their release date the larger is their normalized T-complexity distance which is what we expect. The diagonal elements in Figure 5.3 represent the distances between consecutive releases. In general this distance is fairly small, i.e. they represent bug fixes and minor modifications to the app. However, a notable exception is the transition from release # 33 (August 31, 2010) to release #34 (September 28, 2010). The distance between these two releases is recorded as $d_{NTC}(\#33, \#34) = 0.39$ which is an unusually high value when compared to the distances between the immediately preceding release pairs. Therefore we will examine these two releases in more detail in the next section.

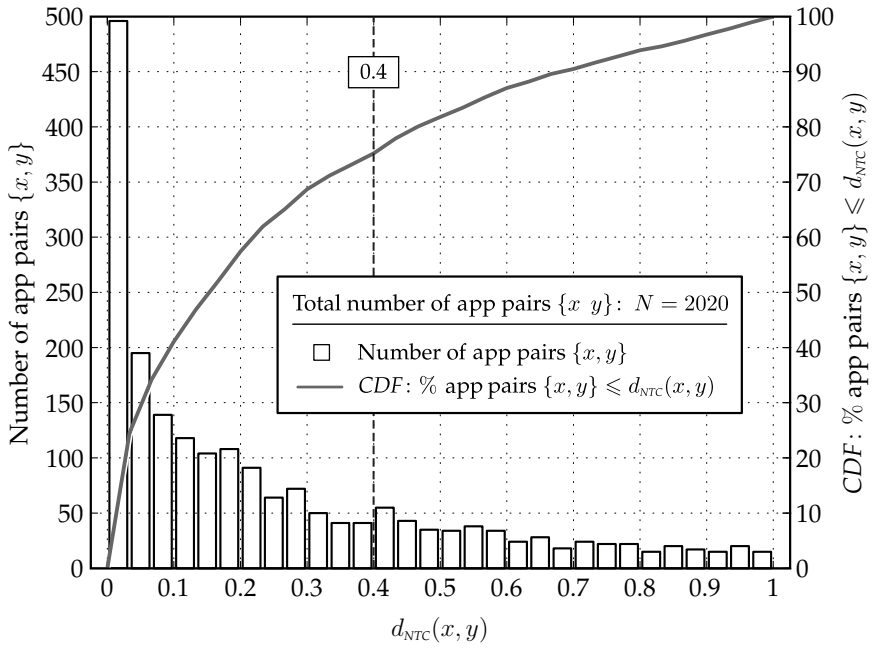


Figure 5.4: Normalized T-complexity distance of 2,020 consecutive releases from top 50 Android Market apps.

First though we would like to answer the question if the distance profile observed for consecutive *foursquared* releases is representative for the development cycle of other Android applications. We try to answer this question by considering a larger dataset which was obtained from the authors of [7]. The Android app corpus used in [7] is a diverse collection composed of 1,100 apps harvested from the top 50 free Android applications in 22 different categories of the "Android Market" (Google's controlled app market). The apps were collected over an eight month period in 2009. Each of the 1,100 apps were checked for updates on a monthly basis, and if a new app version became available it was archived together with all its previous versions. Thus, for each of the top 50 apps a history of at most eight consecutive releases is available. The apps were collected from a wide range of app categories and in this thesis we make the assumption that they are representative for a general app store collection. The data set contains 2,020 apps that published a new release within a one month period. Figure 5.4 shows their normalized T-complexity distance distribution. The majority, i.e. 50%, of distances are below 0.17 suggesting that over a month long development cycle the information content between releases is not likely to change significantly. However, as already

mentioned previously, the normalized T-complexity distance between *foursquared* releases #33 and #34 is 0.39. This means that 75% of the consecutive releases in Figure 5.4 have a smaller distance, which prompts a more detailed investigation of the *foursquared* releases #33 and #34 in the following section.

5.3.3 Local App Evolution Tracking

In this section we show how the instantaneous T-complexity rate can be used to identify the spatial location of changes introduced from one binary app release to the next. We introduced the instantaneous T-complexity rate in Section 5.2 and saw that information re-use does manifest itself in a drop of the instantaneous T-complexity rate. Thus, the T-transform may serve as a tool to locate regions of new and shared information between two apps x and y.

We obtained Figure 5.5 (a) by concatenating the *foursquared* releases #34 and #33 into the string xy and plotting its instantaneous T-complexity rate. The portions of the instantaneous T-complexity rate belonging to the apps x and y are shaded in grey and black respectively. Further, we indicated the portions in xy belonging to the individual apps x and y by separate length scales.

For comparison Figure 5.5 (b) shows the plot of the instantaneous T-complexity rate of releases #30 and #33. Release #30 was published about two months before release #33 and the normalized T-complexity distance between them was determined as 0.042. Since the T-transform algorithm identifies copy patterns *right-to-left* it means that once the algorithm generates copy patterns from the information contained in release #34 (#30) all the information in release #33 has been seen and similarities in information will be reflected in long copy patterns. In Figure 5.5 the portions of the instantaneous T-complexity rate shaded in grey may also be viewed as the *conditional instantaneous T-complexity rate* of releases #30 and #34 given all the information contained in release #33. From Figure 5.5 (a) we can see that a substantial amount of new information seems to have been added to release #34 while Figure 5.5 (b) suggests that the changes between releases #30 and #33 are fairly minor.

