Charles University in Prague

Faculty of Mathematics and Physics

**DOCTORAL THESIS**

Martin Senft

# Suffix Graphs and Lossless Data Compression

Department of Software and Computer Science Education

| | |
|---|---|
| Supervisor of the doctoral thesis: | doc. RNDr. Tomáš Dvořák, CSc. |
| Study programme: | Computer Science |
| Specialization: | I2 - Software Systems |

Prague 2013

I would like to thank for kind assistance and support I have received during my long doctoral studies. For his generous assistance, I would like to acknowledge my consultant prof. RNDr. Jaroslav Pokorný, CSc. who helped me to get started. Special appreciation goes to my supervisor doc. RNDr. Tomáš Dvořák, CSc. who provided me with precious advice, knowledge and expertise and patiently led me towards the final version of this doctoral thesis, and without whom it would never have been finished. And finally, I would be remiss if I did not mention my wife Lenka's love, support and patience during all those trying years of my research.

I declare that I carried out this doctoral thesis independently, and only with the cited sources, literature and other professional sources.
I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

Prague, 24th May 2013

Mgr. Martin Senft

Název práce: Sufixové grafy a bezeztrátová komprese dat

Autor: Martin Senft

Katedra: Katedra software a výuky informatiky

Vedoucí doktorské práce: doc. RNDr. Tomáš Dvořák, CSc., Katedra software a výuky informatiky

Abstrakt: Sufixový strom a příbuzné datové struktury umožňují asymptoticky optimálně řešit řadu úloh o řetězcích a jejich vlastností lze též využít k implementaci metod bezztrátové komprese dat. Cílem práce je prozkoumat možnosti opačného přístupu, tedy využití vlastností sufixových grafů k návrhu kompresních algoritmů. Práce popisuje univerzální konstrukční algoritmus pro sufixový trie, sufixový strom, DAWG a CDAWG, doprovázený analýzou simulace implicitních sufixových hran, která přináší dvě praktické alternativy k tradičnímu řešení. Protože kompresní metody vyžadují udržování textu v posuvném okně, je třeba rozebrat chování sufixových grafů v této situaci. V práci je ověřeno, že pouze sufixový strom je schopen udržovat posuvné okno v amortizovaně konstantním čase, zatímco CDAWG (podobně jako DAWG) vyžaduje čas úměrný délce okna, což řeší hypotézu Inenagy a kol. Na tomto základě je popsána třída kompresních algoritmů, založených pouze na popisu konstrukce sufixového grafu nad komprimovaným textem. Zatímco některé z algoritmů odpovídají klasickým slovníkovým či kontextovým metodám bezztrátové komprese, jiné jsou zcela originální. Výklad je doplněn pilotní implementací navržených algoritmů a experimentálním vyhodnocením na standardních datových korpusech.

Klíčová slova: sufixový strom, CDAWG, konstrukce, posuvné okno, komprese dat

Title: Suffix Graphs and Lossless Data Compression

Author: Martin Senft

Department: Department of Software and Computer Science Education

Supervisor of the doctoral thesis: doc. RNDr. Tomáš Dvořák, CSc., Department of Software and Computer Science Education

Abstract: Suffix tree and its variants are widely studied data structures that enable an efficient solution to a number of string problems, but also serve for implementation of data compression algorithms. This work explores the opposite approach: design of compression methods, based entirely on properties of suffix graphs. We describe a unified construction algorithm for suffix trie, suffix tree, DAWG and CDAWG, accompanied by analysis of implicit suffix link simulation that yields two practical alternatives. Since the compression applications require maintaining text in the sliding window, an in-depth discussion of sliding suffix graphs is needed. Filling gaps in previously published proofs, we verify that suffix tree is capable of perfect sliding in amortised constant time. On the other hand, we show that this is not the case with CDAWG, thus resolving a problem of Inenaga et al. Building on these investigations, we describe a family of data compression methods, based on a description of suffix tree construction for the string to be compressed. While some of these methods resemble classical lossless compression like PPM or dictionary techniques, others appear to be brand-new. Our algorithms are illustrated with a proof-of-concept implementation and experimental evaluation on standard data corpora.

Keywords: suffix tree, CDAWG, construction, sliding window, data compression

# Contents

# Preface

The title of this thesis brings together two concepts that were already combined before. However, while previous work used to select a lossless data compression method and then a suffix graph to implement the necessary string sorting and searching, this text reverses the roles and bases new lossless data compression methods on suffix graphs and their properties. Particularly, the properties of suffix graph construction algorithms and maintenance of suffix graphs in a sliding window are studied thoroughly. The information gained from this analysis is subsequently used to formulate and study the concept of suffix graph based data compression.

The concept of suffix graph based data compression is introduced gradually using the following five steps. Each step has its own dedicated chapter and the last step is followed by **Chapter 6: Epilogue,** which concludes this text.

**Chapter 1: Fundamentals**
reviews the basic building blocks needed in later steps

**Chapter 2: Definitions**
introduces suffix graphs and describes their basic properties

**Chapter 3: Construction**
addresses the issues of incremental suffix graph construction

**Chapter 4: Sliding**
deals with the maintenance of suffix graph for a sliding window

**Chapter 5: Compression**
finally brings the details of the use of suffix graphs and their construction algorithms in lossless data compression

These chapters summarise my research on the construction and sliding of suffix graphs that was driven by the idea of suffix graph based data compression. The text of these chapters is built around an extended and polished version of my previously published work which is glued together with a considerable amount of new text. Here is the list of my contributions, where they were published, and how they are integrated into this text:

- *On-line suffix tree construction with reduced branching*, coauthored by my supervisor Tomáš Dvořák, published in the Journal of Discrete Algorithms

[39]. Its extended version is the base of Section 3.2: Implicit Suffix Link Simulation.

- *Sliding CDAWG Perfection*, written in collaboration with my supervisor Tomáš Dvořák, presented at the 15th International Symposium on String Processing and Information Retrieval (SPIRE 2008), Melbourne, Australia, November 10–12, 2008 [40]. An improved version of this paper, which deals with all four types of suffix graphs, forms Section 4.2: Delete Oldest Symbol.

- *Suffix Tree for a Sliding Window: An Overview*, presented at the 14th Annual Conference of Doctoral Students (WDS'05), Prague, Czech Republic, June 7–10, 2005 [44]. Most of this work is concerned with edge-label maintenance and its extended version can be found in Section 4.3: Edge Labels.

- Suffix tree based data compression was the first basic version of suffix graph based data compression which forms the core of Chapter 5: Compression. The genesis and development of this method is documented in three publications:

    - *Bezztrátová komprese dat pomocí sufixových stromů (Lossless Data Compression using Suffix Trees)*, Master's Thesis [42], explains the main idea (in Czech).

    - *Suffix Tree Based Data Compression*, presented at the 31st Annual Conference on Current Trends in Theory and Practice of Informatics (SOFSEM 2005), Liptovský Ján, Slovakia, January 22–28, 2005 [43], provides a formal description of both data structures and compression algorithms.

    - *Compressed by the Suffix Tree*, presented at the Data Compression Conference (DCC 2006), Snowbird, Utah, March 28–30 2006 [41], supplements the theory with experimental results obtained on large data corpora.

# 1. Fundamentals

The path to the suffix graph based data compression starts in this chapter with a brief review of a few basic building blocks. As this path leads to the lossless data compression algorithms based on suffix graph construction, those building blocks fall mostly into the fields of Stringology, Graph Theory and Lossless Data Compression. While their description and analysis require concepts from other fields, these are used in a standard way and are not reviewed here.

## 1.1 Stringology

The first concepts to be reviewed and the most frequently used are those from Stringology. Unfortunately, there is no suitable textbook with notation that could be used directly. Thus, even the most basic concepts and notation are reviewed below. Nevertheless, most of them can be found in textbooks, like that of Smyth [47], albeit with a slightly different notation.

*String* $\mu$ is a finite sequence of *symbols* taken from a nonempty finite set $\Sigma$ called *alphabet*. The *length* of string $\mu$ is denoted by $|\mu|$ and the only string of zero length is called *empty string* and denoted by $\lambda$. Symbols $\Sigma^*$ and $\Sigma^+$ stand for the *set of all strings on alphabet* $\Sigma$ and the *set of all non-empty strings on alphabet* $\Sigma$, respectively. Note that throughout this text the lower-case latin letters ($a, b, \dots, x$) are used for individual symbols, lower-case greek letters ($\alpha, \beta, \dots, \omega$) for strings, and capital greek letters ($\Sigma$) for alphabets.

The *concatenation* is the only basic operation defined on strings. The result of its application on a pair of strings is the concatenation of sequences forming the two strings. For example, the concatenation of strings $\alpha = a_1\, a_2 \dots a_m$ and $\beta = b_1\, b_2 \dots b_n$, denoted by $\alpha\beta$, is the string $a_1\, a_2 \dots a_m\, b_1\, b_2 \dots b_n$. Sometimes, a concatenation of $k$ copies of string $\alpha$ is needed, this is called the *k-th power of* $\alpha$ and denoted by $\alpha^k$. With the help of concatenation, every string $\mu$ can be written as $\mu = \alpha\beta\gamma$, a concatenation of three possibly empty strings $\alpha, \beta$ and $\gamma$. In this situation we say that $\alpha$ is a *prefix* of $\mu$, $\beta$ is a *factor* of $\mu$ and $\gamma$ is a *suffix* of $\mu$. If any of them is not equal to $\mu$, then it is called *proper* prefix (factor, suffix) of $\mu$. The sets of all prefixes, factors and suffixes of $\mu$ are denoted by $\mathrm{Prefix}(\mu)$, $\mathrm{Factor}(\mu)$, and $\mathrm{Suffix}(\mu)$, respectively.

While the notions of prefix, factor and suffix certainly helps in analysis of string properties, it is often desirable to address only some specific symbols of a string. Hence, the $i$-th symbol of string $\mu$ is denoted by $\mu[i]$, for any $i$ satisfying $1 \le i \le |\mu|$. Similarly, the factor of $\mu$ consisting of symbols $\mu[i],\ \mu[i+1],\ \dots,\ \mu[j]$ is denoted by $\mu[i\,..\,j]$, where $0 \le j \le |\mu|$ and $1 \le i \le j+1$. If $\alpha = \mu[i\,..\,j]$, then we say that $\alpha$ has a (*right*) *occurrence* in $\mu$ at position $j$ and a *left* *occurrence* in $\mu$ at position $i$. The case when $i = j + 1$ ensures that the empty string has both left and right occurrence at every position in every string, including the empty string itself. The set of all positions of right (left) occurrences of $\alpha$ in $\mu$ is denoted by $\mathrm{Occur}^{\mathrm{R}}_{\mu}(\alpha)$ ($\mathrm{Occur}^{\mathrm{L}}_{\mu}(\alpha)$).

One possible use for the concept of occurrence is the categorisation of factors by occurrence count. String $\alpha$ is called *unique* in $\mu$ if it has exactly one occurrence in $\mu$ ($|\mathrm{Occur}^{\mathrm{R}}_{\mu}(\alpha)| = |\mathrm{Occur}^{\mathrm{L}}_{\mu}(\alpha)| = 1$). It is called *non-unique* if it occurs in $\mu$ at least twice ($|\mathrm{Occur}^{\mathrm{R}}_{\mu}(\alpha)| = |\mathrm{Occur}^{\mathrm{L}}_{\mu}(\alpha)| > 1$). Note that the empty string is never unique in a nonempty string. The set of all unique factors of $\mu$ is denoted by $\mathrm{Unique}(\mu)$. Its intersection with the set of all prefixes or the set of all suffixes yields the sets of all *unique prefixes* and *unique suffixes* of string $\mu$, denoted by $\mathrm{UniquePrefix}(\mu)$ and $\mathrm{UniqueSuffix}(\mu)$. On the other hand, the fusion of non-uniqueness and prefix or suffix leads to the set of non-unique prefixes or the set of non-unique suffixes. Longest members of those sets are denoted by $\mathrm{LNUP}(\mu)$ and $\mathrm{LNUS}(\mu)$, respectively.

While occurrence counts are useful, sometimes a more information about the occurrence surroundings is needed. If $\alpha = \mu[i\,..\,j]$ and the symbol $\mu[i-1]$ ($\mu[j+1]$) exists, then it is called the *left context* (*right context*) of this particular occurrence of $\alpha$ in $\mu$. It might be more intuitive to define contexts as the prefix preceding the occurrence and the suffix following the occurrence, but here the single symbol context is preferable. The sets of all symbols that appear as right and left contexts of occurrences of $\alpha$ in $\mu$ are denoted by $\mathrm{Context}^{\mathrm{R}}_{\mu}(\alpha)$ and $\mathrm{Context}^{\mathrm{L}}_{\mu}(\alpha)$, respectively.

The notion of occurrence contexts leads to the concept of branching. When factor $\alpha$ has at least two distinct symbols as right (left) contexts in string $\mu$, i.e. $|\mathrm{Context}^{\mathrm{R}}_{\mu}(\alpha)| > 1$, then factor $\alpha$ is called *right (left) branching* in string $\mu$. The sets of all left and right branching factors of string $\mu$ are denoted by $\mathrm{Branch}^{\mathrm{L}}(\mu)$ and $\mathrm{Branch}^{\mathrm{R}}(\mu)$, respectively.

Finally, the combination of the empty string, the right branching factors of string $\mu$ and the unique suffixes of string $\mu$ yields a very special set $\mathrm{Explicit}(\mu) = \{\lambda\} \cup \mathrm{Branch}^{\mathrm{R}}(\mu) \cup \mathrm{UniqueSuffix}(\mu)$, whose members are called *explicit* factors of $\mu$. The empty string is handled separately as it could be a member of $\mathrm{Branch}^{\mathrm{R}}(\mu)$ or $\mathrm{UniqueSuffix}(\mu)$ or none of them. The last two cases happen only if $\mu = \lambda$ or $\mu = a^k$, respectively. This is easy to overlook and make the definition of suffix graph invalid for these two cases. For all other strings $\mu$, the empty string is right branching and the definition can be changed to $\mathrm{Explicit}(\mu) = \mathrm{Branch}^{\mathrm{R}}(\mu) \,\dot\cup\, \mathrm{UniqueSuffix}(\mu)$. The $\dot\cup$ notation stands for the *union of sets with empty intersection,* that is $\mathrm{Branch}^{\mathrm{R}}(\mu) \cap \mathrm{UniqueSuffix}(\mu) = \emptyset$.

## 1.2 Graph Theory

The second to be reviewed is the field of Graph Theory. Here this text mostly adheres to the well known graph notation and terminology that can be found in textbooks [11]. Basic graph concepts like *graph, vertex, edge, path, incidence* or *cycle* are used in a standard way. The same holds for tree-specific concepts like *tree*, *root* and *leaf*. Also, the use of lower-case latin letters for vertices ($v, u, t, ...$) and edges ($e, f, ...$), and capital latin letters for graphs ($G$) and sets of edges ($E$) and vertices ($V$), is fairly standard as well. However, there are also a few differences.

All graphs used here are *connected directed acyclic graphs*. For every vertex, an *incoming edge* or an *in-edge* is any *incident* edge which has this vertex as its *terminal vertex*. Similarly, an *outgoing edge* or an *out-edge* is any *incident* edge having this vertex as its *initial vertex*. This is used to define an *in-degree* (*out-degree*) of a vertex as the number of in-edges (out-edges) this vertex has. Moreover, every vertex with out-degree larger than one is called *branching*.

Our graphs are further restricted to have exactly one vertex with no in-edges. We call this vertex *origin* in all graphs, but also use the name *root* in trees and *source* in non-tree graphs. Moreover, non-tree graphs are required to have exactly one vertex with no out-edges that we call *sink*. Subsequently, all maximal paths from the origin end in leaves in trees and in sink otherwise. Any vertex that is neither a leaf nor a sink is called *inner node* or just *node*.

To further simplify the discussion of suffix graphs, a special type of graph is used. Throughout this text, graph $G$ is a directed graph with *edges* labelled with nonempty *strings* and separate auxiliary unlabelled edges called *links*. In other words, graph $G = (V, E, L)$ consists of vertices taken from the set $V$, *labelled edges* selected from the set $E \subseteq V \times \Sigma^{+} \times V$, and links chosen from the set $L \subseteq V \times V$. To avoid confusion, when there is more then one graph to choose from, $V(G)$, $E(G)$ and $L(G)$ denotes the vertices, edges and links of graph $G$, respectively.

The fact that the second component of every edge in such graph is a nonempty string, is used to introduce several specific concepts. An *a-edge* of a vertex is its out-edge which has symbol $a$ as the first symbol of its label. Moreover, the edge labels can be concatenated along paths to form *path strings*. The paths starting at the origin are then said to *represent* their path strings in the graph. Among them, the path representing the longest string is called the *backbone*. Note that the target vertex of any path representing string can represent this string as well. However, unlike the path, the vertex may represent several different strings at the same time, when the graph is not a tree.

The correspondence between vertices and strings they represent can be used to add constraints on links in the graph. There are two common constraints in use. One requires the links to be *suffix links* , while the other one forces links to be *prefix links*. In this context, a *suffix link* (*prefix link*) is a link that leads from the initial vertex $u$ to the terminal vertex $v$, which represents the longest proper suffix (prefix) of the string represented by $u$.

The construction algorithms discussed in Chapter 3: Construction are easier to implement, if the origin has a valid suffix link. To this end, the graph is augmented with a new vertex called *bot*, which is the target of a new virtual suffix link from the origin. Moreover, several new edges leading from the bot to the origin are added, one for every symbol of the alphabet $\Sigma$. Note that these

| Component | Basic | Active | New | Active new |
|---|---|---|---|---|
| bot | △ | ▲ | ◬ | ◬ |
| root or source | ◯ | **◯** | ◌ | ◌ |
| explicit node | ○ | **○** | ◌ | ◌ |
| implicit node | ∘ | • | ∘ | ⊗ |
| leaf or sink | □ | ■ | ⬚ | ⬚ |
| implicit edge with label *c* | $-c\rightarrow$ | $-c\rightarrow$ | $\cdot\cdot c\cdot\rightarrow$ | $\cdot\cdot c\cdot\rightarrow$ |
| suffix link | ⟶ | ⟶ | ⇢ | ⇢ |

**Table 1.1.** A summary of visual language used to draw graphs. All listed components are described in Section 1.2. Active and new component versions are used for components that are currently used or created.

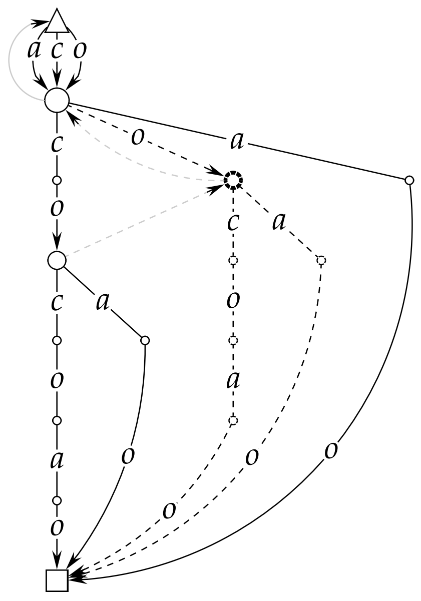


**Figure 1.1.** An example of graph drawn using symbolics described in Table 1.1.

new parts of the graph are also used by compression algorithms described in Chapter 5: Compression.

Interestingly, it is sometimes convenient to imagine a virtual *implicit node* positioned between every two symbols of every edge in the graph. Such nodes are virtually splitting every edge into several *implicit edges* with single symbol labels. The real vertices and edges are called *explicit* in this context. Using the implicit nodes and edges together with their explicit counterparts enables the definition of an *implicit path* as a path connecting explicit and implicit vertices using implicit edges. Like real paths, these implicit paths have their path strings and do represent them, if they start at the origin. The same strings are represented by target vertices of these path, both explicit and implicit.

Concepts introduced here are used throughout this text and are often shown graphically to illustrate various graph algorithms and properties. All of these graph visualisation use common components described in Table 1.1. The use of the these components is illustrated in the first example presented in Figure 1.1.

## 1.3 Lossless Data Compression

The last to be reviewed is the field of lossless data compression. In this case, there are many good textbooks [38] that cover most of the details needed here. The rest is described in relevant places or may be found in the papers cited there.

# 2. Definitions

Using the building blocks set up in Chapter 1: Fundamentals, this chapter moves one step forward on the path to the suffix graph based data compression. It introduces suffix graphs and studies their basic properties.

There exist many suffix based data structures with various interesting properties [52, 6, 7, 28, 23, 9, 47]. The most powerful and popular of them today is the suffix array [28]. However, throughout this text, the concept of suffix graph is limited to just four of them. They are the suffix trie [47], the suffix tree [52], the directed acyclic word graph (DAWG) [6] and the compact directed acyclic word graph (CDAWG) [7]. The main reason for this choice is that all of these structures are closely related to the suffix tree, which motivated the first part of the research into the suffix graph based data compression. Thanks to their similarity, all four structures can be constructed using a slightly adapted versions of a single construction algorithm [51, 22]. This will play an important role in Chapter 3: Construction and later.

In a nutshell, a suffix graph for string $\mu$ is a connected directed acyclic graph $G = (V, E, L)$, as defined in Section 1.2. Its vertices are taken from the factors of $\mu$ or sets of factors of $\mu$, edges are labelled by factors of $\mu$, and its links are suffix links. Moreover, maximal paths from the origin represent the set of all unique suffixes of $\mu$, and the set of strings represented by implicit paths is equal to the set of all factors of $\mu$. There is also at most one $a$-edge leading from any vertex, which makes searches for strings represented in the graph deterministic. All of this is illustrated in Figures 2.1 and 2.2, with two examples of each suffix graph type. More details about each suffix graph type are given below.

## 2.1 Suffix Trie

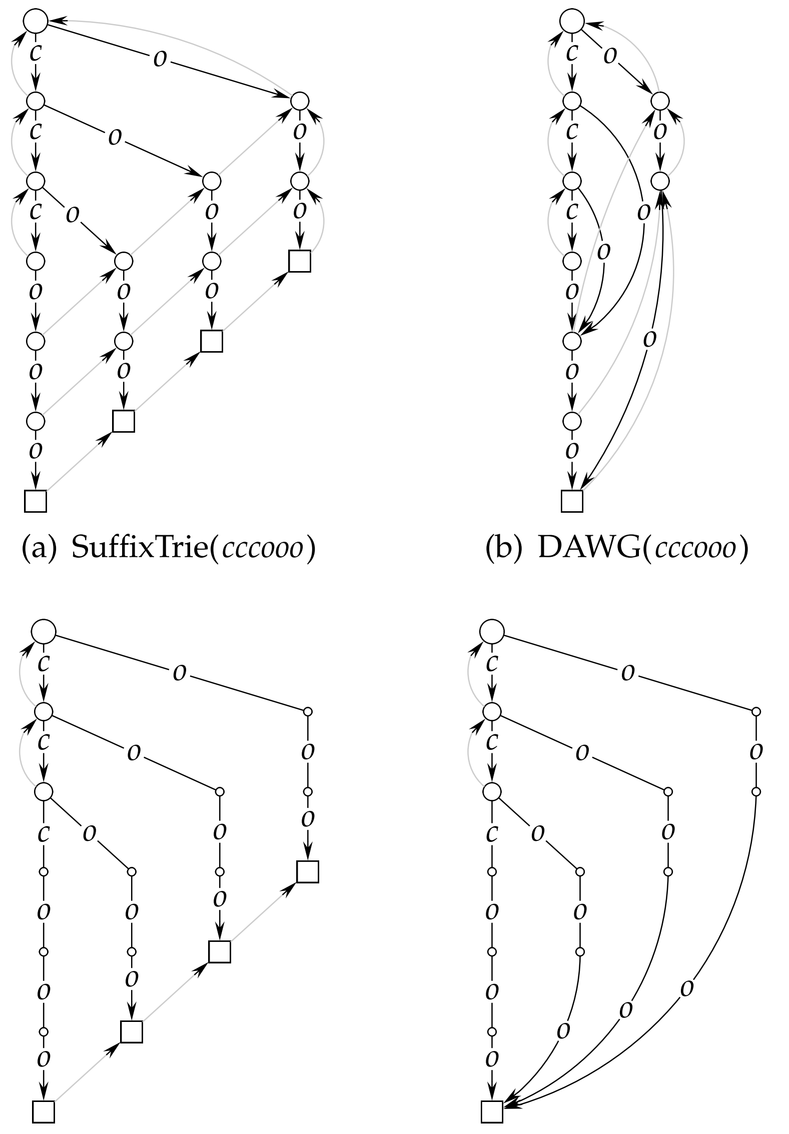
The least complex among the four suffix graph types is the suffix trie, which also happens to have the most straightforward definition [47].

**Definition (Suffix Trie)**
The *suffix trie* for a string $\mu$ is the graph $\mathrm{SuffixTrie}(\mu) = (V, E, L)$, with the following components:

$$V = \left\{ \alpha \mid \alpha \in \mathrm{Factor}(\mu) \right\},$$

$$E = \left\{ (\alpha, a, \alpha a) \mid \alpha, \alpha a \in V \wedge a \in \Sigma \right\},$$

$$L = \left\{ (a\alpha, \alpha) \mid a\alpha, \alpha \in V \wedge a \in \Sigma \right\}.$$



(a) SuffixTrie(*cccooo*) (b) DAWG(*cccooo*)

(c) SuffixTree(*cccooo*) (d) CDAWG(*cccooo*)

**Figure 2.1.** Suffix graphs for string *cccooo*. The symbolics used is described in Table 1.1.

Informally, the suffix trie for string $\mu$ is a graph with the vertex set equal to the set of all factors of string $\mu$. Each edge is labelled with a single symbol so that its terminal vertex is a concatenation of its initial vertex and the edge label. In other words, the initial vertex is the longest proper prefix of the terminal vertex. Moreover, suffix links are placed between every factor and its longest proper suffix. All of this is illustrated by suffix tries in Figures 2.1a and 2.2a.

There are several important properties of suffix trie that are worth noting. All of them are straightforward consequences of the suffix trie definition.

**Claim 2.1 (Shape of the Suffix Trie)**
The suffix trie for string $\mu$ is a directed tree that is rooted at the empty string and its leaves are unique suffixes of $\mu$.

**Claim 2.2 (Contents of the Suffix Trie)**
The set of all strings represented by paths from the root of the suffix trie for string $\mu$ is equal to the set of all factors of $\mu$.

**Claim 2.3 (Unique Branching in the Suffix Trie)**
Every vertex of the suffix trie for string $\mu$ has at most one $a$-edge for every $a$ from alphabet $\Sigma$.