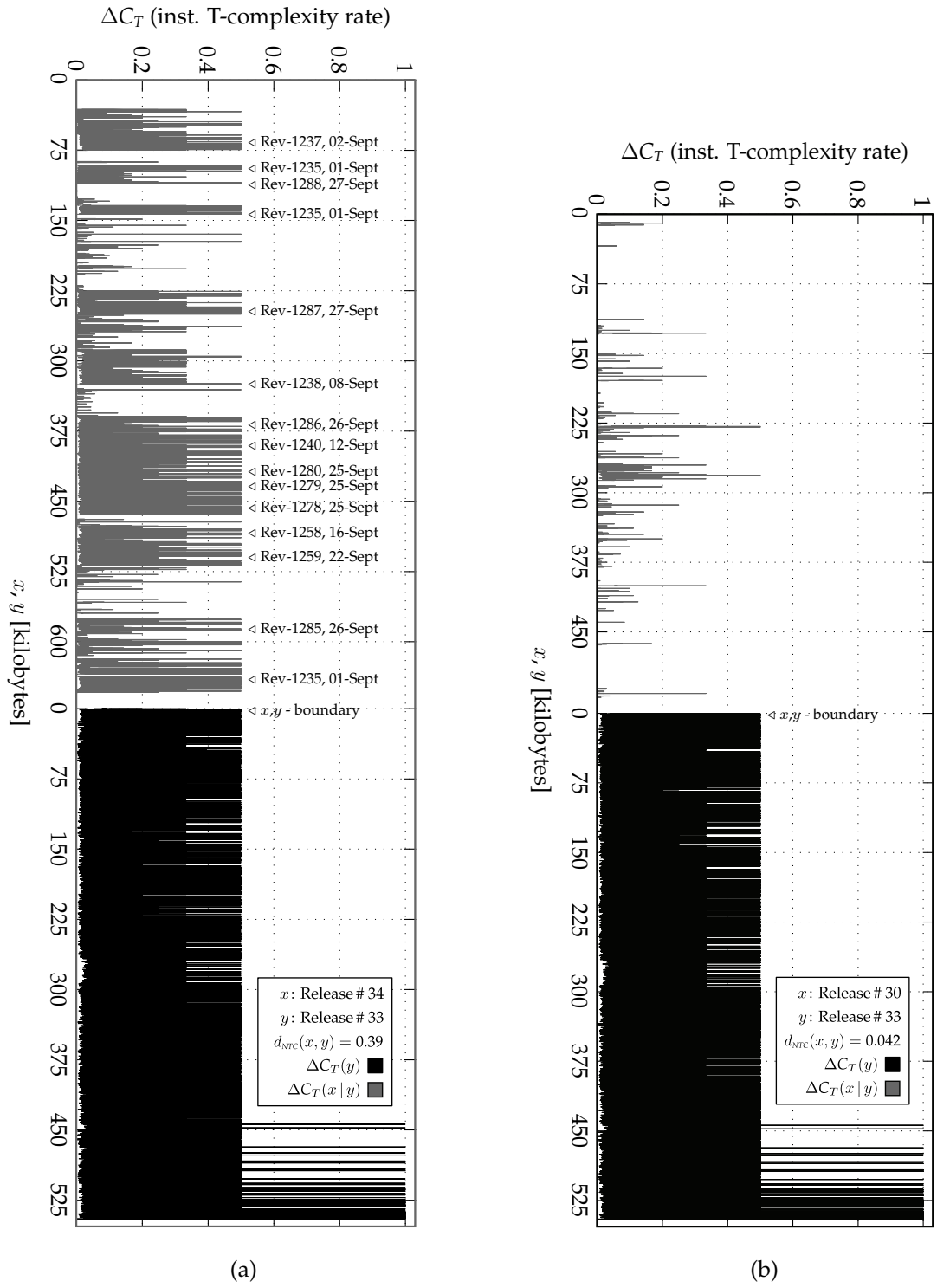


Figure 5.5: Instantaneous T-complexity rate of *foursquared*: (a) releases #33,#34 and (b) releases #33,#30.

	Mercurial Repository *foursquared*		
Date[†]	**Revision**	**Hash**	**Comment**[‡]
01-Sept	1235	`0D6494BC0548`	First pass at adding in new tips support. This implementation gets us the basic display of nearby tips, and the redesigned tip items. The layouts don't have the real artwork in them, just rough placeholders.
02-Sept	1237	`A40E682E0D58`	Rough pass at replacing xml with json. Tried to retain the same parser structure as joe implemented with xml, so no changes required outside of the parser package (besides a few hook up changes).
08-Sept	1238	`C8A7D1F084CB`	Cleaning up presentation of tips tab. Added a segmented button control, which is curiously similar to a skinned tab control. Not sure if we'll end up using it, leaving it in for now.
12-Sept	1240	`D299D373F368`	Adding rough To-do activity as the last top-level tab. Started reformatting the listview row item styles.
16-Sept	1258	`DDC0C15DA06B`	Added tip/todo addition back to venue activity.
22-Sept	1259	`AA6871ED5977`	Almost finished with venue activity redesign. It is a bit complex because changes to tips and todos is reflected in venue activites [sic]. Modifications are passed back as intent extras on activity completions. We may want to switch to a global cache of venues to make this simpler.
25-Sept	1278	`83FE5BDDD7AC`	Added new friends activity which shows friends in common along with all friends.
25-Sept	1279	`60BDCAB9B00B`	Getting special friends/followers friends activity in-place.
25-Sept	1280	`9E1CCB1BC322`	Added support for followers/friends activity for logged-in user. This will replace old activity method.
26-Sept	1285	`B85C7835574C`	Added friend button in user activities so users can accept or make friend requests.
26-Sept	1286	`E9B6CFF7646A`	Added new activity to let user turn on/off for a friend.
27-Sept	1287	`6BC99CDF9B80`	Moving user photo set out of user details activity. Fixed some ui quirks.
27-Sept	1288	`4E1D67BD2818`	Setting new user photo implemented, moved initial json parsing work into JSONUtils class so it could be shared with photo upload code, we'll probably want to clean this up further.

[†] Month of September 2010
[‡] Comments as they appear in [47]

Table 5.4: Comments in *foursquared* repository for release #33 to #34.

If we try to use traditional file comparison utilities such as *diffutils* [22] to achieve comparable results it requires $O(n^2)$ time, where n is the number of logical blocks that individually need to be compared against one another. Traditionally, *diffutils* operate on source code where the logical blocks are individual source files, thus, a *diffutils* based approach would have to be adapted to the binary domain. Naturally, the quality of such an approach largely depends on the number and size of blocks in which the binaries are divided.

In a computer security context with every new app release there is the concern that malicious code might have been injected into an application [44]. Even though the instantaneous T-complexity rate does not tell us anything about the functionality of newly added or modified code sections, it allows us to quickly flag code regions that will likely require to be monitored for their behaviour when executed.

It is generally possible to compare new or heavily modified code sections against a data base of known malicious code exemplars. In particular, the normalized T-complexity distance may be used to classify these flagged code sections as samples of known malcode.

For the *foursquared* app considered in this section we are in the fortunate position to have a source code repository [47] at our disposal. The repository provides us with a data set of detailed comments explaining which author committed which code changes to the repository. A close examination of the comments and source code changes made over the two month period that led from release #30 to release #33 revealed that code changes made were mainly bug fixes and minor modifications of cosmetic nature to the user interface. As already suspected, and confirmed by the comments made in the repository, the code changes made over the transition period from release #33 to #34 are quite substantial.

In Table 5.4 we list the comments belonging to code revisions responsible for any large code modification or newly added functionality. The spatial locations that were affected by these changes are also marked with arrows and the appropriate revision number label in Figure 5.5 (a). Table 5.4 confirms that the month of September appears to have been a very active phase in the app's development during which part of the application was rewritten and a large amount of new functionality was added. Furthermore, the repository comments allowed us to quickly verify if the changes advertised in the comments were actually implemented in the new code base, providing us with evidence that changes made were

extensive, but legitimate and not of a malicious nature.

In the following we illustrate an example in which the T-transform was used to identify malicious code of the *Droiddream trojan* family [5, 24] in a repacked version of the *Magic Hypnotic Spiral*app which was obtained from [59]. The normalized T-complexity distance between benign and infected *Magic Hypnotic Spiral* app is evaluated to 0.7. Only 10% of the consecutive releases in Figure 5.4 have a higher distance which indicates a very unusual change in information content. The instantaneous T-complexity rate of benign and infected version is plotted in Figure 5.6. The infected *Magic Hypnotic Spiral* app code is indicated in grey while

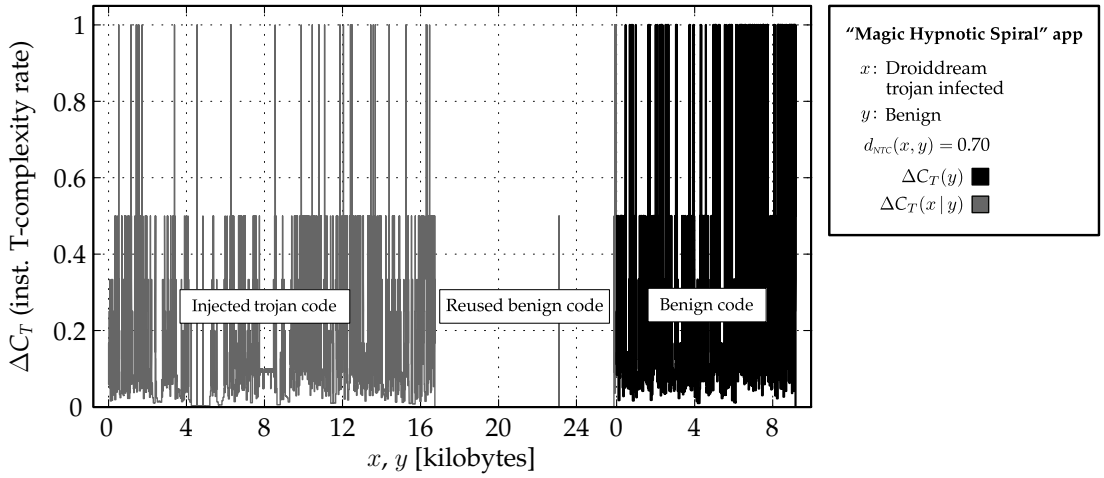


Figure 5.6: Malicious code injection into "Magic Hypnotic Spiral" app.

the benign app code is plotted in black. A benign *Magic Hypnotic Spiral* code portion that was hijacked by the trojan is clearly identifiable as a drop in the instantaneous T-complexity rate. However, the injected *Droiddream trojan* can easily be distinguished in the rate profile as new information. Calculating the normalized T-complexity distance of the new code portion in the *Magic Hypnotic Spiral* app with a *Droiddream trojan* exemplar from an annotated database of known malcode yields a distance of 0.086 which supports a reasonable presumption that the injected code is malicious.