**Claim 2.4 (Space Complexity of the Suffix Trie)**
The number of vertices of the suffix trie for string $\mu$ is bounded by $\Omega(|\mu|^2)$ in the worst case.

As illustrated by Figure 2.1a, suffix tries for strings like $c^k o^k$ have $(k+1)^2$ vertices.

## 2.2 Suffix Tree

While the suffix trie is easy to grasp and handle, its quadratic size is discouraging. One of the reasons why the suffix trie can be so big is that it can have a large number of inner nodes. For example, there are twelve inner nodes in the suffix trie in Figure 2.1a and eight inner nodes in the suffix trie in Figure 2.2a. However, only three of them are branching in each graph. Thus, paths that start in the root or a branching node, go through some non-branching nodes and end in another branching node or leaf could be simplified. They could be replaced by a single edge with label, which is the result of a concatenation of all labels on edges on the path. The results of application of this modification to suffix tries in Figures 2.1a and 2.2a can be seen in Figures 2.1c and 2.2c, respectively. This very common technique is called *path compression* and its application on the suffix trie yields the suffix tree.

The path compression will be introduced into the formal definition of the suffix tree using the following concept.

**Definition (Right Extension)**
Let $\mu \in \Sigma^*$ and $\alpha \in \text{Factor}(\mu)$. The *right extension* of string $\alpha$ in string $\mu$ is denoted by $\langle\alpha\rangle^{\mathrm{R}}_{\mu}$ and defined as follows.

$$\langle\alpha\rangle^{\mathrm{R}}_{\mu} = \begin{cases} \alpha & \alpha \in \text{Explicit}(\mu)\,, \\ \langle\alpha a\rangle^{\mathrm{R}}_{\mu}, \text{where } a \in \Sigma \text{ and } \alpha a \in \text{Factor}(\mu) & \text{otherwise}\,. \end{cases}$$

An alternative and more compact definition would say that the right extension of string $\alpha$ in string $\mu$, denoted by $\langle\alpha\rangle^{\mathrm{R}}_{\mu}$, is the shortest string $\alpha\beta \in \mathrm{Explicit}(\mu)$.

Nevertheless, both definitions agree that any explicit factor $\alpha$ of string $\mu$ is its own right extension in $\mu$. If it is not explicit, $\alpha$ is a non-empty factor of string $\mu$ that is neither a unique suffix nor a right branching factor. Its right extension is then the same as the right extension of string $\alpha a$, which is a one symbol longer factor of $\mu$.

Informally, the right extension of $\alpha$ in $\mu$ is the string obtained by appending symbols to $\alpha$ until an explicit factor of $\mu$ is reached. Note that as no intermediate result can be branching, the right extension has the following important property.

**Claim 2.5 (Right Extension Uniqueness)**
Every factor of string $\mu$ has exactly one right extension in string $\mu$.

The following example illustrates the definition of the right extension on string *cccooo* used in Figure 2.1.

**Example 2.6 (Right Extension)**
To see the results of the application of the right extension to factors of string *cccooo*, the explicit factor set Explicit(*cccooo*) must be known. As it is defined as $\{\lambda\} \cup \mathrm{Branch}^{\mathrm{R}}(cccooo) \cup \mathrm{UniqueSuffix}(cccooo)$, it suffices to find the composition of the last two sets.

$$\mathrm{Branch}^{\mathrm{R}}(cccooo) = \{\lambda, c, cc\}$$

$$\mathrm{UniqueSuffix}(cccooo) = \{ooo, cooo, ccooo, cccooo\}$$

The right extensions of all non-explicit factors of string *cccooo* then look as follows:

$$\langle o\rangle^{\mathrm{R}}_{cccooo} = \langle oo\rangle^{\mathrm{R}}_{cccooo} = ooo$$

$$\langle co\rangle^{\mathrm{R}}_{cccooo} = \langle coo\rangle^{\mathrm{R}}_{cccooo} = cooo$$

$$\langle cco\rangle^{\mathrm{R}}_{cccooo} = \langle ccoo\rangle^{\mathrm{R}}_{cccooo} = ccooo$$

$$\langle ccc\rangle^{\mathrm{R}}_{cccooo} = \langle ccco\rangle^{\mathrm{R}}_{cccooo} = \langle cccoo\rangle^{\mathrm{R}}_{cccooo} = cccooo$$

By modifying the definition of suffix trie using path compression through right extension we obtain the definition of suffix tree.

**Definition (Suffix Tree)**
The *suffix tree* for a string $\mu$ is a graph $\mathrm{SuffixTree}(\mu) = (V, E, L)$, which has the following components:

$$V = \left\{ \alpha \;\middle|\; \alpha \in \text{Explicit}(\mu) \right\},$$

$$E = \left\{ (\alpha, a\beta, \alpha a\beta) \;\middle|\; \alpha, \alpha a\beta \in V \wedge a \in \Sigma \wedge \beta \in \text{Factor}(\mu) \wedge \langle \alpha a \rangle^{\text{R}}_{\mu} = \alpha a\beta \right\},$$

$$L = \left\{ (a\alpha, \alpha) \;\middle|\; a\alpha, \alpha \in V \wedge a \in \Sigma \right\}.$$

Compared to the suffix trie, there are some clearly visible changes in sets of vertices and edges. The vertex set changed from all factors to just explicit factors of string $\mu$, which is also the set of all right extensions of factors of $\mu$. The right extension is also used to connect these vertices using edges. Note that this version of the suffix tree definition, introduced by Ukkonen [51], is adjusted to enable on-line construction. Originally, the suffix tree was defined for strings $\mu\$$, with a special symbol $\$$ appended to the original string $\mu$ [52]. The symbol $\$$ was selected so that is did not occur anywhere in $\mu$ and consequently made all suffixes of $\mu\$$ unique.

Like in the case of suffix trie, the definition of suffix tree is followed by a review of some basic properties. Most of them are shared between the two and the most important difference lies in space complexity.

**Claim 2.7 (Shape of the Suffix Tree)**
The suffix tree for string $\mu$ is a directed tree rooted at the empty string and its leaves are unique suffixes of $\mu$.

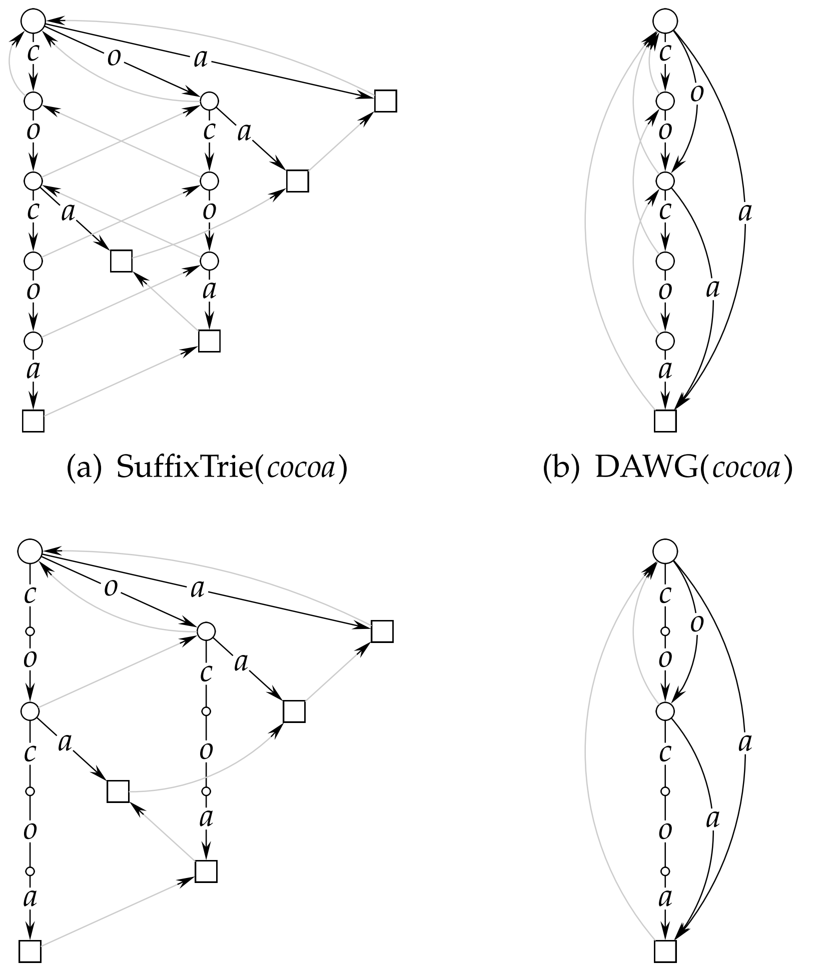
**Claim 2.8 (Contents of the Suffix Tree)**
The set of all strings represented by implicit paths from the root of the suffix tree for string $\mu$ is equal to the set of all factors of $\mu$.

**Claim 2.9 (Unique Branching in the Suffix Tree)**
Every vertex of the suffix tree for string $\mu$ has at most one $a$-edge for every $a$ from alphabet $\Sigma$.

**Claim 2.10 (Space Complexity of the Suffix Tree)**
The are at most $|\mu| - 1$ nodes, at most $|\mu|$ leaves, and at most $2|\mu| - 2$ edges in SuffixTree($\mu$) for any string $\mu$ with at least two symbols.

**Proof:** Using Claim 2.7, the number of leaves equals the number of unique suffixes of $\mu$ which cannot exceed $|\mu|$. Let $n$ and $m$ denote the numbers of nodes and edges, respectively, and assume that $n > 1$. Then each node has at least two out-edges and each vertex except the root has exactly one in-edge. Consequently, $2n \leq m \leq n - 1 + |\mu|$, which implies that $n \leq |\mu| - 1$ and $m \leq 2|\mu| - 2$ as claimed. ∎

**Claim 2.11 (Suffix Links of the Suffix Tree)**
There is a suffix link leading from every vertex of SuffixTree($\mu$), with the exception of root and possibly the shortest leaf. The root has no suffix link and the



(a) SuffixTrie(*cocoa*) (b) DAWG(*cocoa*)

(c) SuffixTree(*cocoa*) (d) CDAWG(*cocoa*)

**Figure 2.2.** Suffix graphs for string *cocoa*. The symbolics used is described in Table 1.1.

suffix link from the shortest leaf exists if and only if the longest non-unique suffix of $\mu$ is a branching factor of $\mu$. All other suffix links lead from one branching node to another branching node or from one leaf to another leaf.

**Proof:** This is a direct consequence of the definition of suffix tree and the following two properties of factors of $\mu$. First, any suffix of branching factor is also a branching factor. Second, any suffix of string $\mu$ that is longer than some unique suffix of string $\mu$ is a unique suffix of $\mu$ as well. ■

## 2.3 Directed Acyclic Word Graph

The savings introduced by the path compression are impressive. However, there is another type of redundancy, which it did not remove. As can be seen on suffix tries and suffix trees in Figures 2.1 and 2.2, there are several identical subtrees in every graph. For example, subtrees under strings *ccco*, *cco*, *co* are

| $\mathrm{Occur}^{\mathrm{R}}_{cccooo}$ | $[\,]^{\mathrm{R}}_{cccooo}$ | $(\,)^{\mathrm{R}}_{cccooo}$ |
|---|---|---|
| $\{0,1,2,3,4,5,6\}$ | $\{\lambda\}$ | $\lambda$ |
| $\{1,2,3\}$ | $\{c\}$ | $c$ |
| $\{2,3\}$ | $\{cc\}$ | $cc$ |
| $\{3\}$ | $\{ccc\}$ | $ccc$ |
| $\{4,5,6\}$ | $\{o\}$ | $o$ |
| $\{5,6\}$ | $\{oo\}$ | $oo$ |
| $\{4\}$ | $\{co,cco,ccco\}$ | $ccco$ |
| $\{5\}$ | $\{coo,ccoo,cccoo\}$ | $cccoo$ |
| $\{6\}$ | $\{ooo,cooo,ccooo,cccooo\}$ | $cccooo$ |

**Table 2.1.** This table illustrates the right end equivalence concept on string *cccooo*. The first column contains the right occurrences shared by the strings of the right end equivalence classes shown in the second column. The last column is then used to highlight the longest member of every class.

identical in the first case, and subtrees under strings *co* and *o* are identical in the second. Note that roots of these copies are connected in the deepest first order via suffix links. Ideally, all such subtrees would be replaced by a single copy and reused through the redirection of all in-edges terminating in roots of removed copies. The effect this modification has on the suffix trie can be seen in Figures 2.1b and 2.2b. This technique is called *vertex minimisation* and its application on the suffix trie yields the DAWG.

The vertex minimisation of the suffix trie relies on factorisation of the vertex set using the right end equivalence. It was originally introduced as "end-set relation" by Blumer et al. [6].

**Definition (Right End Equivalence)**
Strings $\alpha$ and $\beta$ are equivalent under the *right end equivalence* on string $\mu$, denoted by $\alpha \equiv^{\mathrm{R}}_{\mu} \beta$, if and only if $\mathrm{Occur}^{\mathrm{R}}_{\mu}(\alpha) = \mathrm{Occur}^{\mathrm{R}}_{\mu}(\beta)$. The equivalence class where strings equivalent to $\alpha$ belong under the $\equiv^{\mathrm{R}}_{\mu}$ equivalence is called the *right end equivalence class* of $\alpha$ on $\mu$ and denoted by $[\alpha]^{\mathrm{R}}_{\mu}$. Moreover, the *longest member* of this class is denoted by $(\alpha)^{\mathrm{R}}_{\mu}$, if it exists. Finally, the class containing all strings that are not factors of $\mu$ is called the *degenerated class* and denoted by $\mathrm{DC}^{\mathrm{R}}_{\mu}$.

In other words, strings $\alpha$ and $\beta$ are right end equivalent in $\mu$ if and only if, for every occurrence of $\alpha$, there is a corresponding occurrence of $\beta$ ending at the same position in $\mu$ and vice versa. The effects of this definition are illustrated in Table 2.1 and the following claim summarises its basic properties.

**Claim 2.12 (Right End Equivalence Properties [6])**
Right end equivalence has the following properties.

(i) If $\alpha$ and $\beta$ are right end equivalent, then $\alpha\gamma$ and $\beta\gamma$ are right end equivalent for every string $\gamma$.

(ii) If two factors are right end equivalent, then one is a suffix of the other.

(iii) Two factors $\alpha\beta$ and $\beta$ are right end equivalent if and only if every occurrence of $\beta$ is immediately preceded by an occurrence of $\alpha$.

(iv) A factor $\alpha$ of $\mu$ is the longest member of $[\alpha]^{\mathrm{R}}_{\mu}$ if and only if it is either a prefix of $\mu$, or it is left branching in $\mu$.

The following observation is a direct consequence of Claim 2.12.

**Claim 2.13 (Right End Equivalence Class Properties)**
Every non-degenerated class of the right end equivalence on $\mu$ has the shortest member. Moreover, all suffixes of the longest member of this class that are longer than the shortest member also belong to this class. That is

$$\forall \mu \in \Sigma^*,\ \forall \alpha \in \mathrm{Factor}(\mu),\ \exists k \geq 0 :\ [\alpha]^{\mathrm{R}}_{\mu} = \left\{ \beta \,\middle|\, \beta \in \mathrm{Suffix}\big((\alpha)^{\mathrm{R}}_{\mu}\big) \wedge |\beta| \geq k \right\}.$$

Consequently, the factors $\lambda$ and $\mu$ are special in that they are always the longest members of their respective classes under $\equiv^{\mathrm{R}}_{\mu}$. The right end equivalence and its properties are used to combine similar subtrees as follows.

**Definition (Directed Acyclic Word Graph)**
The *directed acyclic word graph* for a string $\mu$ is a graph $\mathrm{DAWG}(\mu) = (V, E, L)$, with the following components:

$$V = \left\{ [\alpha]^{\mathrm{R}}_{\mu} \,\middle|\, \alpha \in \mathrm{Factor}(\mu) \right\},$$

$$E = \left\{ ([\alpha]^{\mathrm{R}}_{\mu}, a, [\alpha a]^{\mathrm{R}}_{\mu}) \,\middle|\, [\alpha]^{\mathrm{R}}_{\mu}, [\alpha a]^{\mathrm{R}}_{\mu} \in V \wedge a \in \Sigma \right\},$$

$$L = \left\{ ([a\alpha]^{\mathrm{R}}_{\mu}, [\alpha]^{\mathrm{R}}_{\mu}) \,\middle|\, [a\alpha]^{\mathrm{R}}_{\mu}, [\alpha]^{\mathrm{R}}_{\mu} \in V \wedge [a\alpha]^{\mathrm{R}}_{\mu} \neq [\alpha]^{\mathrm{R}}_{\mu} \wedge a \in \Sigma \right\}.$$

The use of the right end equivalence changed the definition very significantly compared to the suffix trie definition. The differences are bigger than after the use of the right extension in the suffix tree definition. However, like in the suffix trie, all factors of $\mu$ are explicitly represented in the DAWG, albeit grouped using right end equivalence. While the edges are once again labelled by single symbols only, the introduction of right end equivalence classes made some of DAWG's other properties less obvious.

Unlike the previous two suffix graph types, the DAWG is not a tree. However, it still shares some properties with both of them.

**Claim 2.14 (Shape of the DAWG)**
The DAWG for string $\mu$ is a directed acyclic graph with source $[\lambda]^{\mathrm{R}}_{\mu}$ and sink $[\mu]^{\mathrm{R}}_{\mu}$.

**Claim 2.15 (Contents of the DAWG)**
The set of all strings represented by paths from the source of the DAWG($\mu$) is equal to the set of all factors of $\mu$.

**Claim 2.16 (Unique Branching in the DAWG)**
Every vertex of the DAWG for string $\mu$ has at most one $a$-edge for every $a$ from alphabet $\Sigma$.

**Claim 2.17 (Space Complexity of the DAWG [6])**
The are at most $2|\mu| - 1$ vertices and at most $3|\mu| - 4$ edges in DAWG($\mu$) for any string $\mu$ with at least two symbols.

## 2.4 Compact Directed Acyclic Word Graph

Naturally, one may ask, what would a combination of the path compression and the vertex minimisation yield. What happens to the suffix tree after the vertex minimisation and what about the DAWG after the path compression? The answer is the CDAWG, which is defined below and illustrated in Figures 2.1d and 2.2d. The first figure illustrates significant changes compared to the DAWG, while the second figure highlights the differences against the suffix tree.

The formal definition of the CDAWG is a straightforward combination of the definition of the suffix tree and the DAWG.

**Definition (Compact Directed Acyclic Word Graph)**
The *compact directed acyclic word graph* for a string $\mu$ is a graph CDAWG($\mu$) = $(V, E, L)$ that has the following components:

$$V = \left\{ [\alpha]^{\mathrm{R}}_{\mu} \;\middle|\; \alpha \in \mathrm{Explicit}(\mu) \right\},$$

$$E = \left\{ ([\alpha]^{\mathrm{R}}_{\mu}, a\beta, [\alpha a\beta]^{\mathrm{R}}_{\mu}) \;\middle|\; [\alpha]^{\mathrm{R}}_{\mu}, [\alpha a\beta]^{\mathrm{R}}_{\mu} \in V \wedge a \in \Sigma \wedge \beta \in \mathrm{Factor}(\mu) \wedge \langle \alpha a \rangle^{\mathrm{R}}_{\mu} = \alpha a\beta \right\},$$

$$L = \left\{ ([a\alpha]^{\mathrm{R}}_{\mu}, [\alpha]^{\mathrm{R}}_{\mu}) \;\middle|\; [a\alpha]^{\mathrm{R}}_{\mu}, [\alpha]^{\mathrm{R}}_{\mu} \in V \wedge [a\alpha]^{\mathrm{R}}_{\mu} \neq [\alpha]^{\mathrm{R}}_{\mu} \wedge a \in \Sigma \right\}.$$

Note that, like the suffix tree definition, this definition is adjusted to ease on-line construction. It was introduced by Inenaga et al. [22] and differs slightly from the original definition given by Blumer et al. [7].

When it comes to the properties of the CDAWG, the preceding analysis of properties of the suffix tree and the DAWG left only a little space for surprise. The following five claims recall the basic properties of the CDAWG. Note that the first three are trivial and the other two can be readily obtained from the properties of suffix tree (Claims 2.7 – 2.11) through vertex minimisation.

**Claim 2.18 (Shape of the CDAWG)**
The CDAWG for string $\mu$ is a directed acyclic graph with source $[\lambda]^{\mathrm{R}}_{\mu}$ and sink $[\mu]^{\mathrm{R}}_{\mu}$.

**Claim 2.19 (Contents of the CDAWG)**
The set of all strings represented by implicit paths from the source of the CDAWG($\mu$) is equal to the set of all factors of $\mu$.

**Claim 2.20 (Unique Branching in the CDAWG)**
Every vertex of the CDAWG for string $\mu$ has at most one $a$-edge for every $a$ from alphabet $\Sigma$.

**Claim 2.21 (Space Complexity of the CDAWG)**
The are at most $|\mu|$ vertices and at most $2|\mu|-2$ edges in CDAWG($\mu$) for any string $\mu$ with at least two symbols.

**Claim 2.22 (Suffix Links of the CDAWG)**
There is a suffix link leading from every vertex of CDAWG($\mu$), with the exception of the source and possibly the sink. The source has no suffix link and the suffix link from the sink exists if and only if the longest non-unique suffix of $\mu$ is a branching factor of $\mu$. All other suffix links lead from one branching node to another branching node.

Another interesting point not made by the above claim is that the classes not containing explicit factors of $\mu$ may not be represented by a single implicit vertex. This is illustrated in Figure 2.1d, where the members of classes $[ccco]^{\mathrm{R}}_{\mu} = \{ccco, cco, co\}$ and $[cccoo]^{\mathrm{R}}_{\mu} = \{cccoo, ccoo, coo\}$ are represented by three distinct implicit nodes each. A different situation is shown in Figure 2.2d, where the members of the class $[coco]^{\mathrm{R}}_{\mu} = \{coco, oco\}$ and $[coc]^{\mathrm{R}}_{\mu} = \{coc, oc\}$ are still represented by a single implicit node.

# 3. Construction

After dealing with fundamentals in Chapter 1: Fundamentals and suffix graph definitions in Chapter 2: Definitions, it is time for the next logical step on the path to suffix graph based data compression: the suffix graph construction. This area has seen a lot of research and produced a wide spectrum of construction methods with various goals and properties [52, 32, 51, 12, 20, 49].

The suffix tree was introduced by Weiner with a right-to-left construction algorithm which used prefix links to create the suffix tree in time linear in the length of the input string [52, 18]. Weiner's work was superseded by McCreight's more practical algorithm that works left-to-right and, with the help of suffix links, achieves the same asymptotic complexity [32]. McCreight's method was in turn improved upon by Ukkonen, who created the first left-to-right construction algorithm that can build the suffix tree one symbol at a time (i.e. on-line), and is asymptotically as fast as its predecessors [51]. Note that all three algorithms are bound by $O(|\mu|B)$ in the worst case, where $B$ is the worst-case cost of the operation which finds the requested $a$-edge leading from the current node (*branching operation*).

To remove the branching operation cost, Farach took a different approach and created a linear time algorithm that works in $(\Theta(|\mu| + S(|\mu|)))$, where $S(|\mu|)$ is the time needed for sorting $|\mu|$ symbols [12]. However, this divide-and-conquer algorithm needs to know the whole string before it starts and requires the input alphabet to be indexed, which practically limits the alphabet to be a set of integers. More practical algorithms were created with large strings and disk storage in mind [20, 49, 35, 29]. These algorithm strive to reduce the in-memory storage requirements and eliminate random access introduced by the use of suffix links and branching operations. Consequently, thanks to the nature of current virtual memory hierarchies, these algorithms are reported to be faster than traditional algorithms even in-memory, despite having $\Theta(|\mu|^2)$ or worse worst-case asymptotic bound [20, 49, 35, 29].

Other three suffix graphs have received somewhat less attention. Ukkonen devised a quadratic time on-line algorithm for suffix trie construction that served as a basis for his linear time suffix tree building algorithm [51]. This algorithm proved to be so versatile that it was later adapted to work on DAWG and CDAWG in linear time as well [51, 22]. However, the same can be said about McCreight's algorithm which was altered by Crochemore and Vérin to create CDAWGs in linear time. Nevertheless, these algorithms were preceded by linear time algorithms that were introduced along the definition of both DAWG and CDAWG [6, 7]. Note that, like in the case of the suffix tree construction, these algorithms are bounded by O($|\mu|B$), again with the burden of the branching factor.

While there exist so many interesting construction algorithms, only some are useful in the context of the suffix graph based data compression and sliding window. To be suitable for this purpose, the construction algorithms need to work incrementally left-to-right and construct the desired graph directly. If we add an additional requirement for similarity between construction algorithms for all four suffix graphs, only two choices remain. As Ukkonen's and McCreight's algorithms are very similar, and even do the same steps in the same order [18], we prefer the more elegant work of Ukkonen and Inenaga et al. [51, 22].

The body of this chapter is composed of two sections. The first section gives an overview of Ukkonen-type construction algorithms for all four suffix graph types made by others, while the second section brings our original results regarding the implicit suffix link simulation in these algorithms.

## 3.1 Algorithms

As noted above, all four suffix graph types can be constructed by four variants of the base algorithm devised by Ukkonen [51]. Ukkonen created the base algorithm for the suffix trie and modified it to work on the suffix tree. He also pointed out that it could be easily adapted to construct DAWG and that the resulting algorithm would be similar to that given by Blumer et al. [6]. The last remaining version required to construct CDAWG had to wait to be discovered by Inenaga et al. a few years later [22].

Like all on-line algorithms, Ukkonen's algorithm creates the desired data structure incrementally, while keeping the data structure valid in each step. This high level step consists of appending a new symbol to the underlying string and adjusting the suffix graph accordingly. In other words, the step receives the suffix graph for string $\mu$, a new symbol $a$, and produces the suffix graph for string $\mu a$. This step is performed by function `AppendSymbol` in Algorithm 3.1. The complete construction algorithm is then a straightforward application of symbol appending for every symbol of the input string as demonstrated by function `Build` shown in Algorithm 3.1 as well. The rest of Algorithm 3.1 contains the rest of high-level pseudo-C++ code common to all four algorithm variants. All these shared functions are described below in more detail, along with notes about version-specific functions they use. The construction process itself is then illustrated in Figures 3.1 and 3.2 for the DAWG, Figure 5.3 for the suffix tree, and Figure 5.4 for the CDAWG.

`Build`

This is the the main function, which creates the suffix graph for the input string. It creates an empty graph using function `CreateEmptyGraph` and then uses the function `AppendString` to append all symbols of the input strings.

`AppendString`
The purpose of this function is to iterate over the input string from the left to the right and apply the append symbol operation, in the form of function `AppendSymbol`, to each symbol.

`AppendSymbol`
The idea behind this function is to follow the so called *active point* and do local changes in its vicinity. As pointed out by Ukkonen, all necessary graph changes happen near suffixes of the original string and the likelihood of change decreases with the length of the suffix [51]. Thus, the suffixes of the original string are traversed in the longest to the shortest order. Luckily, they are connected by suffix links and form the so called *boundary path*. The active point is moved along the boundary path and the changes needed to modify the graph around its location are made. Most of this work is done by function `MoveActivePointDown`, but functions `ResetActivePointPosition`, `CheckAndSplitActivePointNode` and `AddLastSuffixLink` are used to prepare and finish the work, respectively.

`MoveActivePointDown`
This function is moving the active point along the boundary path and tries to move it down from its current location, along the *a*-edge dictated by the new symbol *a*. The attempt is made using function `MoveActivePointDownIfPossible`. If it succeeds to move the active point down, then no shorter suffixes of $\mu$ need attention, and the work of function `MoveActivePointDown` is done. On the other hand, if the current implicit or explicit node does not have such edge, the missing edge must be created using function `CreateEdgeFromActivePoint`. After that, the active point has to be moved sideways, along the boundary path, to the next shorter suffix location, using function `MoveActivePointSideways`. Following the active point move sideways, the function loops back to the beginning to make another attempt to move the active point down. To avoid having to handle the origin — which has no suffix link — as a special case, the suffix graphs used by this algorithm are augmented with a bot vertex, its edges and suffix link, as described in Section 1.2. This is reflected in function `CreateEmptyGraph` as shown in Algorithm 3.1.

`CreateEmptyGraph`
This function creates the augmented empty suffix graph, which was described in Section 1.2. It is composed of two vertices, several edges and one suffix link. The vertices are the bot vertex and the origin vertex, and the single suffix link leads from the origin to the bot. All edges lead from the bot to the origin and are created using function `CreateBotOriginEdges`.

```
void Build(String & string)
{
    CreateEmptyGraph();
    AppendString(string);
}
void AppendString(String & string)
{
    for (symbol in string)
        AppendSymbol(symbol);
}
void AppendSymbol(Symbol & symbol)
{
    ResetActivePointPosition();
    MoveActivePointDown(symbol);
    CheckAndSplitActivePointNode();
    AddLastSuffixLink();
}
void MoveActivePointDown(Symbol & symbol)
{
    while (! MoveActivePointDownIfPossible(symbol)) {
        CreateEdgeFromActivePoint();
        MoveActivePointSideways();
    }
}
void CreateEmptyGraph()
{
    bot = CreateVertex();
    origin = CreateVertex();

    CreateBotOriginEdges();
    CreateSuffixLink(origin, bot);
    MoveActivePointTo(origin);
}
void CreateBotOriginEdges()
{
    for (symbol in ALPHABET) {
        CreateEdge(bot, origin, symbol);
    }
}
```

**Algorithm 3.1.** Outline of Ukkonen's construction algorithm that works on all four suffix graph types.

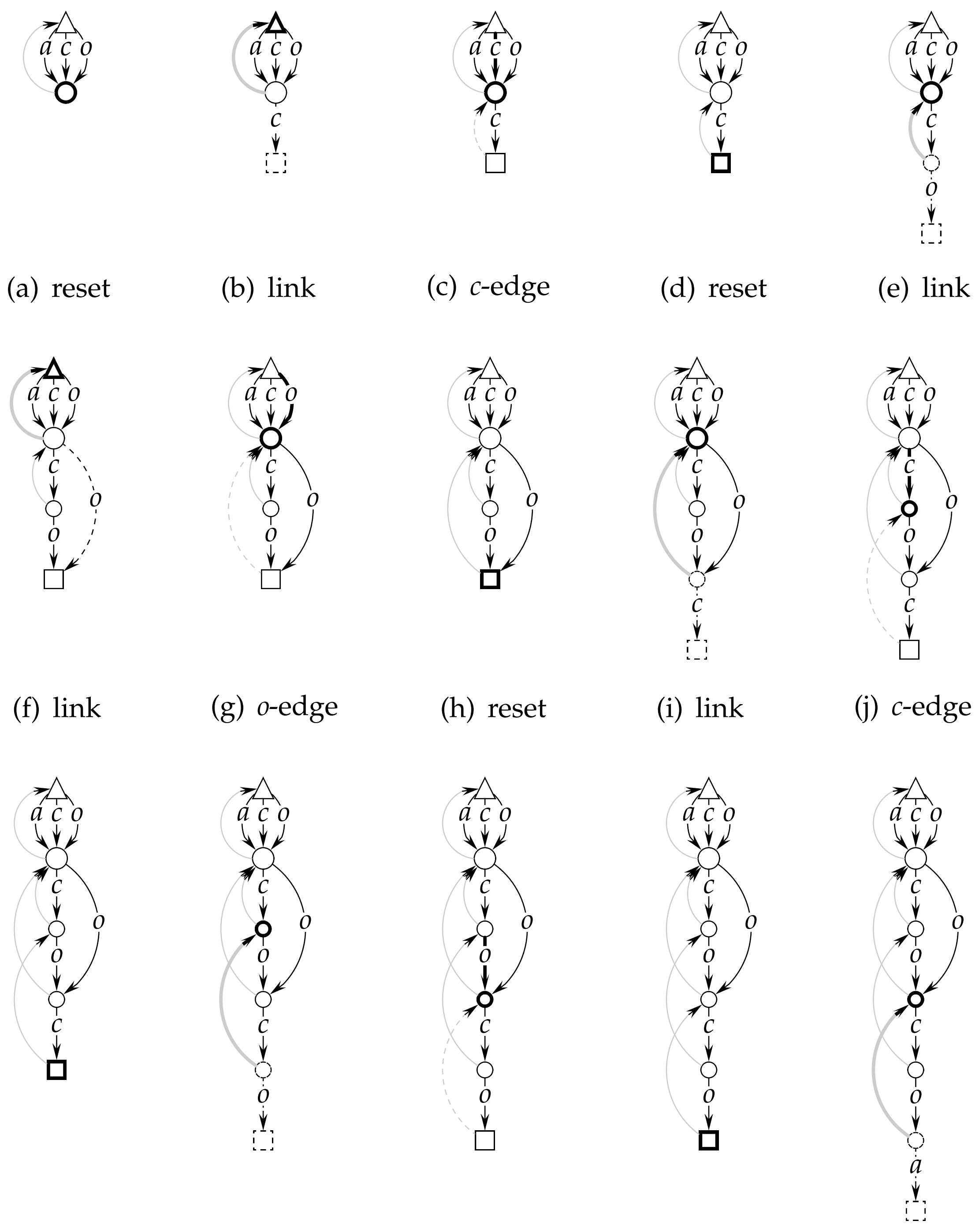
`CreateBotOriginEdges`
: The only thing this function does is that it creates all edges needed between the bot and the origin. The bot is the initial vertex, the origin is the terminal vertex, and exactly one edge with label $c$ is added for every symbol $c$ of the input alphabet.

All parts of the algorithm described so far were universal and are shared by all its versions, however, there are also some graph type specific parts. As these are not only graph but also implementation specific, we provide no pseudocode and give only a textual description instead.

`ResetActivePointPosition`
: This function moves the active point to the longest suffix of $\mu a$, i.e. the $\mu a$ itself, before the next append-symbol-step. However, it only does so for the suffix trie and the DAWG, and leaves the active point where it is for the other two graph types. The reason is that if it did reset the position for the suffix tree, the construction algorithm would not achieve the promised linear time complexity, as there are too many suffixes to be examined. To solve this issue, Ukkonen devised the notion of *open edges* [51]. An open edge is an edge with a leaf as its terminal vertex and an *open label*. That is, the label has only the left end index and its right end index is the end of the underlying string. This ensures that all leaf edges append the new symbol to their labels automatically with each append-symbol-step, and do not need to be updated separately. A similar approach is used for CDAWG and sink in-edges.

`CheckAndSplitActivePointNode`
: This function does nothing for trees, but it checks and possibly splits the current active point node for the DAWG and the CDAWG. The node stays as it is if the active point has reached the current node by moving down the primary in-edge [6, 22]. In this context, the primary in-edge is the last on the longest path from the source to the current node. It can either be marked as primary on node creation, or identified at any time by checking whether the sum of the depth of its initial node and the length of its label equal the depth of its terminal node [6, 22]. The latter approach appears to be more suitable if there are going to be deletes in the graph, like in the sliding window application. Note that the examples of node splits can be seen in Figures 3.2 and 5.4.

`AddLastSuffixLink`
: This simple function updates the suffix link between the vertex representing the shortest unique suffix and the node representing the longest non-unique suffix of $\mu a$. The former is either the last leaf created or the sink, while the latter is the current active point node. The situation is simple in the suffix trie and the DAWG, which only have explicit vertices. However,



(a) reset (b) link (c) *c*-edge (d) reset (e) link

(f) link (g) *o*-edge (h) reset (i) link (j) *c*-edge

(k) reset (l) link (m) *o*-edge (n) reset (o) link

**Figure 3.1.** Step by step construction of DAWG(*cocoao*) with action comments (Part I). The symbolics used is described in Table 1.1.

the implicit active point node case of the suffix tree requires a special treatment. If the current active point node is implicit, no suffix link can be added. Moreover, as some append-symbol-steps do not create new leaves, the suffix link of the last leaf created by preceding steps must be removed, as it is no longer valid. The reason is that the open edges automatically ad-

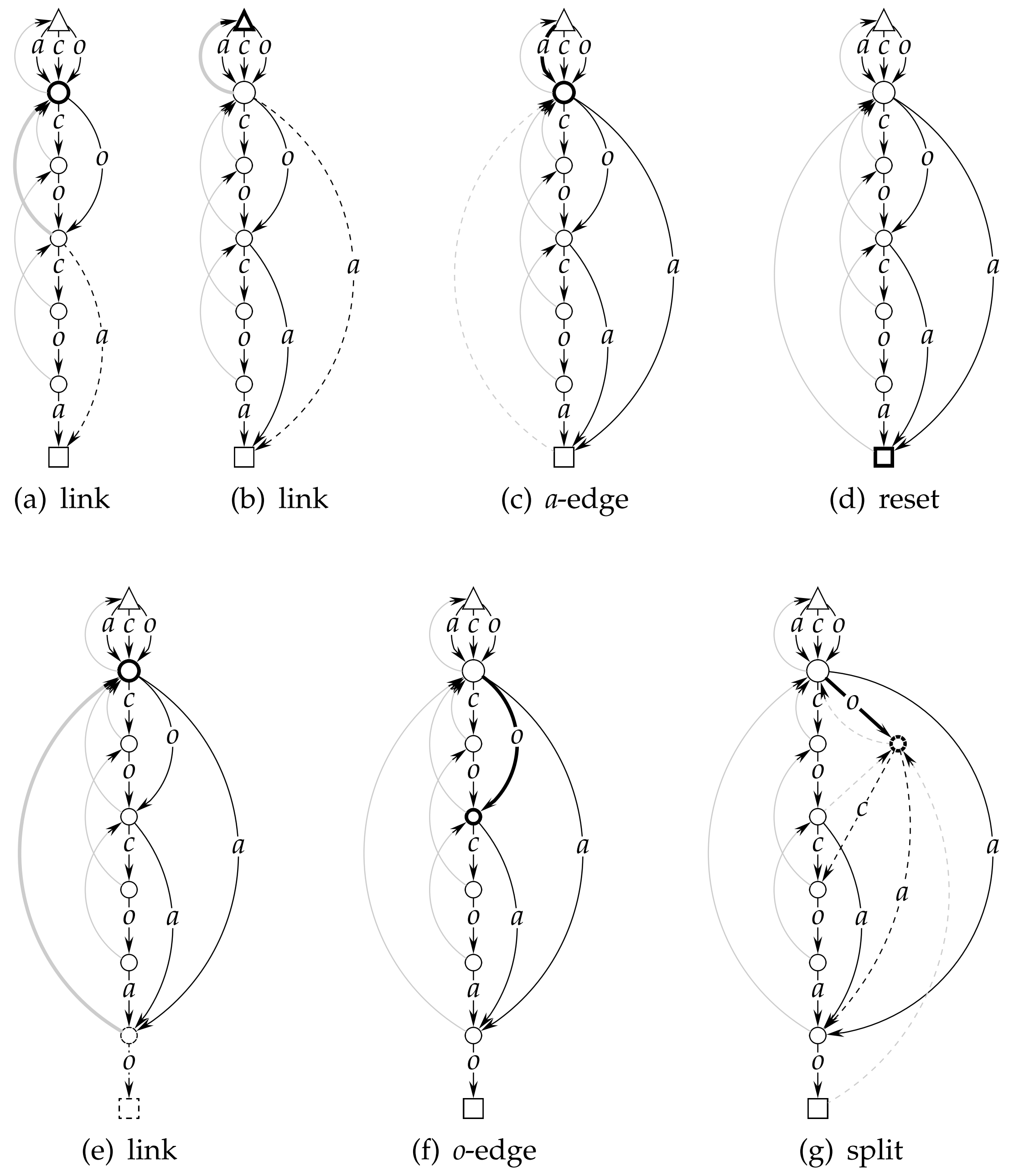


**Figure 3.2.** Step by step construction of DAWG(*cocoao*) with action comments (Part II). The symbolics used is described in Table 1.1.

justed all leaves for the new string, but the target of the last leaf suffix link remained the same. Note that this last suffix link is never needed during construction. Also, all its versions, with the exception of the last, will be replaced by suffix links to the next leaf created. For these reasons, it might be a good idea to ignore this suffix link and only add the final version when the construction is done. Something similar happens to the last suffix link in the CDAWG as well, only the sink is used in place of the last leaf. Like in the case of the suffix tree, the suggested solution is to wait for the final version, if possible.

`MoveActivePointDownIfPossible`

This function tries to move the active point down from its current location using the desired implicit or explicit *a*-edge. It has to be able to check the path down in any of the four suffix graph types and return the implicit or explicit target vertex, if available. The suffix trie and the DAWG are easy, as there are only explicit vertices and edges in these graphs, and edge labels are only one symbol long. However, in the suffix tree or the CDAWG, there are implicit vertices and edges as well as labels specified by a pair of indexes into the input string.

`CreateEdgeFromActivePoint`

This function creates a new edge from the current active point node and labels it with the new symbol *a*. This is easy in the suffix trie, where the current node is explicit. Here, a new leaf is created along with a suffix link from the last leaf created and the required *a*-edge. The situation is even simpler in the DAWG, where leaves are replaced by the sink, which needs to be created only once.

The possibility that the current node might be implicit complicates the situation in suffix trees. In that case, a new explicit node is created in its place, and only after that comes the creation of a new leaf and edge. To create a new explicit node, the current explicit edge has to be split into two new edges at the current implicit node, and a new explicit node is placed between them. Moreover, if this append-symbol-step has already created at least one other new node, a new suffix link has to be added to lead from the last of them to the node which was just created. Note that like in the case of suffix trie, a suffix link must be added to lead from the last leaf created previously to the one just created.

It gets even more complicated in the CDAWG case as illustrated in Figure 5.4. Not only is there a possibility of an implicit current node, but there is also a chance that the appropriate explicit node already exists elsewhere. The reason is that this new explicit node will from now on represent a formerly implicit class, which was, up to now, represented by several implicit nodes. Fortunately, this node would be the last node created in this append-symbol-step. Moreover, it suffices to check that the distance from the nearest explicit vertex below is the same for both the last node created and the current implicit vertex. If they are the same, the current edge is cut at the implicit node and redirected to the last node created, while the rest of the edge is thrown away. Otherwise, the current node is made explicit, like in the suffix tree case. The rest is the same as for the DAWG.

`MoveActivePointSideways`

Function `MoveActivePointSideways` moves the active point to the next shorter suffix of the input string using a suffix link. This is again simple in the suffix trie and the DAWG, where all suffix links are explicit. However, it is also more complicated in the suffix tree and the CDAWG, where some suffix links might be implicit. As these implicit suffix links are not present in the graph, they have to be simulated. The methods of their simulation are studied in detail in the next section.

The remaining functions, e.g. `CreateVertex`, `CreateEdge` and `CreateSuffixLink`, are implementation dependent and do exactly what their names suggest.

## 3.2 Implicit Suffix Link Simulation

As noted in the description of function `MoveActivePointSideways` in the previous section, the two path compressed graphs have to use implicit suffix links for fast boundary path traversal. However, due to their virtual nature, these implicit suffix links have to be simulated using objects that are explicitly present in the graph, e.g. explicit vertices, edges and suffix links. Moreover, while the target of implicit suffix link could be easily found using a search from the origin, the time it would take would be proportionate to the length of the factor we are looking for. The sum of these factor lengths over the whole construction process is going to be quadratic. Obviously, as this would prevent a construction in linear time, a more sophisticated solution is needed. This section, which is based on the original research published previously in the Journal of Discrete Algorithms [39], describes and analyses three such solutions.

A natural way to approach the problem of implicit suffix link simulation is to look for some approximation using explicit objects which will be easy to make exact later. In our situation, a sideways move over the closest explicit suffix link looks like a good start. However, while the current location of the active point is in an explicit node, this node is the last node created by function `CreateEdgeFromActivePoint` and lacks suffix link. On the other hand, suffix links are present at both the initial and the original terminal vertex of the edge that was just split or redirected by function `CreateEdgeFromActivePoint`. The traditional choice in this situation is the suffix link leading from the initial vertex [32, 51, 22]. However, as will be shown shortly, the other choice is sometimes the better choice. From now on, the traditional approach will be represented by function `ReScan`, the new approach will be represented by function `Climb`, and their combination by function `ClimbScan`. To ease the description of these functions, the edge that was just split or redirected by function `CreateEdgeFromActivePoint` is called old in the rest of this section.

`ReScan`

This function, shown in the pseudo-C++ code in Algorithm 3.2, represents the traditional top-down approach to the implicit suffix links simulation. It starts the simulation by moving the active point up to the initial node of the old edge and then sideways using the explicit suffix link of this node. This

```
void ReScan()
{
    String label;

    label = GetLabelAboveActivePoint();
    MoveActivePointUpToEdgeInitialVertex();
    MoveActivePointSidewaysExplicit();
    ActivePointBranchAndJumpDownLoop(label);
}

void ActivePointBranchAndJumpDownLoop(String & string)
{
    do {
        ActivePointBranchAndJumpDown(string);
    } while (Length(string) > 0);
}
```

**Algorithm 3.2.** Outline of `ReScan` implicit suffix link simulation.

```
void Climb()
{
    String label;

    label = GetLabelBelowActivePoint();
    MoveActivePointDownToEdgeTerminalVertex();
    MoveActivePointSidewaysExplicit();
    ActivePointJumpUpLoop(label);
}

void ActivePointJumpUpLoop(String & string)
{
    do {
        ActivePointJumpUp(string);
    } while (Length(string) > 0);
}
```

**Algorithm 3.3.** Outline of `Climb` implicit suffix link simulation.

```
void ClimbScan()
{
    Size jumpLimit = GetReScanEquivalentJumpLimitEstimate();

    SaveActivePoint();

    if (! LimitedClimb(jumpLimit)) {
        RestoreActivePoint();
        ReScan();
    }
}
```

**Algorithm 3.4.** Outline of `ClimbScan` implicit suffix link simulation.

first approximation of the simulated implicit suffix link takes three steps and uses the following three functions.

`GetLabelAboveActivePoint`
: The purpose of this function is to look at the part of the old edge which is now above the active point and return its label.

`MoveActivePointUpToEdgeInitialVertex`
: This function moves the active point up to the initial node of the old edge.

`MoveActivePointSidewaysExplicit`
: As its name suggests, this function moves the active point sideways over an explicit suffix link. The explicit suffix link used is the one leading from the current active point node.

As the active point was taken up before it was moved sideways, the target must be somewhere below the first approximation. It is found using the remembered label string and fast downward search which only has to check first symbols and lengths of edge labels along the way [32]. As the first symbol is checked during a branching operation, this search could be described as branch and jump down loop. It is implemented using the following two functions.

`ActivePointBranchAndJumpDownLoop`
: This function simply runs the `ActivePointBranchAndJumpDown` function until the string which is adjusted by `ActivePointBranchAndJumpDown` function becomes empty.

`ActivePointBranchAndJumpDown`
: As suggested by its name, this function performs a branching operation followed by a downward jump to move the active point closer to its target. The branching operation finds the appropriate edge using the first symbol of the supplied string in $\mathrm{O}(B)$ time. After that, this function compares the length of the string with the length of the current edge label and moves the active point down along the edge. If the string is longer than the current edge label, the active point is moved to the terminal vertex of this edge. Subsequently, a prefix of length equal to the length of the edge label is removed from the string. Otherwise, the active point is moved down the current edge so that its distance from the initial vertex equals the length of the supplied string. After that, the string is made empty.

Among the advantages of `ReScan` is the similarity of its top-down approach to the normal downward movement in the graph, and the ability to work for both the suffix tree and the CDAWG. On the other hand, its chief disadvantage is that the simulation requires many branching operations. These can be relatively complex and costly, when compared to other operations needed for suffix graph construction. The ratio largely depends on the choices made for the implementation of branching and suffix links.

`Climb`

This function is representing the bottom-up approach. As can be seen from the pseudo-C++ code in Algorithm 3.3, it first approximates the simulated implicit suffix link by moving the active point down and then sideways using explicit edge and suffix link. Like in the case of `ReScan`, this is done in three steps. However, the first two function calls are different.

`GetLabelBelowActivePoint`

The purpose of this function is to look at the part of the old edge which is now below the active point and return its label.

`MoveActivePointDownToEdgeTerminalVertex`

This function moves the active point down to the terminal node of the old edge.

This time, the first approximation is below the target of the implicit suffix link being simulated, and the target is found using an upward scan. Note that due to the way in which the first approximation was found, labels of edges being traversed do not have to be checked as they are guaranteed to match the supplied string. Thus, only lengths are important, and the rest of the simulation is just a jump up loop, implemented using the following two functions.

`ActivePointJumpUpLoop`

This function simply runs the `ActivePointJumpUp` function, until the string, which is adjusted by `ActivePointJumpUp` function, becomes empty.

`ActivePointJumpUp`

This function compares the length of the string to be traced with the length of the label of the in-edge terminating in the active point vertex. If the string is longer than this label, the active point is moved to the initial vertex of this edge. Subsequently, a suffix of length equal to the length of the edge label is removed from the end of the string. Otherwise, the active point is moved up this in-edge, so that its distance from the terminal vertex equals the length of the supplied string. After that, the string is made empty.

Note that as the number of in-edges entering the sink in the CDAWG is not bound by any constant, the use of the bottom-up approach is restricted to the suffix tree.

`ClimbScan`

This function combines the bottom-up approach of `Climb` with the top-down approach of `ReScan`, to get the best of both worlds. The idea is to try the bottom-up approach and fall back to the top-down approach, if it takes too long. This is illustrated by the pseudo-C++ code in Algorithm 3.4 and more detail is given below.

`GetReScanEquivalentJumpLimitEstimate`

This function attempts to estimate the number of jumps that can be taken during the bottom-up simulation, before a fallback to the top-down approach is in order. While having one global constant limit would be the easiest way to achieve amortised constant time per one implicit suffix link simulation, there is no obvious way to choose it for all strings. However, as the `ReScan` is known to have the desired complexity, we can use an estimate of the number of steps needed for `ReScan` as a base for a variable limit. As illustrated by the pseudo-C++ code in Algorithm 3.2, the number of branch and jump steps needed is limited by the length of the label above the active point. If we multiply this estimate by some small global constant, say eight, and use it to limit the bottom-up approach, the `ClimbScan` is guaranteed to achieve the same asymptotic bounds as `ReScan`.

`SaveActivePoint`

As suggested by its name, this function saves the information about the active point, so that it can be restored later using function `Restore-ActivePoint`.

`LimitedClimb`

This is a limited version of the `Climb` function. The value of parameter `jumpLimit` is used to limit the number of calls to `ActivePointJumpUp`, in addition to the constraints used in the original `ActivePointJum-pUpLoop`.

`RestoreActivePoint`

As suggested by its name, this function restores the information about the active point, previously saved by a call to function `SaveActive-Point`.

### 3.2.1 Complexity

Now that we have the description of the three implicit suffix link simulation methods, it is time to analyse them. This is easy for `ReScan`, which is well studied, and known to work in O($B$) amortised time per one implicit suffix link simulation [32, 22]. On the other hand, the bottom-up approach is new and in need of analysis.

#### 3.2.1.1 Hidden Costs

A brief look at pseudo-C++ code of functions `ReScan` and `Climb` reveals, that, apart from using two different strategies, they are very similar. Moreover, as there is no need for branching in `Climb`, it should be faster and obviously a better choice. However, there are some costs associated with the bottom-up approach that may not be apparent from the pseudocode.

Reverse Edges

The first issue is that we have to traverse edges in the opposite direction then they are defined and used in construction algorithms. Note that, as pointed out above in the discussion of function `GetReScanEquivalentJumpLimitEstimate`, this is not usable in the CDAWG due to the possibly large number of in-edges per vertex. Consequently, from now on, the discussion is limited to suffix trees only. While reverse edges consume additional resources, they can be integrated into vertices to limit space usage and are updated only once per forward edge modification. Moreover, some algorithms need them anyway, e.g. to make efficient sliding of the suffix tree possible [14, 26, 41]. In that case, there is no additional cost for reverse edges.

Leaf Suffix Links

The second issue which may not be apparent from the pseudocode in Algorithm 3.3 lies in the need to maintain and use leaf suffix links. Despite the fact that these suffix links appear in our suffix tree definition, they are not used by many algorithms, including the original construction by Ukkonen [51]. However, as demonstrated by McCreight [32], there are ways how to hide them, e.g. by a smart leaf numbering.

#### 3.2.1.2 Correctness

While the idea behind the `Climb` method appears to work, its correctness must be verified. However, it is clear that it moves the active point to the correct position as long as both the explicit suffix link and all reverse edges that are used for simulation exist. While the existence of reverse edges is guaranteed, the situation with the explicit suffix links deserves a clarification. There are moments during the suffix tree construction when both the last leaf created and the last node created lack explicit suffix links. However, the simulated suffix link leads from the last node created and the last leaf created is its newer child which will not be used during simulation. Thus, when needed, the required suffix link always exists.