5.3.4 Limitations and Shortcomings

In this section we point out some limitations and shortcomings to the use of the above T-transform based techniques for Android app store security. We first note

that a drop in instantaneous T-complexity rate does not differentiate between tandem repeats and information re-use. Thus, a low instantaneous T-complexity rate does not always indicate information re-use across app instances but may also indicate repetitive redundancy within the individual app. In practice this rarely becomes an issue and might be resolved by superimposing the copy factors as an additional signal on top of the rate plot.

As already mentioned in Section 5.2 the T-complexity of a string does not rise linearly with string length. Thus, for very small strings and strings of vastly different individual lengths the results for the instantaneous T-complexity rate and normalized T-complexity distance may be inaccurate, and a linearized complexity measure such as T-information [85] should rather be used for distance and instantaneous rate measurements.

Further, malicious code can be added to an app incrementally over a large number of releases. This would most likely not affect the normalized T-complexity distance of consecutive releases by much and allow malicious functionality injected into an app to go unnoticed.

Malicious code may also be injected into a binary not as a consecutive block of code but spread out in small pieces over a large area of benign code and linked back together via "jump" instructions during execution. Malicious code obfuscated in such a way would be harder to detect by simply looking at the instantaneous T-complexity rate. However, the normalized T-complexity distance may still be a good indicator since the malicious code spliced into the benign code prevents long copy patterns of benign information from forming. This in turn results in increased decomposition effort and as a consequence in a higher normalized T-complexity distance.

Finally, the normalized T-complexity distance will fail when trying to match encrypted or packed variants of malicious code samples for classification. The normalized T-complexity distance, and instantaneous T-complexity rate for that matter, can only operate on unencrypted and unpacked information. However, the detection and information extraction of packed or encrypted malware is often possible, see for example [62, 54, 38, 18, 28]. Execution traces of apps are a possible avenue to obtain unencrypted and unpacked information, with the caveat that trigger-based malicious behaviour, i.e. "time-bombs" [46], may not be captured.

5.4 Summary

This chapter began by introducing the normalized T-complexity distance as a measure of the global information distance between two strings. We then introduced the instantaneous T-complexity rate as a local measure of information change allowing us to identify the spatial location of new or modified information between two strings. Subsequently, the change history of the open-source Android app *foursquared* was studied in detail via its normalized T-complexity distance matrix and selected instantaneous T-complexity rate profiles. We went on to demonstrate how T-transform techniques can be used to detect and locate malicious code in the *Droiddream trojan* infected *Magic Hypnotic Spiral* app. Finally, this chapter closed by pointing out some of limitations and shortcomings of the techniques presented. The following chapter concludes this thesis by highlighting contributions made and providing areas of future work.

Chapter 6

Conclusion

6.1 Contributions

In this thesis we provided the to-date fastest and most memory efficient linear time and space implementation of the T-transform. We refer to the algorithm as *flott (Fast Low Memory T-Transform)* and provide an open-source *C*-implementation under the *Apache License 2.0* [76] in [60]. The *flott* implementation is 20% faster than the previous *ftd (Fast T-Decomposition)* implementation. In addition, the *flott* algorithm uses 64.3% less memory than the *ftd* algorithm on a 64-bit random-access machine. We showed how the T-transform may be used to compute the deterministic string complexity measure T-complexity which is in many aspects similar to the *LZ*-complexity introduced in Section 2.3.1. We continued by deriving two T-complexity based information measures: the *normalized T-complexity distance* and the *instantaneous T-complexity rate*. The normalized T-complexity distance is a globally operating information measure measuring the similarity or relatedness of two strings. The instantaneous T-complexity rate is a locally operating information measure which allows to determine the spatial location of shared, added, or modified information content between two files.

Previous work defined the normalized compression distance as outlined in Section 2.3 and 5.1.1. We saw that the normalized compression distance differs from the normalized T-complexity distance in that it relies on industry standard compressors to estimate string complexity. The performance of the normalized compression distance largely depends on the implementation of the chosen compressor. Compressors operating on blocks or a sliding window produce inferior

results when compared to compressors with unrestricted dictionaries.

Moreover, a *LZ76* based complexity measure may be implemented in linear time and space using suffix trees which puts such an implementation on par with a T-transform based approach. However, unlike the popular suffix tree implementations cited in this thesis the *flott* algorithm has an alphabet independent runtime by default and does not employ collision prone hash tables.

Finally, we outlined a possible application of the normalized T-complexity distance and instantaneous T-complexity rate to aid in the fight against malicious code targeting Android mobile devices. For this purpose we developed a customized decompiler for Android binaries which in combination with the T-transform based information measures allows the fast visual inspection of code modifications across a set of Android app releases.

6.2 Future Work

The applications of T-transform based data analysis techniques are plentiful. Applications other than the one presented in this thesis most certainly exist – many of them likely unknown at this point. By publishing the *flott* source code under the Apache License 2.0 [76] we allow the distribution and modification of *flott* under very unrestrictive terms. In particular, the Apache License 2.0 allows the royalty-free commercial use of the code in closed source software projects. We hope that this will help in more widespread use of a fast and powerful algorithm inspiring a lot of new and exciting future applications.

In the following sections we provide suggestions for areas of future work, not all of these areas were introduced by this thesis.

6.2.1 Algorithm Improvements

Node Compression

When considering runtime only, the *flott* algorithm in its current form may well be the most efficient sequential implementation. However, it is possible to reduce *flott's* memory consumption even further, by compressing the integer fields that connect match and token list elements. Without going into much detail we sketch the ideas of the approach briefly in the following. For the token list we may store

the distance of successor and predecessor instead of memory slot offsets. Initially these distances are small and we may store these distances in less space. As the T-handle grows so do the distances of neighbouring tokens while at the same time the likelihood that a match list occupies more than one token goes down. Depending on the input data there is a high probability that we can store the majority of tokens in much less than four integers. Whenever we can't fit a token inside a "short" list node we just link to a "long" node at its place. This approach may lower the average memory consumption to $14n$ to $16n$ bytes.

Parallel Processing

The *flott* algorithm may also make use of parallel processing to further reduce the overall runtime. As discussed in detail in [27] T-codes are *self-synchronizing codes*. The self-synchronizing property makes it theoretically possible to split the input of the T-transform algorithm into slightly overlapping blocks and decompose these blocks in parallel according to the copy patterns and copy factors generated in the rightmost block. With a high probability these blocks can be merged at some point along certain self-synchronizing pattern boundaries of adjacent blocks.

6.2.2 Hardware Implementation

The *flott* algorithm may be implemented directly in hardware. For an input of maximum size n a token may be implemented in $5\,log_2\,n$ bits resulting in an overall memory requirement of $5n\,log_2\,n$ bits. Using bit packing techniques one may also implement the *flott* algorithm in $5n\,log_2\,n$ bits of memory on the random-access machine. However, the performance of the algorithm will suffer due to the effort required for bit masking.

6.2.3 Android Market Applications

A useful extension of the Android Market case study provided in Section 5.3 would be the integration of machine learning techniques such as hierarchical clustering, support vector machines, and neural networks to automatically classify app releases as benign or malicious.

6.2.4 Suffix Arrays

It was briefly mentioned in Section 4.2 that suffix arrays provide a very space efficient data structure able to replace suffix trees. Due to scope restrictions and the lack of a reference open-source suffix array *LZ*-complexity implementation no data about how the *flott* algorithm compares to a possible suffix array based implementation is available. Further research and the development of a reference implementation is necessary in order to contrast both approaches in terms of their respective time and space demands.

6.2.5 Bioinformatics

As a string processing algorithm it is not surprising that the *flott* algorithm has applications in bioinformatics. Among possible applications, the T-transform provides a fast and efficient way to compare whole genomes. Other applications in comparative genomics include the construction of phylogenetic trees and identification of gene and genome duplication. Lastly, the instantaneous T-complexity rate can serve as a powerful tool to identify (tandem) repeats in DNA sequences.

Bibliography

[1] M. Abu Rajab, J. Zarfoss, F. Monrose, and A. Terzis. A multifaceted approach to understanding the botnet phenomenon. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, IMC '06, pages 41 – 52, New York, NY, USA, 2006. ACM.

[2] A. V. Aho and J. E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1974.

[3] Apple. iOS 5. http://www.apple.com/ios/, Feb. 2012. Last accessed Feb. 2012.

[4] Apple. The App Store. http://www.apple.com/iphone/built-in-apps/app-store.html, Feb. 2012. Last accessed: Feb. 2012.

[5] M. Balanza, K. Alintanahin, O. Abendan, J. Dizon, and B. Caraig. Droiddreamlight lurks behind legitimate android apps. In *Malicious and Unwanted Software (MALWARE), 2011 6th International Conference on*, pages 73 – 78, Oct. 2011.

[6] M. Ballano. Android.bgserv found on fake google security patch. http://www.symantec.com/connect/blogs/androidbgserv-found-fake-google-security-patch, Mar. 2011. Last accessed: Feb. 2011.

[7] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 73 – 84, New York, NY, USA, 2010. ACM.

[8] M. Cebrián, M. Alfonseca, and A. Ortega. Common pitfalls using normalized compression distance: what to watch out for in a compressor. *Communications in Information and Systems*, 5:367 – 384, 2005.

[9] G. J. Chaitin. A theory of program size formally identical to information theory. *Journal of the Association for Computing Machinery*, 22(3):329 – 40, 1975.

[10] R. Cilibrasi. The Complearn Toolkit. http://www.complearn.org/index.html, Oct. 2010. Last accessed: May 2011.

[11] R. Cilibrasi, P. Vitányi, and R. de Wolf. Algorithmic clustering of music based on string compression. *Computer Music Journal*, 28(4):49 – 67, 2004.

[12] R. Cilibrasi and P. M. B. Vitányi. Clustering by compression. *IEEE Transactions on Information Theory*, 51(4):1523 – 45, 2005.

[13] J. Cleary and I. Witten. Data compression using adaptive coding and partial string matching. *Communications, IEEE Transactions on*, 32(4):396 – 402, Apr. 1984.

[14] F. Cohen. Computer viruses: theory and experiments. *Comput. Secur.*, 6:22 – 35, Feb. 1987.

[15] W. W. Cohen. Enron email dataset. http://www.cs.cmu.edu/~enron/index.html, Aug. 2009. Last accessed: Jan. 2012.

[16] S. A. Cook and R. A. Reckhow. Time bounded random access machines. *Journal of Computer and System Sciences*, 7(4):354 – 375, 1973.

[17] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. John Wiley & Sons, Inc., 2nd edition, 1991.

[18] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 51–62, New York, NY, USA, 2008. ACM.