#### 3.2.1.3 Time Complexity

The last open question regarding the bottom-up approach is the one about its time complexity. As noted above, `ReScan` works in an $\mathrm{O}(B)$ amortised time

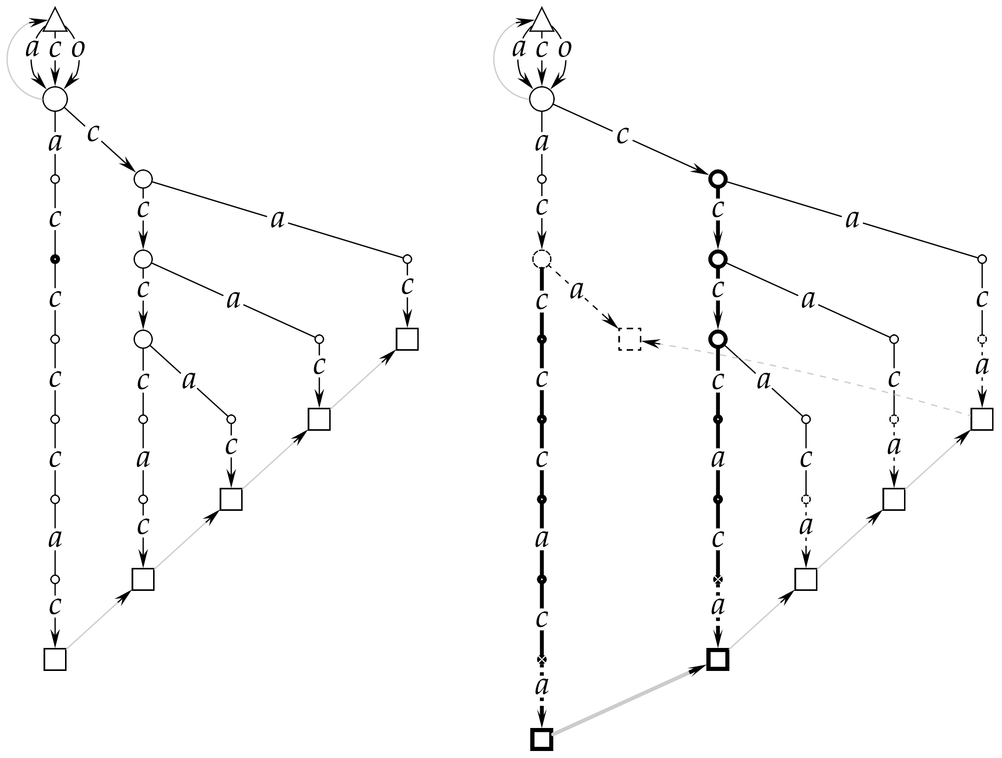


**Figure 3.3.** Pathological behaviour of `Climb` demonstrated on adversary string. The symbolics used is described in Table 1.1.

per simulation, and one would expect a similar bound for the `Climb` algorithm. Unfortunately, it is possible to find an adversary string which forces `Climb` to work in a total superlinear time per suffix tree construction.

**Lemma 3.1 (Adversary String for Climb)**
Let $k \geq m \geq 1$, $a \neq c \in \Sigma$ and $\varsigma = ac^k acac^2 ac^3 \ldots ac^m a$. Then function `Climb` performs at least $\Omega((2k - m)m)$ steps during the construction of SuffixTree($\varsigma$).

**Proof:** Let $T_i$ be the suffix tree created by function `Build` shown in Algorithm 3.1 after the prefix $\varsigma_i = ac^k acac^2 \ldots ac^i a$ of $\varsigma$ is processed for some $0 < i < m$. Note that there are two types of root-leaf paths in this tree, which we name $a$-path and $c$-path, respectively. Paths of the former type represent the suffixes starting with symbol $a$, e.g. $\varsigma_i$. Similarly, paths of the latter type represent the suffixes that start with symbol $c$, e.g. $\varsigma_i[2 \,..\, |\varsigma_i|]$. While the $a$-path contains explicit nodes $ac^j$ for all $1 < j < i$, the $c$-path contains explicit nodes $c^j$ for all $1 < j < k - 1$, since all these nodes correspond to right branching factors of $\varsigma_i$.

Now consider the actions performed by function `Build` to transform $T_i$ into $T_{i+1}$. Note that the call of procedure `CreateEdgeFromActivePoint` makes the implicit node $ac^{i+1}$, which lies on one of the $a$-paths, explicit. After that, function `MoveActivePointSideways` is called, which has to simulate the implicit suffix link from $ac^{i+1}$ to $c^{i+1}$ using function `Climb`. Function `Climb` then starts by moving the active point down to the older leaf connected to the new explicit node $ac^{i+1}$. The next step then moves the active point using the explicit suffix link leading from this $a$-path leaf to some $c$-path leaf. After that, vertices $c^j$ for all $k-1 \geq j \geq i+1$ are visited, until the target $c^{i+1}$ is reached. Hence, at least $k-i$ steps are required for this simulation. As the construction of SuffixTree($\varsigma$) performs such transformation for all $0 < i < m$, it follows that the total number of steps is bounded from below by

$$\sum_{i=0}^{m-1} k - i = \Omega\big((2k-m)m\big) .$$

■

**Corollary 3.2 (Climb Lower Bound)**
Function `Build` requires $\Omega(n^{3/2})$ time in the worst case, when constructing a suffix tree for a length $n$ string, while using function `Climb` for implicit suffix links simulation.

**Proof:** Let $f(i) = 2 + \frac{3}{2}i(i+1)$ for every positive integer $i$. Then, for any arbitrary positive integer $n$, there exists $i_n$ such that $n = f(i_n) + r$, where $r < f(i_{n+1}) - f(i_n) = 3i_n + 3$. Now consider a string $\mu$ of length $n$, whose prefix of length $f(i_n)$ equals

$$\varsigma = ac^{i_n^2} acac^2ac^3 \ldots ac^{i_n}a ,$$

while the remaining symbols are arbitrary. As function `Build` creates SuffixTree($\varsigma$) during the construction of SuffixTree($\mu$), the Lemma 3.1 can be applied. The number $T(n)$ of steps required to construct SuffixTree($\varsigma$), while using function `Climb`, satisfies $T(n) = \Omega((2i_n^2 - i_n)i_n)$. Since $n = \Omega(i_n^2)$, it follows that $T(n) = \Theta(n^{3/2})$ as claimed. ■

This means that the worst case time complexity of this approach is superlinear. However, the exact complexity is as yet unknown. Nevertheless, we can provide the following upper bound estimate.

**Proposition 3.3 (Climb Upper Bound)**
The time needed by function `Build` to construct SuffixTree($\mu$), while using function `Climb` for implicit suffix link simulation, is bounded by

$$\mathrm{O}(nB) + \mathrm{O}\left(\sum_{i=1}^{n}\left|\mathrm{Prefix}\big(\alpha_i\beta_i\big) \cap \mathrm{Branch}^{\mathrm{R}}\big(\mu[1\,..\,i]\big)\right|\right) .$$

Here $n = |\mu|$, $B$ is the complexity of branching, and $\beta_i$ $(1 < i < n)$ is the string defined by $\langle\alpha_i\rangle_\mu^{\mathrm{R}} = \alpha_i\beta_i$, where $\alpha_i$ is the active point of $\mu[1\,..\,i]$.

**Proof:** The first term covers the complexity of Ukkonen's algorithm without implicit suffix link simulation [51]. The second term then bounds the number of distinct right branching prefixes of $\beta_i$. These prefixes correspond to distinct nonleaf vertices visited by function `Climb`, while transforming SuffixTree($\mu[1\,..\,i-1]$) to SuffixTree($\mu[1\,..\,i]$). ■

Note that the height of SuffixTree($\mu[1\,..\,i]$) can be used as a trivial upper bound for $|\mathrm{Prefix}(\beta_i) \cap \mathrm{Branch}^{\mathrm{R}}(\mu[1\,..\,i])|$. Moreover, the height of the suffix tree has a linear worst case and a logarithmic expected case, assuming random input strings whose symbols are drawn independently from $\Sigma$ with a fixed probability distribution [2]. This leads to $\mathrm{O}(n^2)$ worst case time and $\mathrm{O}(n\log n)$ expected time. However, we believe that these bounds are not tight and can be improved upon.

In particular, function `Build` using function `Climb` works in $\mathrm{O}(nB)$ time for every strings $\mu$ satisfying

$$\sum_{i=1}^{n} \left| \mathrm{Prefix}\left(\alpha_i\beta_i\right) \cap \mathrm{Branch}^{\mathrm{R}}\left(\mu[1\,..\,i]\right) \right| = \mathrm{O}\left(|\mu|\right) .$$

Note that results of experiments in Section 3.2.2 suggest that this may be quite common case for real-life data.

### 3.2.2 Experimental Evaluation

Now that we have described and analysed the three simulation techniques, it is the right time for an experimental evaluation. There are two main areas we would like to test. The first is the implementation independent number of operations used for simulation and the second one is the implementation dependent execution time. To this end, we have implemented all three techniques in C++ and compiled them with GNU g++ compiler version 4.5.2 with full optimisations activated. The experiments were conducted on an AMD Athlon II X3 445 processor with a fixed 3.1GHz frequency and 16 GiB of main memory under the Ubuntu 11.04 operating system.

Our suffix tree implementation was originally developed with suffix tree based data compression in mind. Note that it uses linked lists to keep track of children which appears to be the best choice available if we want to use techniques like exclusion [26, 41]. All linked lists are used in one the following two modes.

Basic
: The new child is always prepended to the child list and search operations do not change the list.

Move-to-front (MTF)
: Like in the basic mode, the new child is prepended to the front of the child list. However, a move-to-front heuristic is used to reorganise this list during branching searches made by `MoveActivePointDownIfPossible`. The move-to-front heuristic moves the target of a search to the head of the list so that it can be found faster if it is used frequently.

| File | Description |
|---|---|
| | Pizza&Chili Corpus |
| SOURCES | concatenated source code files |
| PITCHES | human readable MIDI files |
| PROTEINS | protein sequences |
| PROTEINS.400 | 400MiB prefix of PROTEINS |
| PROTEINS.OLD | older version of PROTEINS |
| DNA | DNA sequences |
| ENGLISH | English texts |
| ENGLISH.400 | 400 MiB prefix of ENGLISH |
| XML | XML formatted bibliography |
| | Lightweight Corpus |
| CHR22 | human chromosome 22 |
| ETEXT99 | Project Gutenberg ETEXT99 files |
| GCC | gcc 3.0 source (tarred) |
| HOWTO | Linux Howto text files |
| JDK13 | JDK 1.3 doc (html and java files) |
| LINUX | Linux kernel 2.4.5 source (tarred) |
| RCTAIL96 | Reuters news in XML format |
| RFC | RFC text files |
| SPROT | Swiss prot database |
| W3C | html files from www.w3c.org |
| | A pathological string |
| ADVERSARY | $ac^{i^2}acac^2ac^3\ldots ac^ia$, for $i = 16721$ |

**Table 3.1.** Test file descriptions.

Note that the total space used for the suffix tree of string $\mu$ is $25|\mu|$ bytes for the basic version and $33|\mu|$ bytes for version with parent pointers. The latter version is used by `Climb` and `ClimbScan` while `ReScan` can use the former if the suffix tree is not sliding. These implementations can handle a string of length up to $2^{30}$ over a single byte alphabet. However, our tests are limited to strings 400 MiB long to prevent swapping.

We have selected the following three datasets to thoroughly test all three implicit suffix link simulation techniques. All test files and their short descriptions are listed in Table 3.1.

Pizza&Chili Corpus

This corpus was constructed by Ferragina and Navarro for compressed index evaluation [13]. However, as the practical string length limit on our test machine is 400 MiB, we have replaced the two largest files (ENGLISH and PROTEINS) with their 400 MiB prefixes.

Lightweight Corpus

This corpus was originally compiled by Manzini and Ferragina [30] for the purpose of evaluation of suffix array construction algorithms [31].

A pathological string

To illustrate the negative results of Section 3.2.1.3, the following adversary string from the proof of Corollary 3.2 was also included.

$$ac^{i^2}acac^2ac^3...ac^ia, \text{for } i = 16721$$

The value of $i$ was chosen as the largest integer such that the string length does not exceed 400 MiB.

Table 3.2 provides more details about all test files and also includes the following three statistics for each file.

*Inverse probability of matching* (IPM)

This property is defined as the inverse of the probability that two randomly chosen symbols match. It was suggested by Ferragina and Navarro as a measure of the effective alphabet size [13].

*Maximum* and *average LCP*

These are defined as the maximum and average over the lengths of longest common prefixes of pairs of lexicographically consecutive suffixes. They are included as they appear to affect efficiency of certain suffix sorting algorithms [36].

#### 3.2.2.1 Operation Counts

As the operations counts are implementation independent, we will examine them first and leave execution times to the next section. We will count two types of operations the branching operation and the jump-up operation. The first operation is used by both `MoveActivePointDownIfPossible` and `ReScan`, while the latter is used by `Climb` and `ClimbScan`. Note that failed branching attempts are included in the branching operation counts.

As both `MoveActivePointDownIfPossible` and `ReScan` use branching operations, we would like to know how they are split between the two. The answer our experiments gave to this question lies in Table 3.3. Columns `MoveDown`, `ReScan` and Total contain the number of branching operations used by func-

| File | Size | $\lvert\Sigma\rvert$ | IPM | LCP | |
|---|---|---|---|---|---|
| | [MiB] | | | Average | Maximum |
| **Pizza&Chili Corpus** | | | | | |
| SOURCES | 201.10 | 230 | 24.77 | 371.80 | 307 871 |
| PITCHES | 53.25 | 133 | 39.75 | 262.00 | 25 178 |
| PROTEINS.400 | 400.00 | 26 | 17.19 | 654.36 | 263 313 |
| PROTEINS.OLD | 63.71 | 24 | 16.98 | 33.47 | 6 380 |
| DNA | 385.21 | 16 | 3.91 | 2 420.73 | 1 378 596 |
| ENGLISH.400 | 400.00 | 226 | 15.25 | 5 771.85 | 987 770 |
| XML | 282.42 | 97 | 28.73 | 44.91 | 1 084 |
| **Lightweight Corpus** | | | | | |
| CHR22 | 32.95 | 5 | 4.24 | 1 979.25 | 199 999 |
| ETEXT99 | 100.40 | 146 | 15.74 | 1 108.63 | 286 352 |
| GCC | 82.62 | 150 | 21.76 | 8 603.21 | 856 970 |
| HOWTO | 37.60 | 197 | 14.29 | 267.56 | 70 720 |
| JDK13 | 66.50 | 113 | 35.24 | 678.94 | 37 334 |
| LINUX | 110.87 | 256 | 27.12 | 479.00 | 136 035 |
| RCTAIL96 | 109.40 | 93 | 23.27 | 282.07 | 26 597 |
| RFC | 111.03 | 120 | 10.20 | 93.02 | 3 445 |
| SPROT | 104.54 | 66 | 15.41 | 89.08 | 7 373 |
| W3C | 99.37 | 256 | 38.05 | 42 299.75 | 990 053 |
| **A pathological string** | | | | | |
| ADVERSARY | 399.98 | 2 | 1.00 | 93 197 280.11 | 279 591 840 |

**Table 3.2.** Test files properties.

tion `MoveActivePointDownIfPossible`, function `ReScan` and both functions together. The last column then shows the percentage of all branching operations that was used by calls to `ReScan`. It appears that the split of branching operations between functions `MoveActivePointDownIfPossible` and `ReScan` is fairly even with between 40.22 to 66.07 percent used by `ReScan` on all files except the ADVERSARY. Note that `ReScan` takes almost all branching operations used for construction of the suffix tree for ADVERSARY.

The relatively high percentage of branching operation used by `ReScan` shows that there is a lot to be gained by using the bottom-up approach. However, while the jump-up operation should be much cheaper than the branching operation, we do not yet know how many jump-up operations are

| File | Total | `MoveDown` | `ReScan` | $\frac{\texttt{ReScan}}{\text{Total}}$ [%] |
|---|---|---|---|---|
| SOURCES | 300 123 164 | 139 723 039 | 160 400 125 | 53.44 |
| PITCHES | 94 522 634 | 46 777 918 | 47 744 716 | 50.51 |
| PROTEINS.400 | 682 115 039 | 339 775 135 | 342 339 904 | 50.19 |
| PROTEINS.OLD | 123 363 893 | 73 743 357 | 49 620 536 | **40.22** |
| DNA | 759 666 994 | 415 812 497 | 343 854 497 | 45.26 |
| ENGLISH.400 | 688 349 505 | 371 960 100 | 316 389 405 | 45.96 |
| XML | 362 053 458 | 205 256 107 | 156 797 351 | 43.31 |
| CHR22 | 64 622 549 | 35 053 371 | 29 569 178 | 45.76 |
| ETEXT99 | 172 578 102 | 99 131 563 | 73 446 539 | 42.56 |
| GCC | 117 398 104 | 52 626 042 | 64 772 062 | 55.17 |
| HOWTO | 61 266 618 | 32 676 237 | 28 590 381 | 46.67 |
| JDK13 | 78 704 428 | 28 044 490 | 50 659 938 | 64.37 |
| LINUX | 164 899 266 | 76 269 756 | 88 629 510 | 53.75 |
| RCTAIL96 | 145 205 087 | 70 211 436 | 74 993 651 | 51.65 |
| RFC | 166 051 160 | 77 334 572 | 88 716 588 | 53.43 |
| SPROT | 149 117 731 | 78 927 702 | 70 190 029 | 47.07 |
| W3C | 121 167 964 | 41 111 077 | 80 056 887 | **66.07** |
| ADVERSARY | 699 054 845 | 50 166 | 699 004 679 | **99.99** |

**Table 3.3.** A comparison of total numbers of branch operations used by functions `MoveActivePointDownIfPossible`, `ReScan`, and their combination, during the construction of suffix tree for each file. Boldface denotes the maximum and minimum over all files except ADVERSARY.

needed to replace one branching operation on average. The numbers of branching operations used by `ReScan` are compared to the number of jump-up operations used by `Climb` in Table 3.4. These operations counts are located in the second and third column, respectively. The last column then contains the relative percentage of jump-up operations used by `Climb` to replace branching operations used by `ReScan`. Note that on all files except ADVERSARY, the numbers of jumps in `Climb` and the number of branchings in `ReScan` are relatively close and differ by at most 14 percent. The largest excess is reached on CHR22 and DNA, two files with the lowest IPM. On the other hand, a ten percent saving was achieved on PITCHES, the file with the largest IPM. As expected, the situation is much worse for `Climb` on ADVERSARY where it does almost 3.5 of jump-up operations per removed branching operation. However, `Climb` does not show any signs of pathological behaviour on real-life data.

| File | `ReScan` | `Climb` | $\frac{\texttt{Climb}}{\texttt{ReScan}}$ [%] |
|---|---|---|---|
| SOURCES | 160 400 125 | 152 376 916 | 95.00 |
| PITCHES | 47 744 716 | 43 081 419 | **90.23** |
| PROTEINS.400 | 342 339 904 | 312 477 751 | 91.28 |
| PROTEINS.OLD | 49 620 536 | 45 447 883 | 91.59 |
| DNA | 343 854 497 | 389 694 561 | 113.33 |
| ENGLISH.400 | 316 389 405 | 305 930 651 | 96.69 |
| XML | 156 797 351 | 158 979 917 | 101.39 |
| CHR22 | 29 569 178 | 33 669 019 | **113.87** |
| ETEXT99 | 73 446 539 | 74 097 636 | 100.89 |
| GCC | 64 772 062 | 61 849 248 | 95.49 |
| HOWTO | 28 590 381 | 27 944 722 | 97.74 |
| JDK13 | 50 659 938 | 49 413 385 | 97.54 |
| LINUX | 88 629 510 | 83 753 439 | 94.50 |
| RCTAIL96 | 74 993 651 | 72 128 546 | 96.18 |
| RFC | 88 716 588 | 84 486 767 | 95.23 |
| SPROT | 70 190 029 | 69 425 197 | 98.91 |
| W3C | 80 056 887 | 75 904 742 | 94.81 |
| ADVERSARY | 699 004 679 | 2 410 330 433 | **344.82** |

**Table 3.4.** A comparison of total numbers of branch operations used by function `ReScan` and jump-up operations used by function `Climb`, during the construction of suffix tree for each file. Boldface denotes the maximum and minimum over all files except ADVERSARY.

While we know the total operation counts, we do not have any information about their distribution in individual simulations. Table 3.5 contains more detailed statistics. The second column contains the maximal number of branching operation used in a single implicit suffix link simulation done by `ReScan`. Similarly, the third column has the maximum consecutive number of jump-up operations done during `Climb`. The corresponding average values are placed in the fourth and the fifth column, respectively. Note that the minimum and median values were computed as well, but were all equal to 1 with exception of median of branching values used on ADVERSARY which was equal to 2. If we ignore the expected huge difference on ADVERSARY, there is only one row where the longest sequence of branching operations is much shorter than the the longest sequence of jump-up operations. Specifically, on file SOURCES, the longest jump-up sequence is 2858 jumps long, but the maximum for branchings is just 123.

| File | Maximum | | Average | |
|---|---|---|---|---|
| | `ReScan` | `Climb` | `ReScan` | `Climb` |
| SOURCES | 123 | 2 858 | 1.21 | 1.15 |
| PITCHES | 1 656 | 3 999 | 1.33 | 1.20 |
| PROTEINS.400 | 447 | 372 | 1.26 | 1.15 |
| PROTEINS.OLD | 92 | 68 | **1.35** | 1.24 |
| DNA | 128 | 135 | 1.25 | 1.41 |
| ENGLISH.400 | 60 | 57 | 1.30 | 1.26 |
| XML | **24** | 43 | 1.10 | 1.12 |
| CHR22 | **99 998** | **199 999** | 1.26 | **1.43** |
| ETEXT99 | 90 | 80 | 1.27 | 1.28 |
| GCC | 11 078 | 11 064 | 1.18 | 1.13 |
| HOWTO | 73 | 69 | 1.15 | 1.22 |
| JDK13 | 62 | 57 | **1.07** | **1.04** |
| LINUX | 442 | 475 | 1.22 | 1.15 |
| RCTAIL96 | 63 | 62 | 1.14 | 1.10 |
| RFC | 96 | 99 | 1.22 | 1.16 |
| SPROT | 28 | **36** | 1.13 | 1.12 |
| W3C | 219 | 298 | 1.10 | **1.04** |
| ADVERSARY | 2 | 279 591 840 | 1.67 | 5.75 |

**Table 3.5.** A comparison of sequences of consecutive branch operations used by function `ReScan` and jump-up operations used by function `Climb`, during the construction of suffix tree for each file. The median of `ReScan` for ADVERSARY was equal to 2 while all the remaining median and minimum values were always equal to 1. Boldface denotes the maximum and minimum over all files except ADVERSARY.

However, other columns show that this spike is not common and is levelled out by many shorter jump-up sequences. Like the total operation counts, these detailed results show that while it is possible to force `Climb` to perform a large number of jumps, this is not common on real-life data.

The last question about operation counts that we will try to answer is that about the balance achieved by `ClimbScan`. The numbers of branching and jump-up operations used by `ClimbScan` are compared to operations used by the other two techniques in Table 3.6. The second column provides the number of jump-up operations used by `ClimbScan`, while the third gives the same for branching operations. Note that the number of branching operations is negligible compared to the number of jump-up operations. Hence, like `Climb`,

| File | ClimbScan | | $\frac{\texttt{ClimbScan}}{\texttt{Climb}}$ [%] | $\frac{\texttt{ClimbScan}}{\texttt{Rescan}}$ [%] |
|---|---|---|---|---|
| | Jump | Branch | | |
| SOURCES | 152 291 681 | 23 176 | 99.96 | 94.96 |
| PITCHES | 42 678 609 | 26 862 | 99.13 | **89.45** |
| PROTEINS.400 | 312 044 626 | 53 580 | 99.88 | 91.17 |
| PROTEINS.OLD | 45 438 450 | 2 368 | 99.98 | 91.58 |
| DNA | 389 042 535 | 127 743 | 99.87 | **113.18** |
| ENGLISH.400 | 305 876 547 | 26 763 | 99.99 | 96.69 |
| XML | 158 969 534 | 5 135 | **100.00** | 101.39 |
| CHR22 | 33 224 531 | 10 193 | **98.71** | 112.40 |
| ETEXT99 | 74 086 107 | 5 748 | 99.99 | 100.88 |
| GCC | 61 803 182 | 8 662 | 99.94 | 95.43 |
| HOWTO | 27 930 198 | 5 066 | 99.97 | 97.71 |
| JDK13 | 49 407 831 | 1 852 | 99.99 | 97.53 |
| LINUX | 83 687 734 | 14 656 | 99.94 | 94.44 |
| RCTAIL96 | 72 120 830 | 3 560 | 99.99 | 96.17 |
| RFC | 84 381 579 | 25 521 | 99.91 | 95.14 |
| SPROT | 69 420 291 | 2 036 | **100.00** | 98.91 |
| W3C | 75 891 132 | 3 149 | 99.99 | 94.80 |
| ADVERSARY | 419 529 898 | 16 722 | 17.41 | 60.02 |

**Table 3.6.** A comparison of total numbers of jump-up and branch operations used by functions `ClimbScan`, `Climb` and `ReScan` during the construction of suffix tree for each file. Boldface denotes the maximum and minimum over all files except ADVERSARY.

`ClimbScan` removes a lot of branching operations from the construction of suffix tree. Moreover, the total number of operations used by `ClimbScan` compares favourably to the numbers of operations used by `Climb` and `ReScan` as demonstrated by the last two columns of Table 3.6. The fourth column shows that `ClimbScan` never used more operations than `Climb` does and even does over 82 percent operations less on ADVERSARY. This is a pleasant surprise which shows that `Climb` can handle most situations by itself and only rarely needs a help from `ReScan`. As the numbers of operations of `Climb` and `ClimbScan` are so close, it is no surprise that the last column of Table 3.6 is very similar to the last column of Table 3.4. The only significant difference is in the ADVERSARY row, where `ClimbScan` needs almost 40 percent operation less than `ReScan`.

Before we turn our attention to practical speed tests we can estimate the outcome based on our operation count analysis. As both bottom-up methods remove so many branching operations, they are both expected to be significantly faster than ReScan. Moreover, as Climb is simpler than ClimbScan and pathological situation are not common, the Climb should prevail on most inputs.

#### 3.2.2.2 Construction Time

While the previous section concentrated on implementation independent operation counts, this section deals with execution times which are inherently implementation dependent. To put the performance of our implementation into perspective, we have included an independent alternative implementation in our tests. We have selected a tuned suffix tree implementation developed by Kurtz [25] which is available as a part of the MUMmer3 package, a system for genome alignment [24]. This implementation uses McCreight's algorithm with ReScan for implicit suffix link simulation and an improved linked list implementation introduced by Kurtz [25], this leads to memory usage of 12.18$n$ bytes on average.

The first part of our speed tests was testing construction of the suffix tree only, while the second part included maintenance of the suffix tree for a sliding window. Note that execution times presented here are averages of five independent runs.

The execution times collected for the construction of the suffix tree for each file are shown in Table 3.7. The two dashes in the last column denote the fact that Kurtz's implementation failed on files DNA and ADVERSARY.

In order to verify that our implementations are competitive with the state-of-the-art implementation of on-line construction of suffix trees, the speeds of our implementations are compared to those of the implementation developed by Kurtz [25, 24]. Observe that while the simple version of ReScan needs as much as 28 percent more time for construction than Kurtz's implementation, the move-to-front Climb is requires 20 to 35 percent time less. Thus our implementations are indeed competitive and the rest of this section can focus on the comparison of our three implementations.

Note that ClimbScan with the move-to-front heuristic provides the fastest construction in the majority of cases. With the exception of the ADVERSARY file, a comparison of the second and the fourth column shows that the reduction of the number of branching operations described in Section 3.2.2.1 translates into a 21 to 32 percent shorter construction time. Note that the lowest speed up is obtained on files with the smallest alphabet (CHR22, DNA) and that files with a large alphabet size (SOURCES, ENGLISH.400, LINUX, W3C) are close to the maximum increase. This can be explained by the connection between the alphabet size and the out-degrees of nodes in the graph that influences the total time needed for branching.

The third and fifth columns show that the move-to-front version of Climb saves between 14 and 24 percent of ReScan execution time. These lower values, compared to the simple case, may be explained by the speed up of branching operations brought by the more sophisticated linked lists manipulation, which diminishes the advantage of climbing.

| File | ReScan |  | Climb |  | ClimbScan |  | Kurtz |
|---|---|---|---|---|---|---|---|
|  | simple | mtf | simple | mtf | simple | mtf |  |
| SOURCES | **196.19** | 140.42 | 133.20 | 113.70 | 133.40 | **113.48** | 175.77 |
| PITCHES | **113.54** | 74.24 | 78.84 | **61.53** | 78.90 | 61.63 | 88.47 |
| PROTEINS.400 | **813.39** | 714.30 | 560.57 | 541.32 | 562.69 | **541.00** | 744.62 |
| PROTEINS.OLD | **143.31** | 120.88 | 105.13 | 101.65 | 104.93 | **100.54** | 132.68 |
| DNA | **368.00** | 366.85 | **279.78** | 281.79 | 279.96 | 281.33 | — |
| ENGLISH.400 | **605.54** | 503.61 | 417.83 | 389.88 | 418.28 | **389.83** | 575.02 |
| XML | **152.08** | 128.72 | 111.04 | **103.41** | 110.99 | 103.66 | 134.09 |
| CHR22 | 23.91 | **23.92** | 18.89 | 18.88 | **18.86** | 19.00 | 23.67 |
| ETEXT99 | **132.57** | 111.28 | 94.98 | 88.80 | 95.12 | **88.52** | 122.57 |
| GCC | **58.50** | 42.61 | 40.74 | **35.10** | 40.72 | 35.14 | 53.55 |
| HOWTO | **46.27** | 34.50 | 32.18 | 27.63 | 32.09 | **27.58** | 40.22 |
| JDK13 | **13.78** | 11.47 | 10.30 | 9.70 | 10.26 | **9.64** | 13.62 |
| LINUX | **101.70** | 71.83 | 69.18 | 58.67 | 69.36 | **58.48** | 89.32 |
| RCTAIL96 | **66.44** | 55.33 | 46.61 | 43.45 | 46.62 | **43.44** | 61.82 |
| RFC | **102.49** | 79.02 | 69.98 | 62.27 | 69.96 | **62.16** | 92.05 |
| SPROT | **103.73** | 87.81 | 76.84 | **72.08** | 76.72 | 72.12 | 93.70 |
| W3C | **38.40** | 28.34 | 27.00 | 24.48 | 27.10 | **24.22** | 36.83 |
| ADVERSARY | **11.24** | 11.25 | **22 454.57** | 22 387.51 | 11.85 | 11.68 | — |

**Table 3.7.** A comparison of execution times (in seconds) needed for the construction of suffix tree for each test file. Boldface indicates the maximum and minimum in each row.

A comparison of ClimbScan and ReScan columns reveals that the advantage of ClimbScan over ReScan on real-life data is almost identical to that of Climb, the difference being within one percent of ReScan execution time. However, as ClimbScan avoids the pathological behaviour on ADVERSARY, it exceeds the ReScan execution time on this file by less than 5 percent.

The end of this section inspects the efficiency of the move-to-front heuristic using the simple and mtf columns in Table 3.7 for each of the three techniques. The comparison reveals that this heuristic brings no speedup or even a small slowdown (by less than 1 percent) when tested on files with the smallest IPM, e.g. DNA, CHR22 and ADVERSARY. However, it works well on all other test files, and the highest reduction in construction time happens on PITCHES, the file with the highest IPM. Here the move-to-front ReScan needs 35 percent less time for construction, while the move-to-front Climb / ClimbScan saves 22 percent of execution time.

| File | Window | ReScan | | Climb | | ClimbScan | |
|---|---|---|---|---|---|---|---|
| | [KiB] | simple | mtf | simple | mtf | simple | mtf |
| SOURCES | 32 | **80.29** | 74.64 | 69.61 | **68.70** | 69.88 | 69.04 |
| | 1 024 | **247.72** | 214.20 | 200.57 | **190.94** | 200.25 | 191.63 |
| | 32 768 | **331.92** | 275.39 | 257.01 | **240.12** | 256.85 | 240.74 |
| PITCHES | 32 | **20.84** | 18.22 | 17.73 | 17.00 | 17.57 | **16.81** |
| | 1 024 | **96.57** | 73.45 | 76.44 | **66.84** | 76.35 | 67.02 |
| | 32 768 | **142.08** | 99.36 | 101.90 | **84.53** | 101.97 | 84.57 |
| PROTEINS | 32 | **712.72** | 694.00 | **635.26** | 638.36 | 636.84 | 637.97 |
| | 1 024 | **2 617.50** | 2 420.66 | 2 091.66 | **2 062.04** | 2 086.25 | 2 067.95 |
| | 32 768 | **3 901.03** | 3 559.85 | 3 021.12 | 2 979.15 | 3 014.51 | **2 978.78** |
| PROTEINS.OLD | 32 | **33.36** | 31.93 | 29.17 | **29.11** | 29.33 | 29.46 |
| | 1 024 | **134.61** | 117.09 | 105.67 | 102.98 | 105.67 | **102.77** |
| | 32 768 | **180.88** | 154.97 | 136.03 | 131.34 | 135.97 | **131.26** |
| DNA | 32 | 176.26 | **177.20** | **156.96** | 159.74 | 157.37 | 159.28 |
| | 1 024 | **446.30** | 441.82 | **372.67** | 376.11 | 372.88 | 375.74 |
| | 32 768 | **683.78** | 680.88 | 573.06 | 576.58 | **572.32** | 578.68 |
| ENGLISH | 32 | **1 159.61** | 1 108.80 | 1 007.41 | **1 006.69** | 1 015.78 | 1 007.35 |
| | 1 024 | **3 539.73** | 3 250.21 | 2 819.08 | 2 785.70 | 2 818.41 | **2 779.35** |
| | 32 768 | **5 559.53** | 5 082.22 | 4 415.61 | **4 355.70** | 4 414.26 | 4 358.73 |
| XML | 32 | **85.58** | 83.15 | 79.18 | 78.16 | 78.65 | **78.05** |
| | 1 024 | **297.40** | 273.71 | 253.23 | **246.57** | 252.53 | 247.04 |
| | 32 768 | **349.09** | 320.78 | 291.12 | **285.11** | 290.29 | 285.27 |

**Table 3.8.** A comparison of execution times (in seconds) needed for the sliding of suffix tree over each file of the Pizza&Chili Corpus. Boldface indicates the maximum and minimum in each row.

#### 3.2.2.3 Sliding Window

The last set of experiments concentrates on the speed of a sliding suffix tree. As noted in Section 3.2.1.1, the additional leaf suffix links and reverse edges that are required to enable `Climb` and `ClimbScan` are already present in a sliding suffix tree. Consequently, `ReScan` no longer has a space advantage over `Climb` and `ClimbScan` and might be even slower. However, as will be explained in Section 4.1, there are actions required by sliding that take part of the execution time and disturb the construction algorithm flow. This may result in a somewhat lower gain from the bottom-up approach.

| File | Window | `ReScan` | | `Climb` | | `ClimbScan` | |
|---|---|---|---|---|---|---|---|
| | [KiB] | simple | mtf | simple | mtf | simple | mtf |
| | 32 | 15.44 | **15.54** | **13.75** | 13.92 | 13.77 | 13.92 |
| chr22 | 1 024 | **36.20** | 35.80 | **30.42** | 30.60 | 30.43 | 30.57 |
| | 32 768 | **34.29** | 34.06 | **27.71** | 27.88 | **27.71** | 27.79 |
| | 32 | **50.21** | 47.77 | 43.45 | **43.43** | 43.70 | 43.68 |
| etext99 | 1 024 | **161.24** | 148.86 | 128.12 | **126.51** | 127.76 | 126.60 |
| | 32 768 | **202.66** | 181.38 | 157.04 | **153.01** | 156.98 | 153.04 |
| | 32 | **31.70** | 29.89 | 27.66 | 27.33 | **27.25** | 27.47 |
| gcc | 1 024 | **91.18** | 81.32 | 75.31 | 72.58 | 75.10 | **72.54** |
| | 32 768 | **94.63** | 77.78 | 72.46 | **67.28** | 72.50 | 67.47 |
| | 32 | **18.32** | 17.27 | 15.86 | 15.59 | 15.64 | **15.36** |
| howto | 1 024 | **63.42** | 54.03 | 50.20 | **47.23** | 50.01 | 47.36 |
| | 32 768 | **58.65** | 46.15 | 41.79 | 37.38 | 41.84 | **37.36** |
| | 32 | **19.98** | 19.33 | 17.78 | **17.60** | 17.64 | 17.70 |
| jdk13 | 1 024 | **46.65** | 44.34 | 41.50 | **40.89** | 41.41 | 40.95 |
| | 32 768 | **33.28** | 30.68 | 28.28 | **27.65** | 28.29 | 27.72 |
| | 32 | **43.91** | 40.60 | 37.75 | 37.07 | 37.45 | **36.92** |
| linux | 1 024 | **134.26** | 116.91 | 109.20 | 103.96 | 109.07 | **103.95** |
| | 32 768 | **160.79** | 128.07 | 120.82 | **110.75** | 120.81 | 110.99 |
| | 32 | **39.47** | 37.45 | 34.97 | **34.42** | 34.85 | 34.43 |
| rctail96 | 1 024 | **126.33** | 116.67 | 105.50 | **103.77** | 105.18 | 103.85 |
| | 32 768 | **119.44** | 107.86 | 94.50 | **92.05** | 94.16 | 92.17 |
| | 32 | **45.96** | 43.24 | 39.72 | 39.70 | 39.65 | **39.35** |
| rfc | 1 024 | **148.42** | 131.08 | 120.85 | 116.47 | 120.61 | **116.37** |
| | 32 768 | **167.36** | 143.03 | 127.13 | **120.86** | 127.43 | 121.03 |
| | 32 | **43.99** | 41.67 | 38.28 | 37.99 | 38.36 | **37.84** |
| sprot | 1 024 | **139.93** | 125.58 | 115.14 | **111.81** | 114.87 | 112.17 |
| | 32 768 | **160.27** | 142.31 | 127.67 | **123.69** | 127.58 | 123.82 |
| | 32 | **34.17** | 32.05 | 29.56 | **29.17** | 29.84 | 29.61 |
| w3c | 1 024 | **94.64** | 87.27 | 79.84 | **78.11** | 79.71 | 78.51 |
| | 32 768 | **77.87** | 65.19 | 61.81 | **58.60** | 61.83 | 58.71 |

**Table 3.9.** A comparison of execution times (in seconds) needed for the sliding of suffix tree over each file of the Lightweight Corpus. Boldface indicates the maximum and minimum in each row.

To reveal more information about sliding behaviour, three window sizes were used during these experiments. Similar to window/block sizes used by the data compression algorithms `gzip`, `bzip2` and `xz`, the following window sizes were used: 32 KiB, 1 MiB and 32 MiB. Thanks to these limits, the tests on Pizza&Chilli Corpus files PROTEINS and ENGLISH could be run at their original sizes, i.e. 1.10 GiB and 2.06 GiB, respectively. Recall that only 400 MiB prefixes were used in previous experiments. Each experiment involved maintaining a suffix tree for a perfect sliding window of a given size over the input.

The averages of execution times of three independent runs are shown in Tables 3.8 and 3.9. Note that `Climb` with the move-to-front heuristic provides the fastest sliding in the majority of cases. As the execution time differences between `Climb` and `ClimbScan` are under 2 percent, the rest of this comparison is limited to `ReScan` and `Climb` only.

The advantage of the simple version of `Climb` over the simple version of `ReScan` grows with the sliding window size. `Climb` saves 7 to 14 percent of execution time with a 32 KiB window, 11 to 21 percent with a 1 MiB window, and 15 to 28 percent with a 32 MiB window. This effect is likely connected to the increase in size of sliding suffix trees in combination with more branching choices in such trees. Nevertheless, the advantage of `Climb` over `ReScan` is always smaller than without sliding, even with the largest window. This is likely the result of smaller tree sizes and the adverse effect of additional code needed to delete oldest symbols and maintain edge labels that slows all algorithms down and uses almost no implicit suffix links.

Finally, note that when the move-to-front variants are compared, the savings introduced by climbing decrease to 6 to 10 percent for a 32 KiB window, 8 to 15 percent for a 1 MiB window, and 10 to 19 percent for a 32 MiB window. These lower values are consistent with the results of construction tests and support the conclusions on this issue formulated in Section 3.2.2.2.

# 4. Sliding

This chapter represents the last step, before we finally get to the details of the suffix graph based data compression. It builds on previous chapters and studies the problem of suffix graph maintenance for a sliding window.

While certainly interesting, the problem of suffix graph sliding received much less attention than the construction. Understandably, there appears to be no interest in sliding of the suffix trie. The other three structures are much more appealing. Surprisingly, the sliding of the DAWG was resolved first [8]. Unfortunately, it turned out that it is impossible to make it slide in amortised constant time. Moreover, as we show in Section 4.2, the behaviour of the CDAWG is the same. Inenaga et al. [21] appeared to have solved this issue with their approximate sliding algorithm, but it was later shown to be invalid [15]. Hence, the suffix tree is our last chance for linear time sliding. The first algorithm for suffix tree sliding was presented by Fiala and Greene in 1989 [14]. However, while they found the components needed for linear time sliding, their paper had several issues. This was pointed out by Larsson, who tried to clarify the situation and fix these problems [27]. Nevertheless, despite his attempt, the largest issue regarding the correctness of the edge label maintenance algorithm was not resolved in a satisfactory way. We settle this issue in Section 4.3.

This chapter consists of three sections that deal with the sliding of suffix graphs in more detail. The first section brings an overview of sliding and sliding algorithm based on the construction algorithm studied in Chapter 3: Construction. It is followed by two sections that analyse the details of two functions needed to complete the sliding algorithm. Note that all three sections are based on our previously published works [40] and [44]. The first two sections are based on the former, while the third section is based on the latter.

## 4.1 Basics

This section gives an overview of the suffix graph sliding problem and a sliding algorithm devised from the construction algorithm described in Chapter 3: Construction. It starts with the problem definition, using the following two basic operations.

**Definition (Sliding Operations)**
Let $a, c \in \Sigma$ and $\mu \in \Sigma^*$, then the two basic sliding operations are defined as follows:

```
void Slide(String & string, Size windowSize)
{
    Build(string[1, windowSize]);
    SlideOverString(string[windowSize + 1, Length(string)]);
}
void SlideOverString(String & string)
{
    for (symbol in string) {
        SlideOverSymbol(symbol);
    }
}
void SlideOverSymbol(Symbol & symbol)
{
    DeleteOldestSymbol();
    AppendSymbol(symbol);
    UpdateEdgeLabels();
}
```

**Algorithm 4.1.** Outline of sliding window algorithm that works on all four suffix graph types.

`Delete`
: This operation receives a suffix graph for string $c\mu$ and returns a suffix graph of the same type for $\mu$.

`Append`
: This operation receives a suffix graph for string $\mu$ and returns a suffix graph of the same type for string $\mu a$.

The perfect form of sliding is then defined as follows.

**Definition (Perfect Sliding)**
The input and output of any algorithm implementing the perfect sliding of a suffix graph must be:

INPUT
: A string $\mu \in \Sigma^*$ and an integer $w$, called *window length* from now on, such that $0 < w < |\mu|$.

OUTPUT
: A sequence $G_1, G_2, \ldots, G_{|\mu|-w+1}$, where $G_1$ is a suffix graph for string $\mu[1\,..\,w]$ and $G_{i+1} = \mathsf{Append}(\mathtt{Delete}(G_i), \mu[i+w]) = \mu[i+1\,..\,i+w]$.

An obvious solution to the problem of perfect sliding is a trivial "scrap and build" method. In essence, this algorithm implements operation `Delete` by scraping the old graph and operation Append by building a new graph. While this works for any suffix graph, it is slow. Moreover, some algorithms, including those in Chapter 5: Compression, collect valuable information for vertices and edges, such as usage statistics. When both vertices and edges are removed, this information is lost. Even though it could be partially regenerated during the build phase, this would make the algorithm even slower. Consequently, a more sophisticated solution is needed.

Such algorithm should be based on one of the left-to-right construction algorithm mentioned in Chapter 3: Construction. This construction algorithm would be used for the implementation of `Append` operation and complemented by some form of `Delete` operation. A pseudo-C++ code of one possible solution is shown in Algorithm 4.1. It is using parts from the Ukkonen-based algorithm described in Section 3.1 and shown in Algorithm 3.1. Functions that were not part of the `Build` algorithm are described below.

`Slide`
: This is the main function of the sliding algorithm. As required by the perfect sliding definition, it builds the suffix graph for the first `windowSize` symbols of the input string first. After that, it runs the `SlideOverString` function to do the actual sliding over the rest of the input string.

`SlideOverString`
: As hinted by its name, this function slides the suffix graph over the supplied string. It simply calls the function `SlideOverSymbol` for every symbol of this string.

`SlideOverSymbol`
: This function performs the equivalent of application of the combination of operations `Append(Delete)` on the suffix graph. It is using the function `DeleteOldestSymbol` to perform the `Delete` operation, and the other two functions to implement the `Append` operation. The first part of `Append` operation is done using function `AppendSymbol` and the second part by function `UpdateEdgeLabels`.

`DeleteOldestSymbol`
: As described by its name, this function removes the oldest (leftmost) symbol from the underlying string and adjusts the suffix graph accordingly. While it sounds easy enough, it turns out to be quite a complex matter. It is studied in detail in Section 4.2.

`UpdateEdgeLabels`
: This function finishes the `Append` operation by adjusting the edge labels. As both the suffix trie and the DAWG have only single symbol labels, which are stored explicitly, there is no work to be done for these. However, the longer labels used in both path compressed graphs require the use of indexes into the underlying string. Consequently, these indexes must be

regularly updated to stay valid. The details of an efficient implementation of this function were put into Section 4.3.

It is clear that functions `DeleteOldestSymbol` and `UpdateEdgeLabels` are critical for the performance of this sliding algorithm. As they are nontrivial, each has its own dedicated section. Section 4.2 deals with the details of function `DeleteOldestSymbol`, while Section 4.3 handles the obstacles met by function `UpdateEdgeLabels`.

## 4.2 Delete Oldest Symbol

The purpose of this section is to find an efficient algorithm that removes the oldest symbol from a suffix graph, which is suitable for use in function `DeleteO-ldestSymbol`. It starts by examining the effects that the removal of the oldest symbol from the underlying string has on each suffix graph type.

### 4.2.1 Changes

As all suffix graphs represent the set of all factors of the underlying string, it is natural to look at differences between sets of factors of strings $c\mu$ and $\mu$ first. They are described by the following claim.

**Claim 4.1 (Factor Set Changes on Delete)**
Let $\mu \in \Sigma^*$ and $c \in \Sigma$, then

$$\text{Factor}(c\mu) = \text{Factor}(\mu) \mathbin{\dot\cup} \text{UniquePrefix}(c\mu) \ .$$

Thus, the changes are somewhat similar to those addressed by function `AppendSymbol` during construction. However, where function `AppendSymbol` dealt with suffixes, function `DeleteOldestSymbol` has to handle prefixes. Also, while suffix links had to be added to form a boundary path for easier movement between suffixes, all prefixes are located on the backbone which is a natural part of the graph. Moreover, as all prefixes longer than some unique prefix must be unique as well, all unique prefixes are grouped as explained by the following observation.

**Claim 4.2 (Prefix Blocks)**
Let $\mu \in \Sigma^*$. Then there exists an integer $k$, $0 \le k \le |\mu|$, such that

$$\text{UniquePrefix}(\mu) = \left\{ \alpha \;\middle|\; \alpha \in \text{Prefix}(\mu) \land |\alpha| \ge k \right\} \ .$$

Hence, the prefixes represented by the end of the backbone should be removed from the suffix graph. This information might be sufficient to devise a delete algorithm for the suffix trie, but more is needed for other suffix graph types. Two of these graph types use the path compression to save space. Recall that the path compression depends on the concept of right extension, and it in turn relies on the set of explicit factors. The set of explicit factors then contains the empty string, all right branching factors and all unique suffixes of the underlying string. While the empty string will not change, the sets of unique suffixes and right branching factors might. The following two claims are based on Larsson's [27] analysis of the problem and show the changes that deletion introduces into these two sets.

**Claim 4.3 (Right Branching Factor Set Changes on Delete [27])**
Let $\mu \in \Sigma^*$ and $c \in \Sigma$, then

$$\begin{aligned}\mathrm{Branch}^{\mathrm{R}}(c\mu) = {} & \mathrm{Branch}^{\mathrm{R}}(\mu) \\ & \dot{\cup} \left\{ \alpha \;\middle|\; \mathrm{Context}^{\mathrm{R}}_{c\mu}(\alpha) = \{a, o\} \wedge a \neq o \wedge \alpha a \in \mathrm{UniquePrefix}(c\mu) \right\} .\end{aligned}$$

Thus, there are either no differences in branching factors or one factor ceases to be branching. This factor is the longest non-unique prefix of $c\mu$, which is also right branching and has only two right contexts.

**Claim 4.4 (Unique Suffix Set Changes on Delete [27])**
Let $\mu \in \Sigma^*$ and $c \in \Sigma$, then

$$\begin{aligned}\mathrm{UniqueSuffix}(\mu) \dot{\cup} \{c\mu\} &= \mathrm{UniqueSuffix}(c\mu) \\ &\quad \dot{\cup} \left\{ \alpha \;\middle|\; \alpha \in \mathrm{Suffix}(c\mu) \cap \mathrm{Prefix}(c\mu) \wedge \left|\mathrm{Occur}^{\mathrm{R}}_{c\mu}(\alpha)\right| = 2 \right\} \\ &= \mathrm{UniqueSuffix}(c\mu) \dot{\cup} \left( \left\{ \mathrm{LNUP}(c\mu) \right\} \cap \left\{ \mathrm{LNUS}(c\mu) \right\} \right) .\end{aligned}$$

Like in the case of the set of right branching factors, the changes to the set of unique suffixes are small. One unique suffix is lost and one may be gained. The lost unique suffix is the string $c\mu$ itself. On the other hand, the new unique suffix must be both the longest non-unique prefix and the longest non-unique suffix at the same time. As the active point is located at the longest non-unique suffix after `AppendSymbol`, this happens if and only if the active point lies on the last edge of the backbone.

Changes that result from operation `Delete` appear to be trivial so far. But what about the second technique used to reduce the space requirements? The vertex minimisation depends on the right end equivalence and its classes. As all prefixes of $c\mu$ lose one right occurrence and no other factors do, it is possible that all right end equivalence classes on the backbone change. This need further investigation.

To make the discussion easier, all non-degenerated classes are split into several distinct class types. These class types and their properties are shown in Table 4.1. Clearly, *Type 0* holds all classes outside the backbone, while the backbone classes are split into seven different types. The following lemma which is similar to those of Blumer [8] and Senft and Dvořák [40] describes changes to all

| Type of $[\alpha]^{\mathrm{R}}_{c\mu}$ | Prefix$(c\mu)$ | $\lvert\mathrm{Occur}^{\mathrm{R}}_{c\mu}(\alpha)\rvert$ | $\lvert\mathrm{Context}^{\mathrm{L}}_{c\mu}(\alpha)\rvert$ | Prefix$(\mu)$ | $\lvert[\alpha]^{\mathrm{R}}_{c\mu}\rvert$ |
|---|---|---|---|---|---|
| *Type 0* | $\alpha \notin$ | ? | ? | ? | ? |
| *Type 1* | $\alpha \in$ | $= 1$ | $(= 0)$ | $(\beta \notin)$ | $> 1$ |
| *Type 2* | | | | | $= 1$ |
| *Type 3* | | $> 1$ | $= 1$ | $(\beta \notin)$ | $> 1$ |
| *Type 4* | | | | $\beta \notin$ | $= 1$ |
| *Type 5* | | | | $\beta \in$ | $(= 1)$ |
| *Type 6* | | | $> 1$ | $(\beta \notin)$ | $> 1$ |
| *Type 7* | | | | ? | $= 1$ |

**Table 4.1.** Right end equivalence class types for class $[\alpha]^{\mathrm{R}}_{c\mu}$, where $\mu \in \Sigma^*$, $c \in \Sigma$, $\alpha \in \mathrm{Factor}(c\mu)$, $\alpha = (\alpha)^{\mathrm{R}}_{c\mu}$ and $\beta = \alpha[2 \mathinner{..} |\alpha|]$. Parens mark values that are derived from other properties and question marks stand for unknown values.

eight class types. Note that Blumer's version of this lemma does not distinguish between *Type 4* and *Type 5*, and overall does not handle strings that are prefixes of both $c\mu$ and $\mu$ well.