[19] W. Ebeling, R. Steuer, and M. R. Titchener. Partition-based entropies of deterministic and stochastic maps. *Stochastics & Dynamics*, 1:45, 2001.

[20] R. E. A. Eimann. *Network Event Detection with Entropy Measures*. PhD thesis, University of Auckland, 2008.

[21] Foursquare. Foursquare platform. https://foursquare.com/index.html, Feb. 2012. Last accessed: Feb. 2012.

[22] Free Software Foundation, Inc. GNU Diffutils. http://www.gnu.org/software/diffutils/index.html, May 2010. Last accessed: Feb. 2012.

[23] J. Freke. An assembler/disassembler for android's dex format. http://code.google.com/p/smali/, 2009. Last accessed: May 2011.

[24] S. Gold. 'App napping'?: Will holey handsets get caught. *Engineering Technology*, 6(8):78 – 81, 2011.

[25] Google. Android. http://www.android.com, Feb. 2012. Last accessed: Feb. 2012.

[26] A. Granados, R. Martínez, D. Camacho, and F. de Borja Rodríguez. Relevance of contextual information in compression-based text clustering. In *Intelligent Data Engineering and Automated Learning – IDEAL 2010*, volume 6283, pages 259 – 266. Springer Berlin/Heidelberg, 2010.

[27] U. Guenther. *Robust Source Coding with Generalised T-Codes*. PhD thesis, University of Auckland, 1998.

[28] F. Guo, P. Ferrie, and T.-c. Chiueh. A study of the packer problem and its solutions. In R. Lippmann, E. Kirda, and A. Trachtenberg, editors, *Recent Advances in Intrusion Detection*, volume 5230 of *Lecture Notes in Computer Science*, pages 98–115. Springer Berlin/Heidelberg, 2008.

[29] D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, New York, NY, USA, 1997.

[30] J. Hamada. New android threat gives phone a root canal. http://www.symantec.com/connect/blogs/new-android-threat-gives-phone-root-canal, Mar. 2011. Last accessed: Feb. 2012.

[31] K. Hamano and H. Yamamoto. Data compression based on a dictionary method using recursive construction of T-codes. In *Data Compression Conference, 2010. Proceedings. DCC 2010*, page 531, 2010.

[32] K. Hamano and H. Yamamoto. A randomness test based on T-complexity. *IEICE Transactions*, 93-A(7):1346–1354, 2010.

[33] R. Herken, editor. *A half-century survey on The Universal Turing Machine*, New York, NY, USA, 1988. Oxford University Press, Inc.

[34] W.-K. Hon, K. Sadakane, and W.-K. Sung. Breaking a time-and-space barrier in constructing full-text indices. In *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '03, pages 251 –, Washington, DC, USA, 2003. IEEE Computer Society.

[35] A. J. Hughes, M. R. Titchener, J. J. J. Chen, and M. P. Taylor. Pseudoresistance entropy as an approach to diagnostics and control in aluminium production. *Asia-Pacific Journal of Chemical Engineering*, 2(5):355–361, 2007.

[36] M. Hypponen. Malware goes mobile. *Scientific American*, (November 2006):70 – 77, Nov. 2006.

[37] ID Quantique. Quantis, a physical random number generator exploiting an elementary quantum optics process. http://www.idquantique.com/true-random-number-generator/products-overview.html, Jan. 2012. Last accessed: Jan. 2012.

[38] M. G. Kang, P. Poosankam, and H. Yin. Renovo: a hidden code extractor for packed executables. In *Proceedings of the 2007 ACM workshop on Recurring malcode*, WORM '07, pages 46–53, New York, NY, USA, 2007. ACM.

[39] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *J. ACM*, 53:918–936, Nov. 2006.

[40] G. Keizer. Researcher plants rogue app in apple's app store. http://www.computerworld.com/s/article/9221615/Researcher_plants_rogue_app_in_Apple_s_App_Store, Nov. 2011. Last accessed: Feb. 2012.

[41] D. Kim, J. Jo, and H. Park. A fast algorithm for constructing suffix arrays for fixed-size alphabets. In C. Ribeiro and S. Martins, editors, *Experimental and Efficient Algorithms*, volume 3059 of *Lecture Notes in Computer Science*, pages 301 – 314. Springer Berlin/Heidelberg, 2004.

[42] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3(2—4):143 – 156, 2005.

[43] A. N. Kolmogorov. Three approaches to the quantitative definition of information. *Problems of Information Transmission*, 1(1):1 – 7, 1965.

[44] D. Kravets. Android Market Apps Hit With Malware. http://www.wired.com/threatlevel/2011/03/android-malware/, Mar. 2011. Last accessed: Feb. 2012.

[45] S. Kurtz. Reducing the space requirement of suffix trees. *Software – Practice and Experience*, 29:1149 – 1171, 1999.

[46] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi. A taxonomy of computer program security flaws. *ACM Comput. Surv.*, 26:211 – 254, 1994.

[47] J. LaPenna and M. Wyszomierski et al. Google code Mercurial repository: Foursquare for Android, *foursquared*. https://code.google.com/p/foursquared/, Apr. 2009. Last accessed: May 2011.

[48] A. Lempel and J. Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, IT-22(1):75 – 81, 1976.

[49] M. Li, X. Chen, X. Li, B. Ma, and P. M. B. Vitányi. The similarity metric. *IEEE Transactions on Information Theory*, 50(12):3250 – 3264, 2004.