**Lemma 4.5 (Right End Equivalence Class Changes on Delete [40])**
Let $\mu \in \Sigma^*$, $c \in \Sigma$, $\alpha \in \mathrm{Factor}(c\mu)$, $\alpha = (\alpha)^{\mathrm{R}}_{c\mu}$, $\beta = \alpha[2 \mathinner{..} |\alpha|]$, and $\gamma$ be the longest suffix of $\alpha$ not in $[\alpha]^{\mathrm{R}}_{c\mu}$. In this situation, when $[\alpha]^{\mathrm{R}}_{c\mu}$ is of

*Type 0* then $[\alpha]^{\mathrm{R}}_{\mu} = [\alpha]^{\mathrm{R}}_{c\mu} \mathbin{\dot\cup} \{\gamma\}$ if and only if $[\gamma]^{\mathrm{R}}_{c\mu}$ is of *Type 3* or *Type 4*. Otherwise, $[\alpha]^{\mathrm{R}}_{\mu} = [\alpha]^{\mathrm{R}}_{c\mu}$.

*Type 1* then $[\beta]^{\mathrm{R}}_{\mu} = [\alpha]^{\mathrm{R}}_{c\mu} \setminus \{\alpha\}$ and $[\alpha]^{\mathrm{R}}_{\mu} = \mathrm{DC}^{\mathrm{R}}_{\mu}$.

*Type 2* then $[\alpha]^{\mathrm{R}}_{\mu} = \mathrm{DC}^{\mathrm{R}}_{\mu}$.

*Type 3* then $[\beta]^{\mathrm{R}}_{\mu} = [\alpha]^{\mathrm{R}}_{c\mu} \setminus \{\alpha\}$, $\mathrm{Context}^{\mathrm{L}}_{c\mu}(\alpha) = \{a\}$ and $[\alpha]^{\mathrm{R}}_{\mu} = [a\alpha]^{\mathrm{R}}_{\mu}$.

*Type 4* then $\mathrm{Context}^{\mathrm{L}}_{c\mu}(\alpha) = \{a\}$ and $[\alpha]^{\mathrm{R}}_{\mu} = [a\alpha]^{\mathrm{R}}_{\mu}$.

*Type 5* then $\mathrm{Context}^{\mathrm{L}}_{c\mu}(\alpha) = \{a\}$, $a = c$, $\alpha = c^k$ and $[\alpha]^{\mathrm{R}}_{\mu} = [\alpha]^{\mathrm{R}}_{c\mu} = \{\alpha\}$.

*Type 6* then $[\beta]^{\mathrm{R}}_{\mu} = [\alpha]^{\mathrm{R}}_{c\mu} \setminus \{\alpha\}$ and $[\alpha]^{\mathrm{R}}_{\mu} = \{\alpha\}$.

*Type 7* then $[\alpha]^{\mathrm{R}}_{\mu} = [\alpha]^{\mathrm{R}}_{c\mu} = \{\alpha\}$.

| Type | | $[\alpha\beta]^{\mathrm{R}}_{c\mu}$ | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | *Type 1* | *Type 2* | *Type 3* | *Type 4* | *Type 5* | *Type 6* | *Type 7* |
| | *Type 1* | + | | | | | | |
| | *Type 2* | + | + | | | | | |
| | *Type 3* | + | | + | | | | |
| $[\alpha]^{\mathrm{R}}_{c\mu}$ | *Type 4* | + | + | + | + | | | |
| | *Type 5* | + | + | + | + | + | | |
| | *Type 6* | + | | + | | | + | |
| | *Type 7* | + | + | + | + | + | + | + |

**Table 4.2.** Dependencies between the right end equivalence class types of $\alpha$ and $\alpha\beta$ on $c\mu$. Valid class type combinations are marked using +.

While the lemma above describes what changes happen to each class type, there is not much information that could be used to identify class types of vertices during operation `Delete`. The following claim solves part of this problem as it describes partial ordering of class types on the backbone as shown in Table 4.2. Note that the first two items are straightforward consequences of definitions of occurrence and context, while the third item is easily obtained from Claim 2.12.

**Claim 4.6 (Right End Equivalence Class Type Dependencies)**
Let $\alpha\beta \in \mathrm{Factor}(\mu)$, then

(i) $|\mathrm{Occur}^{\mathrm{R}}_{\mu}(\alpha)| \geq |\mathrm{Occur}^{\mathrm{R}}_{\mu}(\alpha\beta)|$,

(ii) $|\mathrm{Context}^{\mathrm{L}}_{\mu}(\alpha)| \geq |\mathrm{Context}^{\mathrm{L}}_{\mu}(\alpha\beta)|$,

(iii) $|[\alpha]^{\mathrm{R}}_{\mu}| \leq |[\alpha\beta]^{\mathrm{R}}_{\mu}|$.

Another bit of information useful in the identification of class types is revealed in the following trivial claim.

**Claim 4.7 (Right End Equivalence Class Types of $[\lambda]^{\mathrm{R}}_{c\mu}$ and $[c\mu]^{\mathrm{R}}_{c\mu}$)**
Let $\mu \in \Sigma^*$, $c \in \Sigma$, then

(i) $[\lambda]^{\mathrm{R}}_{c\mu}$ is of *Type 5* if and only if $c\mu = c^k$, and it is of *Type 7* otherwise.

(ii) $[c\mu]^{\mathrm{R}}_{c\mu}$ is of *Type 1* if there exists another unique suffix of $c\mu$, and it is of *Type 2* otherwise.

Finally, the following claim describes the detection of class type for vertices in both the DAWG and the CDAWG. It is a straightforward consequence of DAWG and CDAWG definition.

**Claim 4.8 (Right End Equivalence Class Types Detection)**
Let $\mu \in \Sigma^*, c \in \Sigma, \alpha \in \text{Factor}(c\mu), \alpha = (\alpha)^{\text{R}}_{c\mu}$ and $[\alpha]^{\text{R}}_{c\mu}$ is a vertex on the backbone of graph $G$ which is a DAWG or a CDAWG for string $c\mu$, then

(i) $|\text{Occur}^{\text{R}}_{c\mu}(\alpha)| > 1$ if and only if there exists $a \in \Sigma$ such that $[a\alpha]^{\text{R}}_{c\mu}$ is a vertex of graph $G$.

(ii) $|\text{Context}^{\text{L}}_{c\mu}(\alpha)| = 1$ if and only if there exists exactly one $a \in \Sigma$ such that vertex $[a\alpha]^{\text{R}}_{c\mu}$ exists in graph $G$.

(iii) $|\text{Context}^{\text{L}}_{c\mu}(\alpha)| > 1$ if and only if there exist $a, o \in \Sigma$ such that $a \neq o$ and vertices $[a\alpha]^{\text{R}}_{c\mu}$ and $[o\alpha]^{\text{R}}_{c\mu}$ exist in graph $G$.

(iv) $\alpha \in \text{Prefix}(\mu)$ if and only if $\alpha = c^k$ and $c$ is the next symbol on the backbone.

(v) $|[\alpha]^{\text{R}}_{c\mu}| = 1$ if and only if no vertices above $[\alpha]^{\text{R}}_{c\mu}$ on the backbone, including $[\alpha]^{\text{R}}_{c\mu}$ itself, have more than one in-edge.

The question of existence of vertices like $[a\alpha]^{\text{R}}_{c\mu}$ can be answered using reversed suffix links in the DAWG [8]. However, this will not work in the CDAWG [40]. This is one of the side effects of the path compression and the resulting change of many vertices and suffix links from explicit to implicit. For instance, string *aa* is both left and right branching in string $\mu$ = *aaaoaacaa*, but vertex $[aa]^{\text{R}}_{\mu}$ has no incoming suffix link. That means that to monitor their existence, we have to resort to something more complicate like tracing these strings in parallel with the strings on the backbone.

## 4.2.2 Algorithms

Now that we have enough information about changes made by a single operation `Delete`, it is time to convert that knowledge into algorithms. The two simple algorithms for the suffix trie and the suffix tree will be reviewed only briefly, while most attention will be paid to the other two suffix graph types.

The changes in the suffix trie and the suffix tree are limited to the end of the backbone. Specifically, they are limited to the part between the node representing the longest non-unique prefix at one end and the leaf representing the longest prefix at the other. In the case of the suffix trie, the longest leaf and vertices above are examined, and if they are neither branching nor holding the active point, they are deleted. The work of `DeleteOldestSymbol` is done as soon as we reach the first vertex that is not deleted. While we may also want to update the active point by moving it sideways, the reset made by ResetActivePoint-Position makes this action unnecessary.

The deletion process in the suffix tree starts in the longest leaf as well. However, due to the path compression, the longest non-unique prefix is right above it either as an explicit node or an implicit node with the active point. If the active point is located on the in-edge terminating in the longest leaf, its current implicit vertex is made explicit and becomes the terminal vertex of the suffix link leading from the last but one leaf created. Moreover, it is attached to the parent of the longest leaf using the edge that formerly terminated in the longest leaf, which is deleted. The last action done by DeleteOldestSymbol in this case is to move the active point sideways to the location of the new longest repeated suffix. Otherwise, when the longest non-unique prefix is branching, then the longest leaf and its in-edge must be deleted. However, the node representing the longest non-unique prefix may now have only a single out-edge. If this is the case, this node is changed from explicit to implicit and its incident edges are concatenated into a single one. No more changes to the suffix tree are necessary.

The description used above neglects to mention anything about the need to find the longest leaf to start from. However, it is quite easy to keep track of its changes during both AppendSymbol and DeleteOldestSymbol. The longest leaf is identical to the root in the empty graph and becomes separate as soon as the first leaf is created. During AppendSymbol, the longest leaf does not change in the suffix tree, but needs to be updated to the new leaf under it in the suffix trie. On the other hand, as DeleteOldestSymbol removes it, its successor must be found by using its suffix link, before it is removed from the graph. Note that all actions needed by DeleteOldestSymbol and the longest leaf tracking are cheap enough not to harm the time complexity and the sliding is asymptotically as fast as the construction for both graph types.

With the simple cases of DeleteOldestSymbol out of the way, we may turn our attention to the complex cases of DAWG and CDAWG. As the changes to these graph types may be quite extensive at times, it might be hard if not impossible to implement the perfect sliding in a time linear in the length of the input string. Indeed, as proved by Blumer [8] for DAWG and Senft and Dvořák [40] for CDAWG, the perfect sliding in linear time is really not possible. The following lemma, adapted from [40], shows one type of string which prevents fast sliding which is illustrated in Figures 4.1 and 4.2. Note that unlike previous versions of this lemma, this version handles both DAWG and CDAWG. Moreover, as it requires only a binary alphabet, it also improves the result obtained by Blumer for DAWG [8] that needed an alphabet with four symbols.

**Lemma 4.9 (Adversary String for DAWG and CDAWG Sliding)**
Let $k \geq 2$, $m \geq 1$, $a, c \in \Sigma$, $a \neq c$, $\varsigma = (ac)^m a$ and $\mu \in \mathrm{Factor}(\varsigma^k)$ such that $\varsigma a \in \mathrm{Factor}(\mu)$. Let $R = \{ [\alpha]^{\mathrm{R}}_{\mu} \mid \alpha \in \mathrm{Branch}^{\mathrm{R}}(\mu) \}$ and $r = |R|$, then

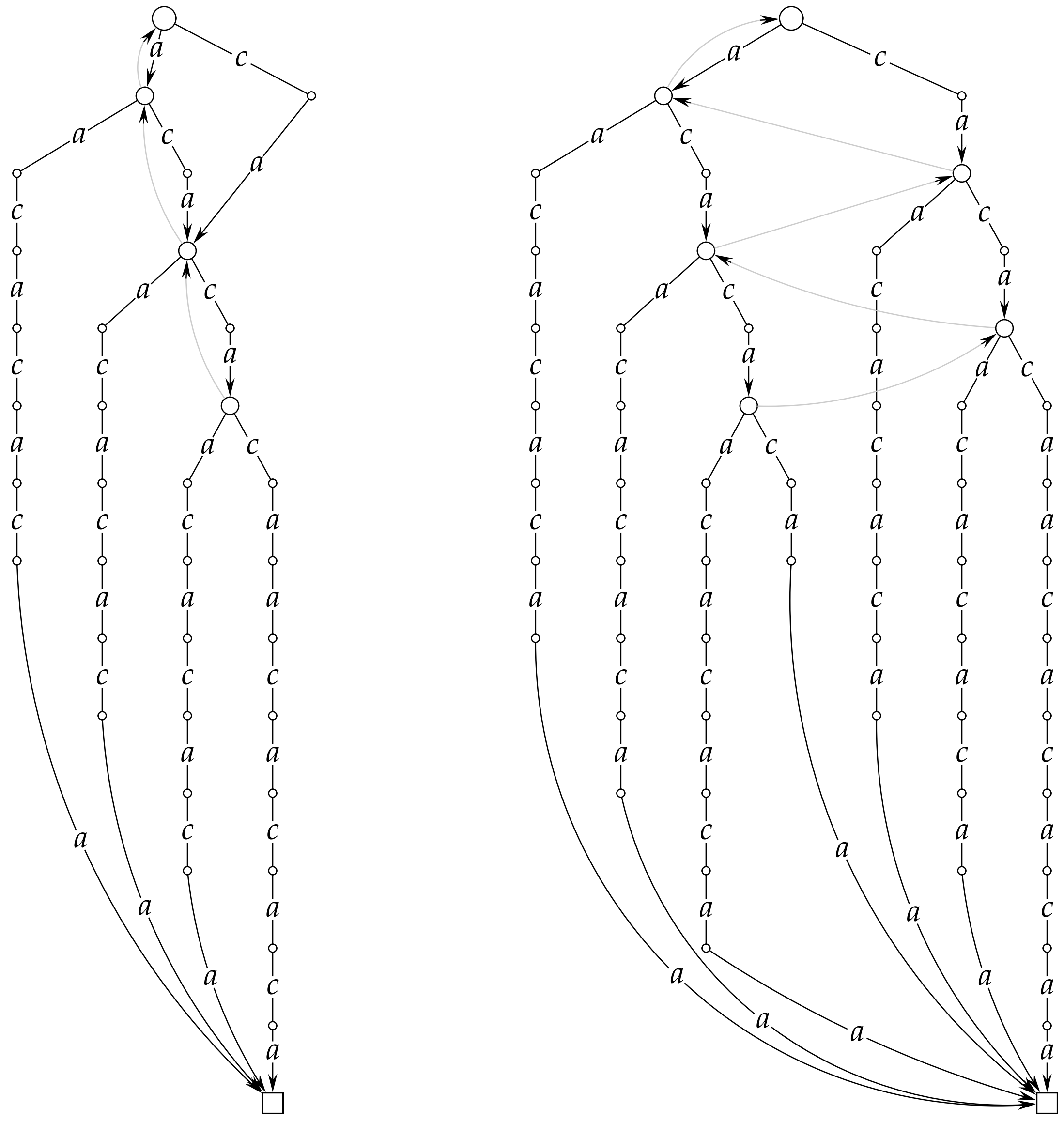


**Figure 4.1.** CDAWG sliding over adversary string *acacacaacacacacaacacaca* in a window of size $w = 14$ (Part I). The first three sliding steps after the build phase are shown. The symbolics used is described in Table 1.1.

$$r \begin{cases} = m+1 & \text{if } \mu[1] = a\,, \\ \geq m+p+1, \text{where } 0 \leq p < m \text{ and } c(ac)^p a\varsigma a \in \text{Prefix}(\mu) & \text{if } \mu[1] = c\,. \end{cases}$$

**Proof:**
Note that $R \setminus \{\lambda\} = \{(ac)^i a \mid 0 \leq i \leq m-1\} \cup \{c(ac)^i a \mid 0 \leq i \leq m-2\}$ in either case. Moreover, $(ac)^i a \equiv^{\mathrm{R}}_{\mu} (ac)^j a$ if and only if $i = j$, and the same holds for $c(ac)^i a \equiv^{\mathrm{R}}_{\mu} c(ac)^j a$.

If $\mu[1] = a$, then we also have $c(ac)^i a \equiv^{\mathrm{R}}_{\mu} (ac)^{i+1} a$ for every $0 < i < m-2$. Subsequently, $R \setminus \{\lambda\} = \{[(ac)^i a]^{\mathrm{R}}_{\mu} \mid 0 \leq i \leq m-1\}$ and $r = m+1$, as stated.

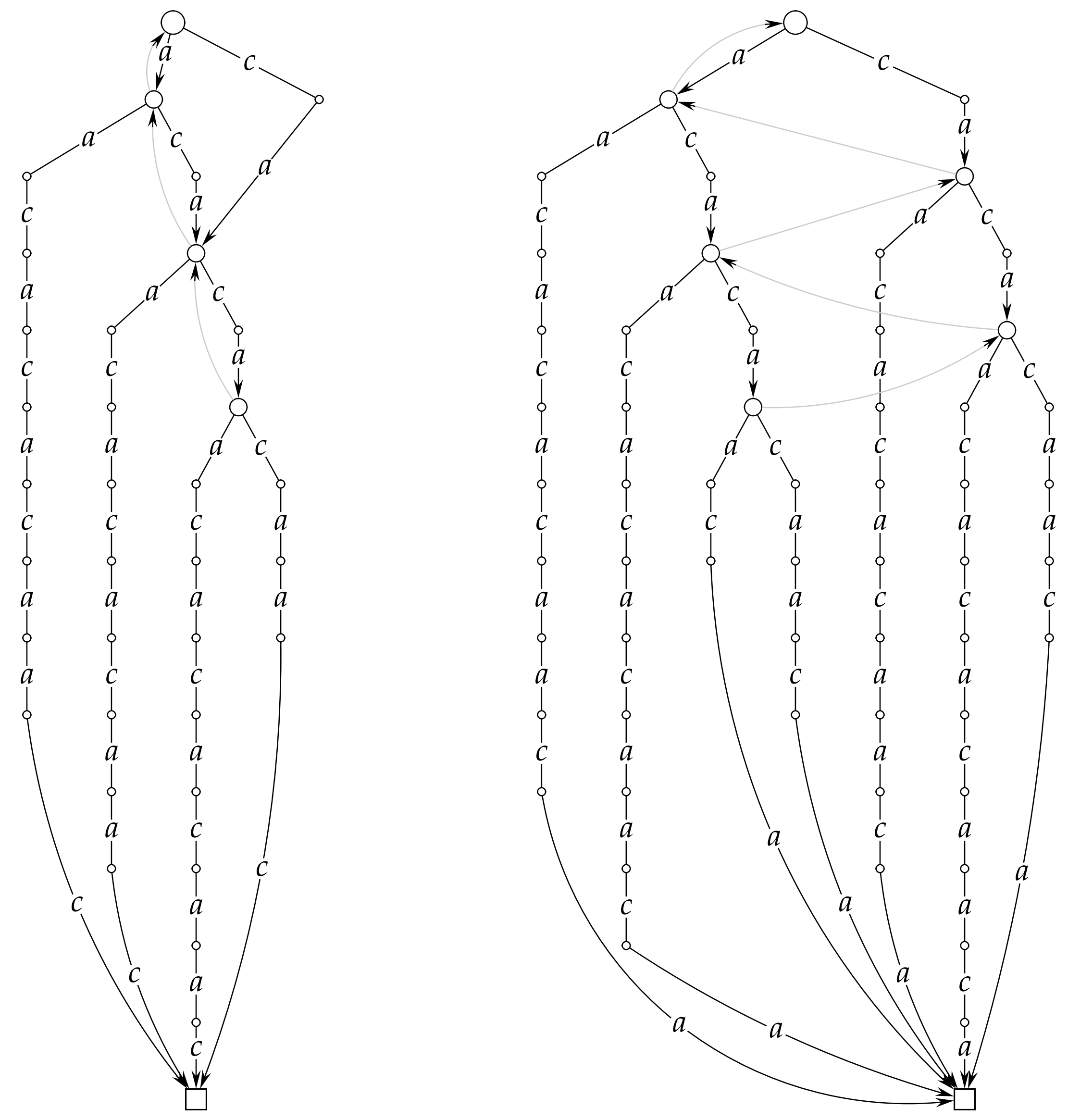


**Figure 4.2.** CDAWG sliding over adversary string *acacacaacacacacaacacaca* in a window of size $w = 14$ (Part II). The first three sliding steps after the build phase are shown. The symbolics used is described in Table 1.1.

When $\mu[1] = c$, there exists and integer $p$ such that $0 \le p < m$ and $c(ac)^p a\text{ç}a \in \text{Prefix}(\mu)$. Thus $c(ac)^i a \in \text{Prefix}(\mu)$ for every $i$ such that $0 \le i \le p$. Consequently, $c(ac)^i a \not\equiv^{\mathrm{R}}_{\mu} (ac)^j a$ for any $j > i$. Moreover, as $a(ac)^j a \in \text{Factor}(\mu)$ for every $j$ satisfying $0 \le j < m$, string $(ac)^j a$ is left branching in these cases. Therefore, $c(ac)^i a \not\equiv^{\mathrm{R}}_{\mu} (ac)^j a$ for every $i$ satisfying $j \le i \le p$. On the other hand, equivalence $c(ac)^i a \equiv^{\mathrm{R}}_{\mu} (ac)^{i+1} a$ still holds for every $i$ such that $p < i < m - 2$, as in the previous case. It follows that

$$R \setminus \{\lambda\} = \left\{ [(ac)^i a]^{\mathrm{R}}_{\mu} \mid 0 \le i \le m-1 \right\} \dot\cup \left\{ [c(ac)^i a]^{\mathrm{R}}_{\mu} \mid 0 \le i \le \min(p, m-2) \right\}$$

is *a* set of size $m + \min(p, m-2) + 1 \ge m + p$. Thus, $r \ge m + p + 1$, as expected. ∎

The following theorem uses the adversary string from the lemma above to show the impossibility of perfect sliding of DAWG or CDAWG in linear time.

**Theorem 4.10 (Perfect Sliding Lower Bound for DAWG and CDAWG)**
Let $|\Sigma| \geq 2$, $\mu \in \Sigma^*$ and $0 < w < |\mu|$. Then any algorithm maintaining DAWG or CDAWG for a perfect sliding window of size $w$ over string $\mu$ requires *AsymptoticLowerBoundIsw*$(|\mu| - w)$ time in the worst case.

**Proof:**
Since at least $|\mu| - w$ steps of the algorithm are needed to move the sliding window from the initial to the final position, the lower bound holds for all $w \leq 5$.

For $w \geq 6$, put $m = \left\lfloor \frac{w-2}{4} \right\rfloor$ and $\varsigma = (ac)^m a$ for distinct $a, c \in \Sigma$. Now, consider the input string $\mu = \varsigma^{|\mu|}[1 .. |\mu|]$ where $|\mu| > w$. As $w \geq 4m + 2$, each string $\mu[i .. i + w - 1]$ contains $\varsigma a$ as a factor, for every $i$ satisfying $1 < i < |\mu| - w + 1$. Since $\mu$ is a power of a string of length $2m + 1$, it suffices to consider only $i < 2m + 1$. Let $R_i$ denote $\{ [\alpha]^{\mathrm{R}}_{\mu} \mid \alpha \in \mathrm{Branch}^{\mathrm{R}}(\mu[i .. i + w - 1]) \}$. Then, for $i = 1, 2, \ldots, 2m + 1$, we obtain the following from Lemma 4.9:

$$|R_i| \begin{cases} = m + 1 & \text{if } i \text{ is odd ,} \\ \geq 2m - \dfrac{i}{2} + 1 & \text{if } i \text{ is even .} \end{cases}$$

Subsequently, for every even $i$, at least $m - \frac{i}{2}$ branching vertices must be altered to create $R_i$ from $R_{i-1}$. Assuming that this modification takes at least one step of the algorithm, the total number of steps required to create $R_i$ from $R_{i-1}$ for every $i = 2, 3, \ldots, 2m + 1$ is at least

$$\sum_{i=2, i:\text{even}}^{2m} \left(m - \frac{i}{2}\right) = \sum_{i=1}^{m} (m - i) = \frac{(m^2 - m)}{2} .$$

Thanks to the periodic nature of $\mu$, this is also a lower bound on the number of steps to construct $R_i$ for every $i = t + 2, t + 3, \ldots, t + 2m + 1$, where

$$t = q(2m + 1) \text{ and } 0 \leq q < \left\lfloor \frac{|\mu| - w + 1}{2m + 1} \right\rfloor .$$

Therefore, the total number of steps of the perfect sliding algorithm for DAWG or CDAWG can be bounded from below by

$$\left\lfloor \frac{|\mu| - w + 1}{2m + 1} \right\rfloor \frac{m^2 - m}{2} = \Omega\Big(\big(|\mu| - w\big)\, m\Big) = \Omega\left(\big(|\mu| - w\big) \left\lfloor \frac{w - 2}{4} \right\rfloor \right) = \Omega\Big(\big(|\mu| - w\big) w\Big) .$$

Consequently, the total time complexity is bounded by $\Omega(w(|\mu| - w))$. ■

As the trivial scrap and build sliding algorithm reaches the above lower bound, the exact bound on the complexity of perfect sliding of DAWG or CDAWG is as follows.

**Theorem 4.11 (Perfect Sliding Complexity of DAWG and CDAWG)**
The problem to maintain a DAWG or CDAWG for a perfect sliding window of size $w$ over a string on an alphabet $\Sigma$ has the following amortised time complexity per one sliding window move:

$$\begin{aligned} &\Theta(w) && \text{if } |\Sigma| \geq 2\,, \\ &\Theta(1) && \text{if } |\Sigma| = 1\,. \end{aligned}$$

This settles the question of the complexity, but the motivation to create an incremental algorithm that does only the minimal changes necessary remains. We still want to preserve any information collected in the graph during the sliding process. One possible solution is shown in Algorithm 4.2. Similar to what we do with the active point during `AppendSymbol`, we follow the so called *backbone point*, this time down the backbone. All necessary changes are made in its vicinity and are finished by eventual cleanup of unreachable part of the graph. The details of every function used in this implementation of `DeleteOldestSymbol` are presented below.

`SimpleDeleteOldestSymbol`
: This function detects the case when the delete operation is simple, and makes the necessary changes. The simple delete takes place only if there is exactly one edge leading from the source. This in turn happens only if $c\mu = c\alpha^k$, which means that all vertices on the backbone below the source are of *Type 2*. In that case, this function modifies the graph and returns true, while it leaves the graph as is and returns false otherwise. The modifications needed in the simple case are minor. The sink and its incoming edge are deleted in DAWG and nothing is done in CDAWG, as the open edge leading to the sink will shorten automatically. Note that only the source is left in the graph if $c\mu = c$. Moreover, the active point must be moved one symbol up from its current position in the new sink, to the vertex representing the new longest non-unique suffix.

`ComplexDeleteOldestSymbol`
: This function performs the deletion of the oldest symbol in complex cases, which the `SimpleDeleteOldestSymbol` function cannot handle. The delete is performed in four steps, where the first three only do initialisation. Functions `InitBackbonePoint`, `InitLeftContexts` and `InitClassTypeFlags` prepare the backbone point, the left context of the backbone point monitoring and the class type flags, respectively. The delete itself is then done using function `AdjustBackbonePointVertexLoop`.

```
void DeleteOldestSymbol()
{
    if (! SimpleDeleteOldestSymbol())
        ComplexDeleteOldestSymbol();
}
void ComplexDeleteOldestSymbol()
{
    InitBackbonePoint();
    InitLeftContexts();
    InitClassTypeFlags();
    AdjustBackbonePointVertexLoop();
}
void AdjustBackbonePointVertexLoop()
{
    while (   MoveBackbonePointDown()
           && AdjustBackbonePointVertex())
        ;
}
Bool AdjustBackbonePointVertex()
{
    UpdateLeftContexts();
    UpdateClassTypeFlags();

    return AdjustBackbonePointVertexByType();
}
Bool AdjustBackbonePointVertexByType()
{
    switch (GetBackbonePointClassType()) {
    case 1: return AdjustBackbonePointVertexOfTypeOne();
    case 2: return AdjustBackbonePointVertexOfTypeTwo();
    case 3: return AdjustBackbonePointVertexOfTypeThree();
    case 4: return AdjustBackbonePointVertexOfTypeFour();
    case 5: return AdjustBackbonePointVertexOfTypeFive();
    case 6: return AdjustBackbonePointVertexOfTypeSix();
    case 7: return AdjustBackbonePointVertexOfTypeSeven();
    default: return true;
    }
}
```

**Algorithm 4.2.** Outline of `DeleteOldestSymbol` that works on both the DAWG and the CDAWG.

`InitBackbonePoint`
: This function initialises the position of the backbone point, which starts in the source.

`MoveBackbonePointDown`
: This function moves the backbone point one symbol down the backbone. It returns false when the backbone point is in the sink before the move is attempted, and returns true otherwise.

`InitLeftContexts`
: This function initialises the tracing of strings confirming the existence of left contexts of the backbone point vertex. It also remembers the last non-unique vertex, i.e. when the backbone point had at least one left context left for the last time. This is initialised to the source. As the backbone point moves to the empty string at first, all one symbol long strings represented in the graph are entered into the evidence.

`UpdateLeftContexts`
: This functions updates the information about the existence of left contexts of the backbone point vertex after it moved down the backbone. It moves down from all vertices representing strings in its evidence, using the last symbol used to move the backbone point down on the backbone. If such move is not possible for some string, it is removed from the evidence. Moreover, if there is still at least one string in the evidence, the last non-unique vertex is set to the backbone point vertex.

`InitClassTypeFlags`
: This functions initialises the flags used to detect the type of the right end equivalence class of the backbone point vertex. There are only two flags needed and both are initially set to true. One flag is for monitoring if the class is trivial, i.e. has only a single member. The other flag is for monitoring if the longest member of the backbone point class is a simple string, i.e. $c^k$.

`UpdateClassTypeFlags`
: This function updates the flags used to detect the type of the right end equivalence class of the backbone point vertex. The flags are left unchanged with the exception of the following two cases. If the trivial class flag is true and the current backbone point vertex has two or more in-edges, then the trivial class flag is set to false. If the simple string flag is true and the last symbol used to move the backbone point down differs from the first symbol of the underlying string, then the simple string flag is changed to false.

`AdjustBackbonePointVertexLoop`
: This function moves the backbone point down the backbone in a loop and adjusts backbone vertices as required. It moves the backbone point down using function `MoveBackbonePointDown` and then makes the necessary changes using function `AdjustBackbonePointVertex`. It stops when

either of the two function calls returns false, which is when the last vertex that needed changes was adjusted.

`AdjustBackbonePointVertex`
This function is responsible for adjustments of the backbone around the backbone point. It starts by updating the information needed for the detection of the right end equivalence class type of the backbone point vertex. Functions `UpdateLeftContexts` and `UpdateClassTypeFlags` are used for this. After that, the call to function `AdjustBackbonePointVertexByType` performs the changes required by the detected class type. This function returns the value returned by `AdjustBackbonePointVertexByType`.

`AdjustBackbonePointVertexByType`
This function detects the appropriate right end equivalence class type for the backbone point using `GetBackbonePointClassType` and calls the appropriate function to handle that type. These specialised functions are: `AdjustBackbonePointVertexOfTypeOne`, `AdjustBackbonePointVertexOfTypeTwo`, `AdjustBackbonePointVertexOfTypeThree`, `AdjustBackbonePointVertexOfTypeFour`, `AdjustBackbonePointVertexOfTypeFive`, `AdjustBackbonePointVertexOfTypeSix` and `AdjustBackbonePointVertexOfTypeSeven`.

`GetBackbonePointClassType`
This function uses the information from flags, left context monitoring and graph around the backbone point to detect the right end equivalence class type. It returns 0, if the backbone point is at an implicit node. Otherwise, it finds final values of properties deciding the class type and uses Table 4.1 to find and return the detected type. While the trivial class flag and the context count do not need adjustment, the simple string flag does. If the symbol under the backbone point on the backbone is the same as the first symbol of the underlying string, the final value of the simple string flag is true, and it is false otherwise.

`AdjustBackbonePointVertexOfTypeOne`
This function handles the case when the right end equivalence class at the backbone point is of type *Type 1*. This is easy in DAWG, where the last explicit edge used to move the backbone point is removed from the graph. The changes in CDAWG are complicated by implicit vertices. Like in the case of the suffix tree, the deciding factor is the last non-unique vertex, which may be either explicit or implicit. If it is implicit, it is located on the last explicit edge used to move the backbone point down. In that case, this explicit edge has its label shortened to reach only the last non-unique vertex position. Moreover, the active point, which was in the last non-unique vertex, must be moved sideways, to the vertex representing the new longest non-unique suffix. Otherwise, the last explicit edge used to move the backbone point down is removed from the graph. After that, the last non-unique vertex is

verified to be either the source or a branching node, and is made implicit otherwise. This function always returns true.

`AdjustBackbonePointVertexOfTypeTwo`
: This function handles the case when the right end equivalence class at the backbone point is of type *Type 2*. Any calls to this function signal a bug, as all vertices of this type were already solved in function `SimpleDeleteOldestSymbol`. This function always returns false.

`AdjustBackbonePointVertexOfTypeThree`
: This function handles the case when the right end equivalence class at the backbone point is of type *Type 3*. The last explicit edge used to move the backbone point down is redirected to the location of the vertex confirming the last existing left context. If this vertex is implicit, the edge being redirected and its label are adjusted accordingly. This function always returns false.

`AdjustBackbonePointVertexOfTypeFour`
: This function handles the case when the right end equivalence class at the backbone point is of type *Type 4*. It performs the same steps as function `AdjustBackbonePointVertexOfTypeThree` does. However, it then continues by removing all vertices and edges that are no longer accessible from the source, starting with the backbone point vertex. Note that suffix links incident with vertices being deleted must be replaced to keep the suffix graph valid after cleanup. Likely the easiest way to do so is to use reverse suffix links and change every suffix link terminating in a node before it is deleted. The terminal node of these suffix links is set to the terminal node of the suffix link leading from the node that is currently being removed. Finally, if the active point is encountered during cleanup, it must be moved sideways to its new location. This function always returns false.

`AdjustBackbonePointVertexOfTypeFive`
: This function handles the case when the right end equivalence class at the backbone point is of type *Type 5*. No changes are necessary for this class type. This function always returns true.

`AdjustBackbonePointVertexOfTypeSix`
: This function handles the case when the right end equivalence class at the backbone point is of type *Type 6*. It creates a new explicit vertex with out-edges that are duplicates of out-edges of the backbone point vertex. After that, the last explicit edge used to move the backbone point down is redirected to this new vertex. Also, reverse suffix links of the old node are used to redirect all suffix links terminating in the old node to the new node and the suffix link of the new node is terminated in the old node. Finally,

the backbone point is relocated to the new vertex. This function always returns true.

`AdjustBackbonePointVertexOfTypeSeven`
: This function handles the case when the right end equivalence class at the backbone point is of type *Type 7*. No changes are necessary for this class type. This function always returns true.

Now that the description is finished, it is time to examine its time complexity. However, the following useful lemma will be proved first.

**Lemma 4.12 (Backbone Left Context Non-Equivalence)**
Let $a, c \in \Sigma$, $\alpha, \beta \in \text{Prefix}(\mu)$ and $a\alpha, c\beta \in \text{Factor}(\mu)$. If $a\alpha \neq c\beta$, then $a\alpha \not\equiv^{\text{R}}_{\mu} c\beta$.

**Proof:** Suppose that this is not the case and $a\alpha \equiv^{\text{R}}_{\mu} c\beta$. Without loss of generality assume that $|a\alpha| \leq |c\beta|$. By Claim 2.12 (ii), $a\alpha$ must be a suffix of $c\beta$ and $|a\alpha| < |c\beta|$. Moreover, since $|a\alpha| \leq |\beta| < |c\beta|$, we get $\beta \equiv^{\text{R}}_{\mu} c\beta$ from Claim 2.13. However, by Claim 2.12 (iv), $\beta$ must be the longest member of $[c\beta]^{\text{R}}_{\mu}$ which is a contradiction, as $c\beta$ is a longer member this class. ■

The following theorem settles the question of the time complexity of Algorithm 4.2.

**Theorem 4.13 (`DeleteOldestSymbol` Complexity)**
The time required by a single call to Algorithm 4.2 is $\text{O}(wB)$ in the worst case. Here $w$ is the length of the sliding window and $B$ is the complexity of branching in the worst case.

**Proof:** Since all functions called outside the loop in `AdjustBackbonePointVertexLoop` are called only once, their combined time complexity is within the $\text{O}(wB)$ bound in the worst case. On the other hand, functions called from `AdjustBackbonePointVertexLoop` might be used up to $w$ times. However, all but `UpdateLeftContexts`, `AdjustBackbonePointVertexOfTypeFour` and `AdjustBackbonePointVertexOfTypeSix` are bounded by $\text{O}(B)$ in the worst case. The total time used by these function calls is then $\text{O}(wB)$ in the worst case. Thus the required time complexity in the worst case is met so far and the total cost of all calls to `UpdateLeftContexts`, `AdjustBackbonePointVertexOfTypeFour` and `AdjustBackbonePointVertexOfTypeSix` remains to be analysed.

`UpdateLeftContexts`
: This function is tracing vertices confirming the existence of left contexts of prefixes. However, by Lemma 4.12, no vertex can be used to confirm the existence of context twice. Moreover, by Claims 2.17 and 2.21, the number

of vertices is bound by O($w$) for both DAWG and CDAWG. Consequently, the time required by this function is bound by O($wB$) in the worst case.

`AdjustBackbonePointVertexOfTypeFour`
: Apart from the cleanup of the no longer accessible part of the graph, this function is the same as `AdjustBackbonePointVertexOfTypeThree` and bound by Θ($B$) in the worst case. Moreover, it is called at most once as the loop in `AdjustBackbonePointVertexLoop` is terminated after this function returns. Hence, we only need to verify that a single cleanup meets the required bound in the worst case. Clearly, there can be no explicit nodes of *Type 0* to be removed as that would mean that this class changed in a way not allowed by Lemma 4.5. Consequently, as only the explicit vertices of the backbone are deleted, the number of vertices, edges and suffix links changed is bound by O($w$) in the worst case. Note that the numbers of vertices and edges are given by Claims 2.17 and 2.21, while the number of suffix links is limited by the number of explicit vertices, as there is at most one suffix link leading from any vertex.

`AdjustBackbonePointVertexOfTypeSix`
: All components of this function with the exception of edge duplication and redirection easily fit into O($B$) in the worst case. Since every edge is redirected and duplicated at most once, the total time required by these actions is bounded by O($wB$) in the worst case. Note that this is the result of Claims 2.17 and 2.21 which show that the number of edges is bounded by O($w$) in the worst case.

The preceding analysis confirmed that every component of Algorithm 4.2 is bound by O($wB$) time in the worst case. Therefore, as the number of components is constant, the complexity of the entire algorithm is O($wB$) in the worst case, as expected. ■

## 4.3 Edge Labels

While functions `AppendSymbol` and `DeleteOldestSymbol` do all the structural modifications, they would not work if function `UpdateEdgeLabels` did not help them. The reason why `UpdateEdgeLabels` is necessary lies in the path compression which forces both the suffix tree and the CDAWG to have edge labels with no constant length bound.

To achieve the desired linear space complexity, the edge labels are identified by a pair of indexes to the input string. Even though the exact way of storing these indexes may differ in each implementation, there is still at least one index used for every edge label. These indexes might be integrated into vertices and shared on edge paths or on suffix link paths, but there are still going to be many of them.

Now, when sliding comes into play, the string buffer referenced by label indexes needs to be reused to conserve memory. However, as the buffer gets overwritten with new symbols, the label indexes stored in the graph become outdated and edge labels will change unintentionally. To prevent this undesirable effect, a way to keep these indexes up to date is needed.

There are several trivial algorithms that can update these indexes. For example, all label indexes in the graph can be updated during a depth-first traversal after every delete or update. Another approach would update all offsets on the path from the root to the last leaf created following every new leaf creation. However, these algorithms are slow and update indexes more often than necessary. The rest of this section is devoted to the description and analysis of more efficient techniques that solve this issue. Note that this section is based on material presented at WDS'05 [44].

### 4.3.1 Batch Update

Likely the most straightforward way to keep labels valid in amortised constant time per update is a batch update algorithm [42, 44] which works for both the suffix tree and the CDAWG. This algorithm does a simple depth-first full update throughout the graph, propagating the newest indexes found up. To achieve the desired time complexity, it does so only once every $fw$ symbols, where $f > 0$ is some fixed positive fraction. Consequently, the size of the string symbol buffer has to grow to at least $(1 + f)w$ symbols, to prevent overwriting of symbols that are still in use. Moreover, the data types used to store indexes have to be enlarged to compensate for the value interval enlargement. However, this approach does not need reverse edges to operate unlike those below.

### 4.3.2 Fiala's and Greene's Percolating Update

While the batch update works, it is quite crude and more sophisticated update methods working in total linear time in the length of string exist, at least for the suffix tree. The first of them was devised by Fiala and Greene when they made the suffix tree slide for the first time [14].

They observed that, due to the heavy use of suffix links during the suffix tree construction, many nodes are not entered regularly and their indexes and labels on their edges can become outdated. Also, while it would be possible to update all ancestors after every new leaf creation, it would be unnecessarily slow and would update nodes near the root far too often. To prevent this, they devised the so called *percolating update* that employs accounting to prevent overupdating and achieve a total linear number of updates over the whole sliding process.

First, the suffix tree is extended to include accounting information. Every node in the suffix tree gets a credit counter that is initially set to 0 and can only grow to 1. Moreover, function `UpdateEdgeLabels` does nothing and functions

`CreateEdgeFromActivePoint` and `DeleteOldestSymbol` must be adjusted to do the following actions.

`CreateEdgeFromActivePoint`
: When a new leaf is created, two credits are spent for updates. The first credit is paid for the update of the leaf's parent in-edge, while the second credit is sent to this parent to be used later. After that function `UpdateAnd-Propagate` is called on leaf's parent to continue updates above.

`DeleteOldestSymbol`
: When a node is deleted, the node's credit counter is ignored, and two new credits are spent, like on creation. However, this case requires caution, as the child's index may not always be more recent than the parent's index during this update.

`UpdateAndPropagate`
: When a node receives a credit from one of its children, its credit counter is checked, and one of the following two actions is taken. When the credit counter is equal to 0, the counter is just incremented to 1 and update ends here. Otherwise, the credit in the node's account is used to update the index in node's parent and the second credit is sent to this parent. The credit counter in the current node is decremented and the update proceeds to its parent.

Note that an efficient implementation of this technique requires the use of reverse edges.

The accounting method can be used to verify that this algorithm works in a total linear time. The total amount of credits used depends on the number of create-leaf and delete-leaf operations and the credits used for each of them. As there are at most as many new leaves created and deleted as there are symbols in the input string, and two credits are distributed during every create or delete operation, the sum is linear in the length of the input string. Moreover, every update is paid for by one credit and so the number of updates is also linear. Consequently, if the update takes only a constant time to perform, this algorithm works in an total linear time.

### 4.3.3 Larsson's Percolating Update

Building on Fiala's and Greene's work, Larsson developed an algorithm that works differently on vertex deletion. This time the node being deleted only updates its parent if it has a credit in its credit counter. The rest of the algorithm, as well as the time complexity is shared with the original. Note that a version of this algorithm adapted for the CDAWG was outlined by Inenaga et al. [21]. However, it is designed for their approximate sliding algorithm and is not guaranteed to achieve a total linear time in the worst case if used for perfect sliding.

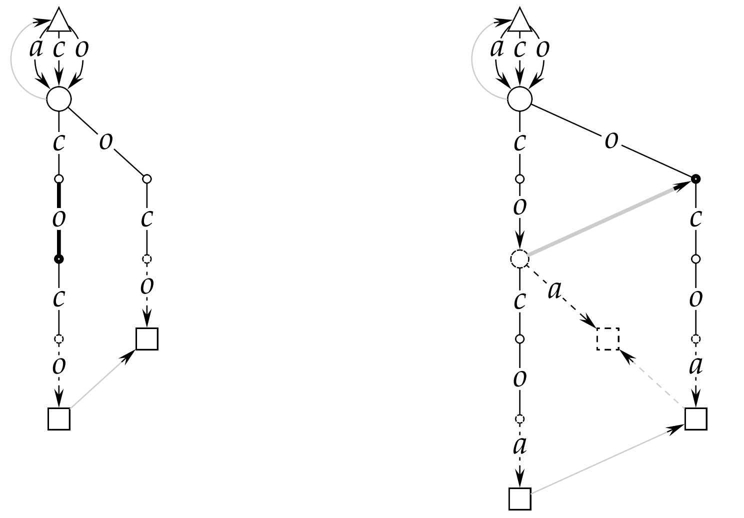


**Figure 4.3.** Example of a situation where percolating update proofs fail. This is the first step in the transition from the suffix tree for string *coco* to the suffix tree for the string *cocoa*. The full construction is in Figure 5.3. The symbolics used is described in Table 1.1.

### 4.3.4 Percolating Update's Correctness

While the proof of correctness of the batch update is trivial, the percolating update calls for a detailed analysis.

So far, there were three attempts to prove the correctness of the percolating update. The first two attempts were made by Fiala and Greene for their original algorithm [14] and Larsson for his version of the algorithm [27]. Unfortunately, as was noted by Senft [42, 44], neither of these attempts succeeded, despite trying two different approaches. The theorem formulated by Fiala and Greene is too weak to prove the label correctness, while the theorem that Larsson tried to prove is invalid. The correctness issue was finally settled by Senft, who proved both versions to be correct [44].

The theorem that Fiala and Greene attempted to prove reads as follows: "Using percolating update, every internal node will be updated at least once every *w* sliding window moves.". However, this index update may update the index value to a newer, but still relatively old value which turns invalid before the next update. Moreover, the proof given is invalid for exactly the same reasons as that of Larsson.

Apparently, Larsson noticed the issues of Fiala's and Greene's algorithm, correctness theorem and its proof. He tried to fix all of these issues by adjusting the algorithm and providing a new theorem and proof. He used the notion of a *fresh credit* to formulate his theorem which reads: "Each node has received a fresh credit from each of its children.". The fresh credit is a credit that reached the current node from a leaf that is still present in the tree. Unfortunately, this theorem is invalid, as any new node created in the tree does not update its parent during its creation. Consequently this parent does not receive a fresh credit from this child, which is a contradiction. Moreover, the new node breaks the condition as well, as it does not receive a fresh credit from its older child. Both issues are illustrated in Figure 4.3. Note that the same situation also breaks Fiala's and Greene's proof of their weak theorem.

Now that the issues of the first two attempts to prove correctness of percolating update are known, it is time to show a solution to the question of its correctness. To ease the discussion, the concept of *ancient node* is introduced. A node is *ancient* if the leaf it was created with was already deleted from the tree. Clearly, every ancient node must be older than any non-ancient node or any leaf. Moreover, all non-ancient nodes and leaves have their indexes up-to-date and only ancient nodes need to be verified. Also, root node has no in-edge that needs updating as edges from the bot are handled differently.

**Theorem 4.14**
Every ancient node with the exception of the root received at least one fresh credit from every subtree rooted in one of its children.
**Proof:** To prove the theorem by the way of contradiction, let there be a suffix tree with ancient node $\alpha$ that did not receive a fresh credit from subtree $S$ rooted at its child $\alpha\beta$. Moreover, let $\alpha$ be such that all other ancient nodes in its subtree received all required fresh credits. What happened to fresh credits from each subtree is analysed below.

1. If the root of the subtree $S$ is a leaf $\alpha\beta$, then the ancient node $\alpha$ received a fresh credit from it for the following reasons. As the ancient node $\alpha$ is older than any fresh credit, no fresh credit could skip it. Moreover, credits are never lost during node deletion and if they are used for update, they are used in pairs, and then the one that is more fresh is sent up. Thus a fresh credit had to reach the ancient node.

2. If the root of the subtree $S$ is another ancient node $\alpha\beta$, then the situation is similar to case 1. All ancient nodes were present in the tree before the fresh credit. Moreover, $\alpha\beta$ received at least two fresh credits and subsequently sent one up, and credits never get lost.

3. If the root of the subtree $S$ is a non-ancient node $\alpha\beta$, then there are two possibilities:

    a. The non-ancient node $\alpha\beta$ received a fresh credit from at least one child different from the leaf it was created with. Thus it must have sent a fresh credit to its parent. Again, the fresh credit from this subtree reaches $\alpha$ for reasons given in cases 1 and 2.

    b. Otherwise, the non-ancient node $\alpha\beta$ received a fresh credit only from the leaf it was created with. Due to this, it must have only two children as any additional out-edge would have to be created with a new leaf, which would update $\alpha\beta$ and give it a second fresh credit. One subtree contains the leaf that came with $\alpha\beta$, along with one fresh credit. The other subtree did not send a fresh credit to $\alpha\beta$ yet. Thus the root of this subtree $\alpha\beta\gamma$ can not be a leaf newer than $\alpha\beta$, but it can be of any of the following three types.

i. The root $\alpha\beta\gamma$ is an ancient node or a leaf older than its parent. Such vertex sent a fresh credit up before the $\alpha\beta\gamma$ was created. This fresh credit was sent from the subtree $S$ up and received by $\alpha$.

ii. The root $\alpha\beta\gamma$ is a non-ancient node that sent a fresh credit up before the current parent was created. Again, a fresh credit was sent from the subtree $S$ and received by $\alpha$.

iii. The root is a non-ancient node that has not yet sent a fresh credit up. There can be an arbitrarily long, but finite, sequence of such non-ancient nodes in a parent-child relationship. The last node in this sequence must have a child that is covered by case i or case ii. Thus even in this case the subtree $S$ sent a fresh credit that was received by $\alpha$.

Hence, by every accounting, the ancient node $\alpha$ must have received a fresh credit from subtree $S$ rooted at $\alpha\beta$, which is a contradiction. ■

# 5. Compression

The last step on the path to the suffix graph based data compression is made here. This chapter builds on all previous chapters to derive, describe and analyse this concept. All material used in this chapter is based on our original research, but parts of it were already published elsewhere [42, 43, 41].

While both the suffix tree based data compression and the suffix graph based data compression ideas are novel, suffix graphs were used in data compression long before they were formulated. Specifically, it was the suffix tree that was used to implement searching and sorting in various compression methods. These methods include the Ziv-Lempel'77 (ZL77) dictionary compression type [14, 38], the Prediction by Partial Matching (PPM) type [26, 38] and the Burrows-Wheeler Compression (BWC) type [3, 38]. Luckily, an apparent inefficiency that was spotted in the work of Fiala and Greene on ZL77 [14] led to the idea of the suffix tree based data compression [42, 43, 41]. This idea is extended here to the idea of suffix graph based data compression.

The first section of this chapter describes the origins of the suffix tree based data compression. The later sections then elaborate on the idea, add details and provide both theoretical and practical analysis.

## 5.1 Suffix Tree and Data Compression

As already noted above, the suffix tree has a long history in data compression applications [14, 26, 3]. At least three major lossless data compression methods were implemented using the suffix tree. For example, in 1989 Fiala and Greene used a depth limited sliding suffix tree to implement the match searching in a ZL77-type dictionary compression method [14, 38]. Seven years later, Larsson described a way to use the suffix tree for context searches in a PPM-style compression method with unbounded context length [26, 38]. Three years after that, Balkenhol et al. replaced the string rotation sorting Burrows-Wheeler Transform (BWT) used in BWC with a similar suffix tree based suffix sorting transform [3, 38]. While these applications certainly proved the versatility of the suffix tree, they did not take a full advantage of this structure and its properties. It was mainly because the suffix tree was perceived only as a tool for an efficient string searching and sorting.

The use in the BWC alternative is the best example of this approach among the three application mentioned above. In this case, the suffix tree is used solely for suffix sorting and nothing else. On the other hand, the application to the PPM-style compression is the one closest to realising the full potential of the suffix tree. Here Larsson takes advantage of the natural ability of the active point

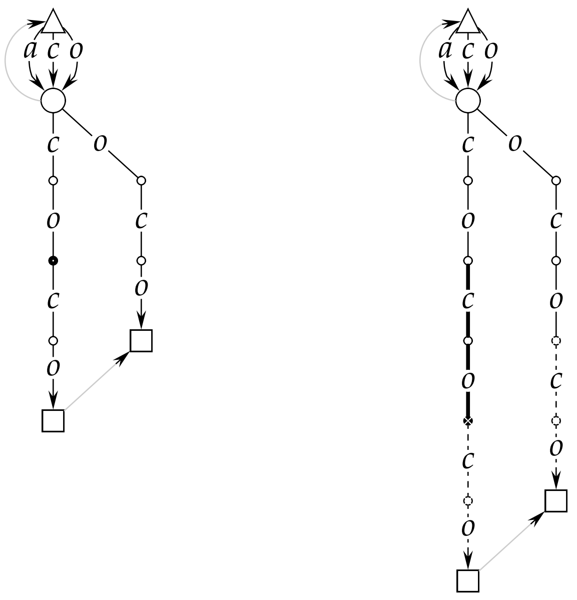


**Figure 5.1.** ZL77-type match search as a byproduct of the suffix tree construction. The input string is *cococo* and a suffix tree was already constructed for prefix *coco*. As the algorithm proceeds to append the next two symbols, a two symbol long match *co* is found. The symbolics used is described in Table 1.1.