[50] M. Li and P. Vitányi. *An Introduction to Kolmogorov Complexity and its Applications*. Springer, 3rd edition, 2008.

[51] M. Li and P. M. B. Vitányi. Two decades of applied Kolmogorov complexity: in memoriam Andrei Nikolaevich Kolmogorov 1903 – 87. In *Structure in Complexity Theory Conference, 1988. Proceedings., Third Annual*, pages 80 – 101, June 1988.

[52] Lookout. Security Alert: Geinimi, Sophisticated New Android Trojan Found in Wild. http://blog.mylookout.com/2010/12/geinimi_trojan/, Dec. 2010. Last accessed: Feb. 2012.

[53] U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935 – 948, 1993.

[54] L. Martignoni, M. Christodorescu, and S. Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 431–441, Dec. 2007.

[55] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23:262 – 272, Apr. 1976.

[56] D. P. Mehta and S. Sahni. *Handbook of data structures and applications*. Chapman & Hall/CRC, 2005.

[57] R. Nicolescu and M. R. Titchener. Uniqueness theorems for T-codes. *Romanian Journal of Information Science and Technology*, 1:243 – 258, 1998.

[58] H. H. Otu and K. Sayood. A new sequence distance measure for phylogenetic tree construction. *Bioinformatics*, 19(16):2122 – 2130, Nov. 2003.

[59] M. Parkour. Contagio malware dump. http://contagiodump.blogspot.com/2011/03/take-sample-leave-sample-mobile-malware.html, July 2011. Last accesses: Feb. 2012.

[60] N. Rebenich and S. W. Neville. Fast Low Memory T-Transform. http://www.t-codes.org, Apr. 2012. Last accessed: Apr. 2012.

[61] M. Rodeh, V. R. Pratt, and S. Even. Linear algorithm for data compression via string matching. *J. ACM*, 28:16 – 24, January 1981.

[62] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *Computer Security Applications Conference, 2006. ACSAC '06. 22nd Annual*, pages 289–300, Dec. 2006.

[63] D. Salomon. *Data compression: the complete reference*. Springer, 2000.

[64] D. Salomon, G. Motta, and D. Bryant. *Handbook of Data Compression*. Springer, 2009.

[65] R. Sedgewick. *Algorithms in C*. Addison-Wesley series in computer science. Addison-Wesley Pub. Co., 1990.

[66] R. J. Solomonoff. Complexity based induction systems: comparisons and convergence theorems. pages 129 –, New York, NY, USA, 1977.

[67] U. Speidel. T-complexity and T-information theory – an executive summary, 2^{nd} revised version. Technical report, University of Auckland, 2006.

[68] U. Speidel. A forward-parsing randomness test based on the expected codeword length of T-codes. Technical report, University of Auckland, 2011.

[69] U. Speidel, R. Eimann, and N. Brownlee. Detecting network events via T-entropy. In *6th International Conference on Information, Communications and Signal Processing*, pages 951 – 5, Piscataway, NJ, USA, 2007.

[70] U. Speidel and T. A. Gulliver. A secure authentication system based on variable-length codes. In *2010 International Symposium On Information Theory & Its Applications (ISITA 2010)*, pages 684 – 9, Piscataway, NJ, USA, 2010.

[71] U. Speidel and T. A. Gulliver. An analytic upper bound on T-complexity. In *in IEEE International Symposium on Information Theory (ISIT2012)*, pages 1–6. MIT, Cambridge, USA, July 2012. (accepted for publication).

[72] U. Speidel, T. A. Gulliver, and T. Shongwe. Multicarrier error correction using T-codes. In *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PacRim), 2011*, pages 877–880, Aug. 2011.

[73] U. Speidel and M. R. Titchener. How well do practical information measures estimate the Shannon entropy. In *Proceedings of the 5th International Symposium on Communication Systems and Digital Signal Processing (CSNDSP) 2006*, 2006.

[74] R. E. Tarjan. *Data structures and network algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1983.

[75] S. A. Terwijn, L. Torenvliet, and P. M. B. Vitányi. Nonapproximability of the normalized information distance. *J. Comput. Syst. Sci.*, 77:738 – 742, July 2011.

[76] The Apache Software Foundation. Apache license, version 2.0. http://www.apache.org/licenses/LICENSE-2.0, Jan. 2004. Last accessed: Feb. 2012.

[77] M. R. Titchener. Digital encoding by means of new T-codes to provide improved data synchronisation and message integrity. In *IEE Proceedings E: Computers and Digital Techniques*, volume 131, pages 151 – 153, 1984.

[78] M. R. Titchener. Deterministic computation of complexity, information and entropy. In *IEEE International Symposium on Information Theory*, MIT, Boston, Aug. 1998.

[79] M. R. Titchener. A novel deterministic method for evaluating the entropy of language texts. In *3rd Conference in Information-Theoretic Approaches to Logic, Languages, and Computation (ITALLC'98), Hsi-tou, Taiwan*, 1998.

[80] M. R. Titchener. A measure of information. In *Data Compression Conference, 2000. Proceedings. DCC 2000*, pages 353–362, 2000.

[81] M. R. Titchener. Compressor performance, absolutely! In *Data Compression Conference, 2002. Proceedings. DCC 2002*, page 474, 2002.

[82] M. R. Titchener. T-entropy of EEG/EOG sensitive to sleep state. In *Symposium on Nonlinear Theory and Applications (NOLTA)*, Bologna, Italy, Sept. 2006.

[83] M. R. Titchener and W. B. Ebeling. Deterministic chaos and information theory. In *Data Compression Conference Proceedings*, pages 520 –, Snowbird, UT, United states, 2001.

[84] M. R. Titchener, T. A. Gulliver, R. Nicolescu, U. Speidel, and S. L. Deterministic complexity and entropy. Technical report, University of Auckland, 2004.

[85] M. R. Titchener, R. Nicolescu, L. Staiger, A. Gulliver, and U. Speidel. Deterministic complexity and entropy. *Fundam. Inf.*, 64:443–461, July 2004.

[86] M. R. Titchener, U. Speidel, and J. Yang. A comparison of practical information measures. Technical report, University of Auckland, 2005.

[87] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249 – 260, 1995.

[88] P. M. B. Vitányi, F. J. Balbach, R. L. Cilibrasi, and M. Li. Normalized information distance. In *Information Theory and Statistical Learning*, pages 45 – 82. Springer US, 2009.

[89] S. Wackrow and M. R. Titchener. tcalc.c, program code in C, GNU GPL. http://tcode.auckland.ac.nz/~corpus/tcalc.c, May 2005. Last accessed: Jan. 2012.

[90] T. A. Welch. High speed data compression and decompression apparatus and method. *U.S. Patent 4 558 302*, Aug. 1984.

[91] T. A. Welch. A technique for high-performance data compression. *Computer*, 17(6):8 – 19, 1984.

[92] J. Yang. *Fast String Parsing and its Application in Information and Similarity Measurement.* PhD thesis, University of Auckland, 2005.

[93] J. Yang and U. Speidel. An improved T-decomposition algorithm. In *Proceedings of the 2003 Joint Conference of the Fourth International Conference on Multimedia*, volume 3, pages 1551 – 1555, Dec. 2003.

[94] J. Yang and U. Speidel. A fast T-decomposition algorithm. *Univeral Computer Science (J.UCS)*, 11(6):1083 – 1101, June 2005.

[95] J. Yang and U. Speidel. String parsing-based similarity detection. In *IEEE ITSOC Information Theory Workshop on Coding and Complexity*, Piscataway, NJ, USA, 2005.

[96] J. Yang and U. Speidel. A T-decomposition algorithm with $O(n\ log\, n)$ time and space complexity. In *Proceedings. International Symposium on Information Theory (ISIT2005)*, pages 23 – 27, 2005.

[97] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *Information Theory, IEEE Transactions on*, 23(3):337 – 343, May 1977.

[98] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *Information Theory, IEEE Transactions on*, 24(5):530 – 536, Sept. 1978.

Appendix A

Source Code *(flott)*

The present version of this document does not incorporate a printed copy of the *flott* source code. The latest *flott* source code is available from [60]. *flott* is provided under the terms of the Apache Licence 2.0 as outlined below. A copy of the Apache Licence 2.0 may also be obtained from [76].

http://www.t-codes.org

LICENCE

Apache License
Version 2.0, January 2004
http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally

submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of

the Derivative Works; and

(d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be

NOTICE

FAST LOW MEMORY T-TRANSFORM

CONTACT INFORMATION

Niko Rebenich: niko@ece.uvic.ca
Department of Electrical and Computer Engineering,
University of Victoria, British Columbia, Canada.

Stephen Neville: sneville@ece.uvic.ca
Department of Electrical and Computer Engineering,
University of Victoria, British Columbia, Canada.

The latest version of this software is available at
http://www.t-codes.org.

ABOUT THIS SOFTWARE

This product includes software developed as part of Niko Rebenich's
M.A.Sc. thesis at the Information Security and Privacy Research Lab
at the University of Victoria, British Columbia, Canada
http://www.inspire.ece.uvic.ca.

The Fast Low Memory T-transform (flott) is used to compute the T-complexity of a string. In addition, it allows to compute instantaneous information rates and information distances between strings. Additionally to T-complexity, flott provides the information measures T-information and T-entropy as discussed in [1].

The T-transform algorithm is also known as T-decomposition algorithm and has its origin in coding theory, more precisely, the algorithm is used in the construction process of T-codes. T-codes were proposed by Mark Titchener in 1984 as prefix-free variable length codes.

In 1993 Mark Titchener first proposed the T-decomposition algorithm. A 1995 implementation (tcalc [2]) by Titchener and Wackrow executes in O(n^2 logn) time on a random-access machine with logarithmic cost measure. Subsequently, Jia Yang and Ulrich Speidel published the Fast T-decomposition (ftd) in 2005 having O(n logn) time and space complexity on a random-access machine with logarithmic cost measure [3,4]. The Fast Low Memory T-transform's main improvement over the Fast T-decomposition algorithm is a lower overall memory usage.

REFERENCES

[1] M. R. Titchener, R. Nicolescu, L. Staiger, T. A. Gulliver, and U. Speidel. Deterministic complexity and entropy. Fundam. Inf., 64:443461,

July 2004.

[2] S. Wackrow and M. R. Titchener. tcalc.c, program code in C, GNU GPL. [Online] http://tcode.auckland.ac.nz/~corpus/tcalc.c, 1993.

[3] J. Yang and U. Speidel. A T-decomposition algorithm with O(n logn) time and space complexity. In Proceedings. International Symposium on Information Theory (ISIT2005), pages 2327, 2005.

[4] J. Yang. Fast String Parsing and its Application in Information and Similarity Measurement. PhD thesis, University of Auckland, 2005.