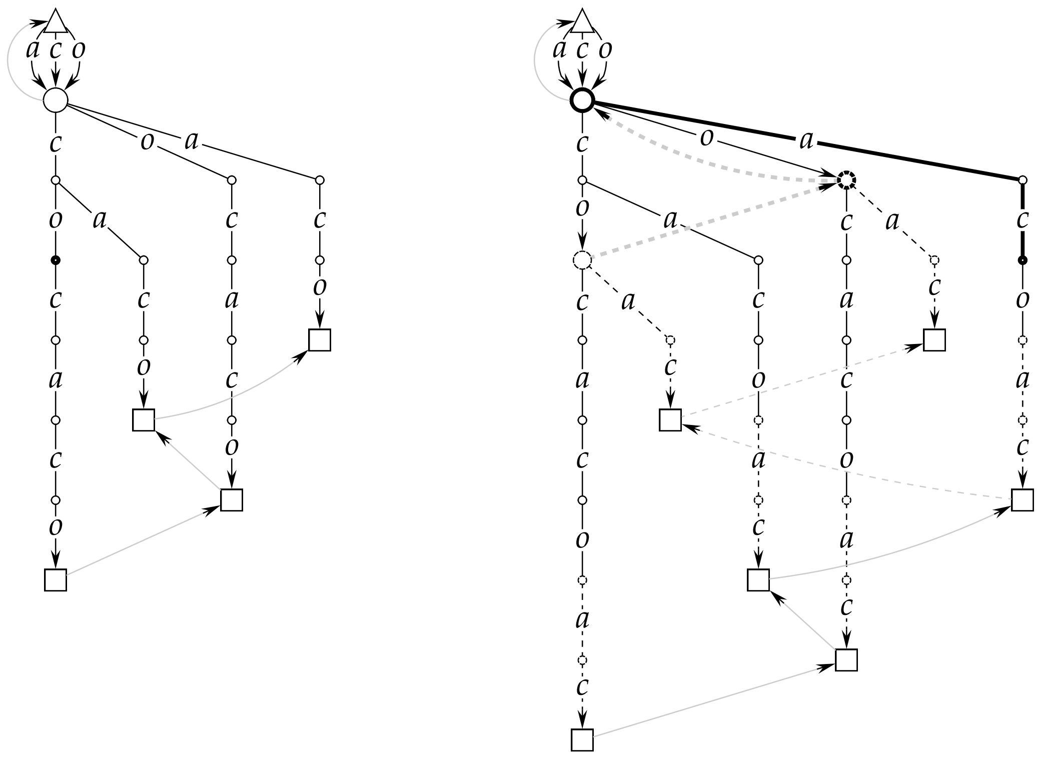


**Figure 5.2.** ZL77-type match search as a byproduct of the suffix tree construction. The input string is *cocacoac* and a suffix tree was already constructed for prefix *cocaco*. As the algorithm proceeds to append the next two symbols, a two symbol long match *ac* is found. The symbolics used is described in Table 1.1.

to follow PPM contexts of the current symbol during construction. While the properties of the suffix tree are not used as much in the third application, the fact that they are not used helped us to devise the suffix tree based data compression. Fiala and Greene implemented a ZL77-type dictionary method with a sliding window to save space and a depth limited suffix tree to speed up searches for string matches. Even though they were the first to find a way to make the suffix tree slide, they did not use its full potential. Whether the depth limit on the tree was the cause or the effect is not clear. However, they certainly look for matches using searches starting at the root, while they could have obtained them as a byproduct of the construction algorithm of the normal suffix tree. The details of this match searching approach are revealed in the next section.

## 5.2 Suffix Tree and Match Search

When a ZL77-type dictionary compression method looks for a match, it searches for a prefix of the as yet unprocessed part of the input string in the already processed part [38]. The match may extend into the unprocessed part of the string, provided that it starts in the already processed and known part. If we have a suffix tree built for the already processed part of the input string, we may search for a match by starting at the root and following the unprocessed part of the string as far down as we can. While this is straightforward and finds all matches in a guaranteed linear time, it is not the most efficient method available. A more sophisticated method would employ the properties of the construction algorithm to find matches directly during construction, without separate searches. Obviously, the construction algorithm has to be selected with this application in mind. Fiala and Greene used McCreight's algorithm, but Ukkonen's algorithm can be used as well. As these construction algorithms move the active point, they search for a prefix of the unprocessed part of the input string in the suffix tree. The movement of the active point can then be translated into a match as follows.

As the search for the next match begins, the active point is located somewhere in the tree. Note that this search cannot start while the active point is in the bot as this signals that the algorithm is dealing with a symbol which is not in the suffix tree. From the match searching point of view, the only important property of the active point is its depth in the tree. As soon as the search begins, any downward movement of the active point extends the match length by one symbol. Note that this works only as long as the number of sideways moves is lower than or equal to the original depth of the active point. The reason is that every move sideways brings the vertex where the match starts closer to the root. Moreover, the root is reached exactly when then number of sideways moves equals the original depths of the active point. The search for a match can be stopped at any time, for instance, when a match length limit is reached or the match is deemed to be good enough. The offset needed to complement the match length can then be obtained from a nearby edge label easily.

Two examples of the conversion of the active point movement to a match are shown in Figures 5.1 and 5.2. The first one shows a straightforward case, where the match is found immediately. The second case is more complicated and requires the active point to move all the way to the root before any match is found at all. Note that even more complex cases with downward and sideways movement interleaving are possible.

Apparently, this approach to match search was never published before it appeared in my Master's Thesis [42]. However, a similar suffix tree based search algorithm was already used in an early versions of the Info-ZIP, before it was replaced by a hash based search algorithm [1]. Nevertheless, this discovery for ZL77 — in combination with discoveries made by Larsson for PPM — led to the idea of the suffix tree based data compression, which is described in the next section.

## 5.3 Suffix Tree Based Data Compression

The two preceding sections demonstrated that, in a way, both PPM and ZL77 compression methods just replace the input with a description of the active point movement throughout the suffix tree. While they share this quality, they represent two very different lossless data compression groups, which brings an interesting question. If the suffix tree is so versatile that it can do this, what other compression methods can be implemented this way? Or better yet, what new compression methods can be devised?

The input string determines the behaviour of the suffix tree construction by influencing the movement of the active point. Consequently, any description of the input string, or active point movement in general, can be used to drive the construction algorithm. So, for example, as the output of every lossless data compression method fully describes the input string, it can drive the suffix tree construction. Thus, one extreme view says that every lossless data compression method falls into the group of suffix tree based data compression methods. Nevertheless, using the suffix tree construction algorithm to reproduce the same data might be really hard if not entirely impossible. Hence, only methods where compression can be implemented easily using the suffix tree construction will be included in the new lossless data compression method (STC) family.

The general idea is to use one of the left-to-right construction algorithms, and, while doing so, somehow save the directions needed to mirror the active point movement on decompression. However, it may not be clear whether this description contains enough information for the decompression of the input string. The reason why it does is that every time the active point moves down, a symbol must be selected through the edge choice. Since these symbols are the same symbols that made the active point move down during the compression, they can be used to recover the original string.

While a suitable left-to-right construction algorithm was already described and analysed, the description of the active point movement needs more attention. We have already dealt with two approaches. One uses the conversion to matches, described in Section 5.2, to create a ZL77-type compression method, which does not need the suffix tree for the decompression. The other one describes every single *a*-edge and suffix link choice separately, to create a PPM-style compression method as suggested by Larsson [26].

One important difference between the two methods is that one describes the active point movement in chunks, while the other handles each step separately. These method types will be denoted by `stc_m` and `stc_u` from now on. Here the last letters are chosen to remind us of the similarity these approaches have to those used in construction algorithms by McCreight and Ukkonen, respectively. Like McCreight's algorithm, `stc_m` methods appear to be optimistic and believe that they will be able to move down a long way before a move sideways is needed. On the other hand, `stc_u` methods are similar to Ukkonen's algorithm in that they appear to be more pessimistic and try to move only one symbol at a time.

Note that `stc_m`-type methods behave like dictionary compression methods enhanced with a PPM-style dictionary selection. When compared to the related LZFG-PM method by Hoang et al. [19], the methods introduced here maintain more compact dictionaries with higher level contexts. On the other hand, when compared to the LZP4 introduced by Bloom [5], the main difference is that our methods can have more than one phrase in each context.

The following lists several methods that can be created using different approaches to the active point movement description. To better illustrate how these methods work, string *cocoao* is converted to instructions that would be saved by each respective method. The step-by-step construction of the suffix tree for string *cocoao* is displayed in Figure 5.3 to aid in the process.

`stc_u`

As already noted above, this method saves each edge or suffix link choice separately. Its instructions are of two types. One type is ($a$), which signals that the *a*-edge was used to move down from the current active point location. Note that this is used even in implicit nodes, where there is no choice. The other type then signals the use of the explicit or implicit suffix link by storing token (esc). The instructions saved by this method for our example string *cocoao* read as follows:

$$(\text{esc})(c)(\text{esc})(o)(c)(o)(\text{esc})(\text{esc})(\text{esc})(a)(o)$$

`stc_m0`

This method is a variant of the ZL77 compression method, which uses the conversion of the active point movement to matches and literals. Literals are denoted by ($a$), while matches are described using a pair of integers $(1, 7)$. Here the first component is the match length and the second is an index of one of its left occurrences starting in the already processed part

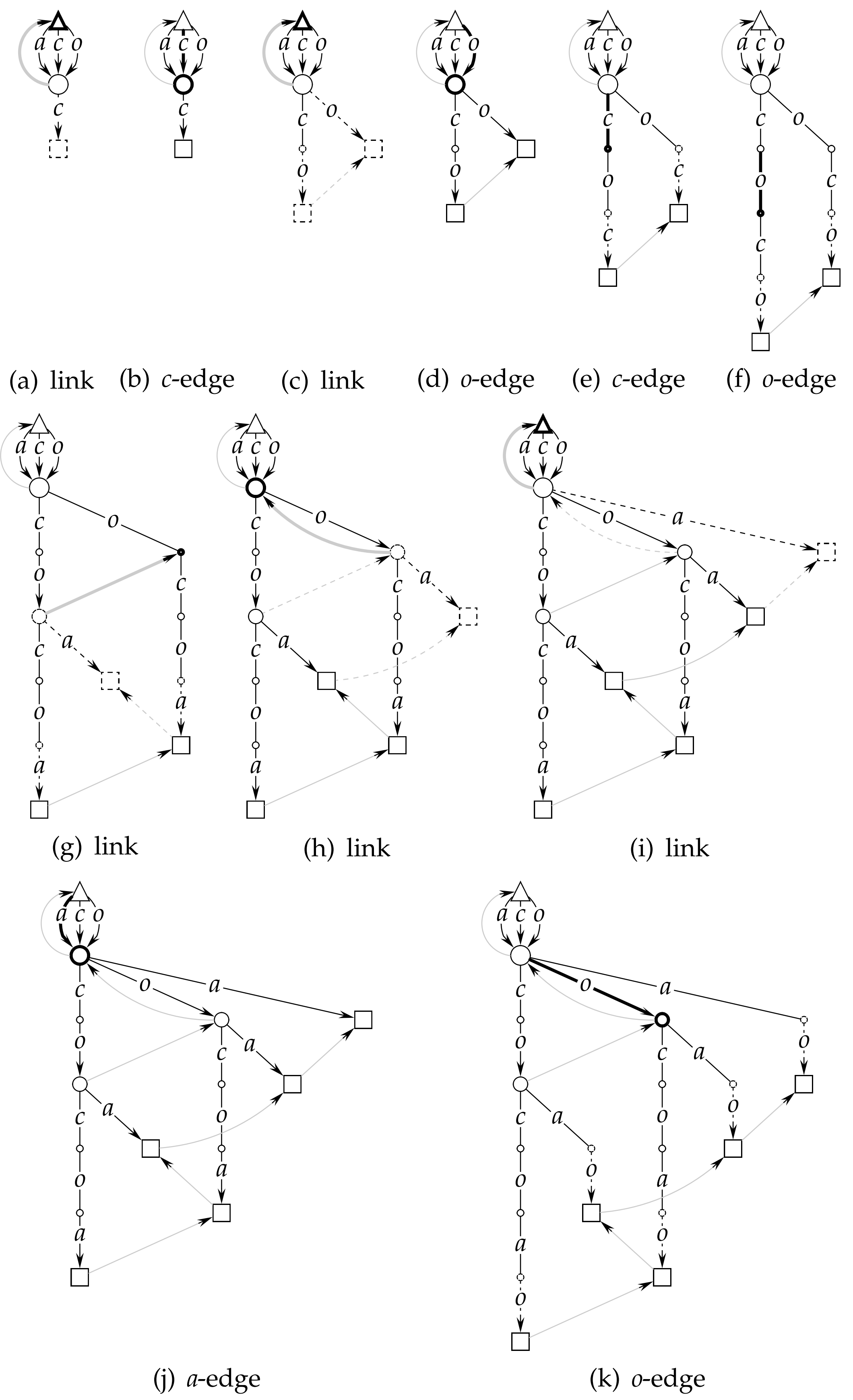


**Figure 5.3.** Step by step construction of SuffixTree(*cocoao*) with action comments. The symbolics used is described in Table 1.1.

of the input string. The instructions saved by this method for our example string *cocoao* read as follows:

$$(c)(o)(2,1)(a)(1,2)$$

`stc_m1`
This method stores the active point movement description split into maximal sequences of downward moves. Such sequence is ended by the first sideways move. Each sequence is described using its length and all the edge choices made in explicit nodes on the way $(2, ao)$. The sideways move that followed this sequence is added to the end of every downward sequence automatically on decompression and is not saved explicitly. The instructions saved by this method for our example string *cocoao* read as follows:

$$(0)(1,c)(3,oc)(0)(0)(2,ao)$$

`stc_m2`
This method changes its behaviour depending on whether or not the active point lies in an explicit node. If it does, then the edge choice made here or a sideways move is saved using tokens used in `stc_u`. Otherwise, a flag signalling whether the next explicit node was reached during the downward movement is saved. If it was not, then the number of moves down from this implicit node is saved as well. This pair is represented by tokens like (false, 0), where the first component is the flag. The instructions saved by this method for our example string *cocoao* read as follows:

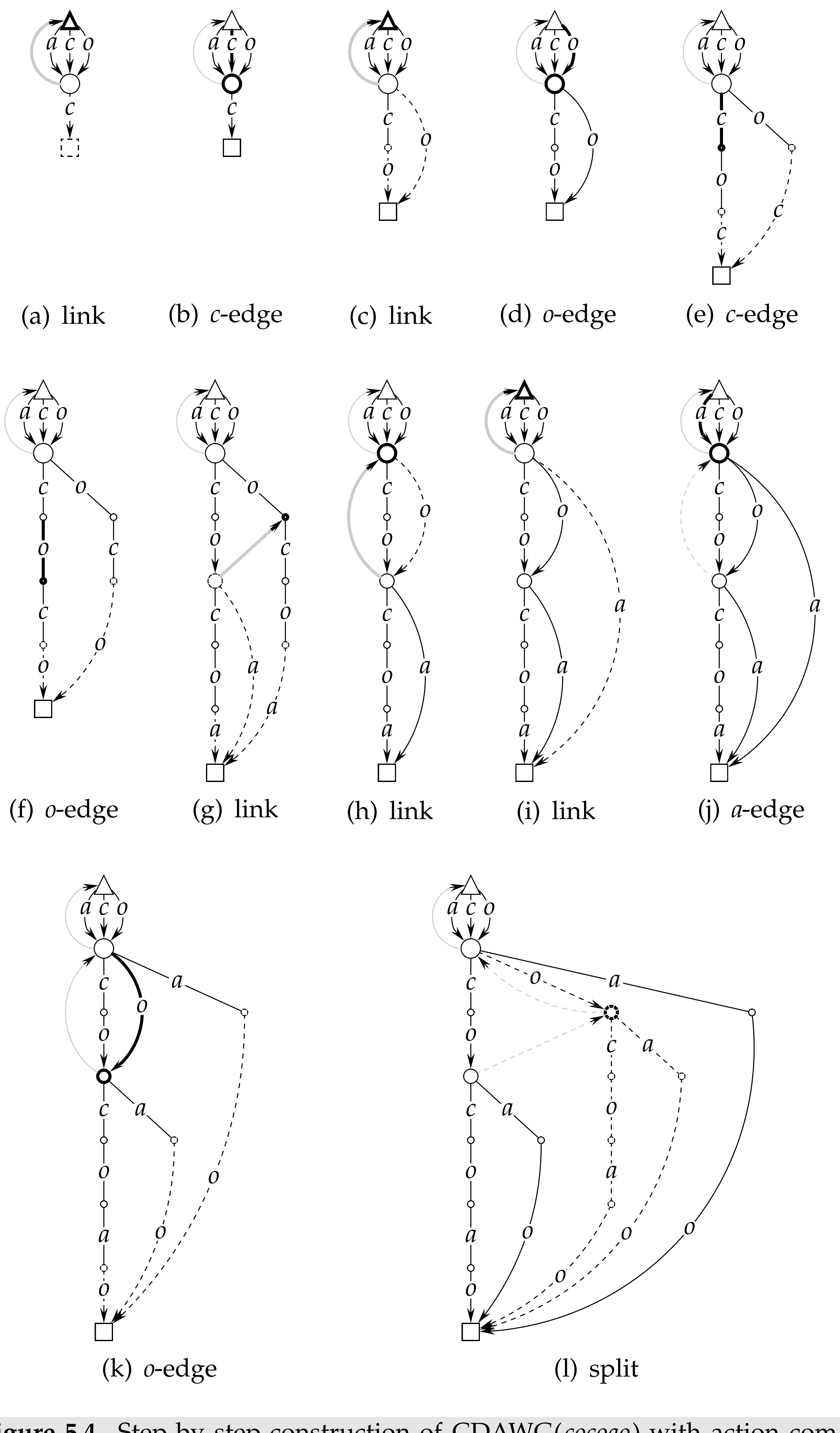
$$(\text{esc})(c)(\text{esc})(o)(c)(\text{false},1)(\text{false},0)(\text{esc})(a)(o)$$

`stc_m3`
This method is a modification of method `stc_m2`. If the target node on the current edge is not reached, this method falls back to a `stc_u`-like decision description. Apart from tokens used by `stc_u`, this method also makes use of a flag token type (false). The instructions saved by this method for our example string *cocoao* read as follows:

$$(\text{esc})(c)(\text{esc})(o)(c)(\text{false})(o)(\text{esc})(\text{false})(\text{esc})(\text{esc})(a)(o)$$

`stc_m4`
This method is a simple modification of method `stc_m1`. After a sideways move places the active point in an explicit node, only the edge or link choice made here is saved in a `stc_u`-like fashion. Otherwise, the behaviour is the same as that of `stc_m1`. The instructions saved by this method for our example string *cocoao* read as follows:

$$(\text{esc})(c)(\text{esc})(o)(2,c)(0)(\text{esc})(a)(1,o)$$


(a) link (b) *c*-edge (c) link (d) *o*-edge (e) *c*-edge

(f) *o*-edge (g) link (h) link (i) link (j) *a*-edge

(k) *o*-edge (l) split

**Figure 5.4.** Step by step construction of CDAWG(*cocoao*) with action comments. The symbolics used is described in Table 1.1.

`stc_m5`

This method saves the identification of the implicit or explicit node where the downward movement sequence stopped. Note that this is similar, but not identical, to the C2 method described by Fiala and Greene [14]. It uses tokens of two types used for explicit and implicit node, respectively. One type stores only the identifier of the explicit node ($[coa]$). The other type type has two components ($[ao], 2$). They are the terminal vertex of the current edge and the distance from this vertex to pinpoint the active point location. Note that this method has to fall back to an `stc_u`-like behaviour in the bot. The instructions saved by this method for our example string *cocoao* read as follows:

$$([\lambda])(c)([\lambda])(o)([cocoa], 3)([ocoa], 3)([\lambda])(a)([o])$$

## 5.4 Suffix Graph Based Data Compression

While the preceding section dealt only with the suffix tree, the other three suffix graph types are very similar and might be used in the same way. As described in Chapter 3: Construction, all four suffix graph types have construction algorithms with a common base. Consequently, all important properties are shared among these algorithms and the active point movement can be used like in the case of the suffix tree based data compression. However, while the conversion to the suffix trie is straightforward, some methods, like `stc_m5`, are not convertible to the DAWG or the CDAWG. Nevertheless, while the simpler edge label of the suffix trie might be useful, the DAWG and the CDAWG are more interesting alternatives due to their common subtree elimination. One of the positive effects this can have is that there can be less suffix links to be traversed when an attempt to move down fails. Note that this is not the case in our example with string *cocoao*, as the subtree merging happens too late to influence the outcome. This is illustrated in Figure 5.4. Consequently, the outputs of all above mentioned methods that can be converted from the suffix tree to the CDAWG, stay the same. However, if we change the string to *cocoacoo*, there will be one less use of a suffix link.

## 5.5 Implementation

So far, all descriptions of our compression methods were high level. The purpose of this section is to fill in the practical details.

As the main memory is not likely to be able to hold a suffix graph for long input strings, its use must be limited. The are basically two choices, one starts from scratch after reaching memory limit, while the other tries to free some space and continue. Since we would like to preserve the information collected in the graph so far, we prefer the second alternative in the form of a perfect sliding window algorithm. However, as shown in Chapter 4: Sliding, this is

| File | Description | Size [B] |
|---|---|---|
| | Calgary Corpus | |
| BIB | Bibliography (refer format) | 111 261 |
| BOOK1 | Fiction book | 768 771 |
| BOOK2 | Non-fiction book (troff format) | 610 856 |
| GEO | Geophysical data | 102 400 |
| NEWS | USENET batch file | 377 109 |
| OBJ1 | Object code for VAX | 21 504 |
| OBJ2 | Object code for Apple Mac | 246 814 |
| PAPER1 | Technical paper | 53 161 |
| PAPER2 | Technical paper | 82 199 |
| PIC | Black and white fax picture | 513 216 |
| PROGC | Source code in C | 39 611 |
| PROGL | Source code in LISP | 71 646 |
| PROGP | Source code in PASCAL | 49 379 |
| TRANS | Transcript of terminal session | 93 695 |
| | Silesia Corpus | |
| DICKENS | Collected works of Charles Dickens | 10 192 446 |
| MOZILLA | Tarred executables of Mozilla 1.0 (Tru64 UNIX edition) | 51 220 480 |
| MR | Medical magnetic resonance image | 9 970 564 |
| NCI | Chemical database of structures | 33 553 445 |
| OOFFICE | A dll from Open Office.org 1.01 | 6 152 192 |
| OSDB | Sample database in MySQL format from Open Source Database Benchmark | 10 085 684 |
| REYMONT | Text of the book Chłopi by Władysław Reymont | 6 627 202 |
| SAMBA | Tarred source code of Samba 2-2.3 | 21 606 400 |
| SAO | The SAO star catalogue | 7 251 944 |
| WEBSTER | The 1 913 Webster Unabridged Dictionary | 41 458 703 |
| XML | Collected XML files | 5 345 280 |
| X-RAY | X-ray medical picture | 8 474 240 |
| | Linux Kernel | |
| LINUX | Tarred source code of Linux kernel 2.4.14 | 126 566 400 |

**Table 5.1.** Test file descriptions.

easy only for the suffix tree as the time complexity of alternatives is quadratic. This led us to the decision to initially limit our tests to suffix tree methods.

Methods `stc_u`, `stc_m1` and `stc_m4` were selected for implementation and practical tests. They maintain a suffix tree for the perfect sliding window using variants of the algorithm described in Chapter 4: Sliding, which employs the batch label maintenance technique. All three methods also produce the active point movement description in tokens described above. However, a way to save these tokens as efficiently as possible is needed. Two components are necessary to make this possible, one is the probability estimation and the other is the coding algorithm. The problem of the coding algorithm is solved using an advanced combination of the Piecewise Integer Mapping arithmetic coder [48] and Schindler's Byte Renormalisation [46]. However, the solution to the probability estimation problem is more complicated.

There are several types of probability estimation contexts to handle. In `stc_u` there are two types of contexts, one in an implicit node and another in an explicit one. The situation in `stc_m1` and `stc_m4` is similar in that there are also contexts for explicit nodes, but the second context type is used for sequence lengths. As we aim for simplicity for these proof-of-concept implementations, we use only a single global context for all lengths. The length frequency counters used for estimation are stored in a Moffat Tree [33] and any length must appear at least twice before it is added with a count of one. Moreover, to estimate the probability of lengths that are not yet present in this structure, we use escape probability estimation technique AX [34]. Note that the match length limit for `stc_m1` and `stc_m4` is set to the sliding window size.

When it comes to the explicit and implicit node contexts, another simple solution is used. Edge usage counter is added to every edge. It is incremented when the edge is chosen during branching and halved during batch edge label update. The probabilities in an explicit node are then estimated using the combination of all out-edge counters and method AX. However, the bot node is a special case as the active point cannot move sideways. In this case all input symbols and the end-of-file symbol are handled as equiprobable. The end-of-file symbol is used to signal the end of the compression/decompression process and is not written out on decompression. In a way similar to the explicit node case, the implicit node probabilities are decided using the current edge counter and the use of an implicit suffix link has a fixed frequency of one. To improve these estimates, the exclusion technique [38, Section 6.3.4] is used to eliminate edges that cannot be selected from the probability computation.

## 5.6 Experiments

Following the analysis of both theoretical and practical issues, it is time for some experiments. Note that the results shown here were previously presented at the Data Compression Conference [41].

| File | bzip2 | ctw | gzip | stc_m1 | stc_m4 | stc_u |
|---|---|---|---|---|---|---|
| MOZILLA | **2.798** | 3.049 | 2.966 | 3.481 | 2.944 | *2.922* |
| MR | 1.958 | **1.827** | 2.947 | 2.656 | *2.104* | 2.438 |
| NCI | **0.432** | 0.519 | 0.712 | 0.641 | *0.465* | 0.484 |
| OOFFICE | 3.722 | **3.669** | 4.018 | 4.333 | *3.844* | 3.853 |
| OSDB | 2.223 | **2.042** | 2.947 | 2.978 | *2.417* | 2.436 |
| REYMONT | 1.504 | **1.238** | 2.198 | 2.131 | 1.697 | *1.677* |
| SAMBA | **1.684** | 1.929 | 2.002 | 2.140 | *1.741* | 1.744 |
| SAO | 5.450 | **5.191** | 5.876 | 6.628 | 5.930 | *5.923* |
| WEBSTER | 1.668 | **1.331** | 2.327 | 2.174 | 1.772 | *1.741* |
| X-RAY | 3.824 | **3.562** | 5.699 | 4.620 | *4.030* | 4.529 |
| XML | 0.660 | **0.643** | 0.991 | 0.935 | *0.704* | 0.720 |
| Avg. | 2.343 | **2.235** | 2.976 | 2.956 | *2.498* | 2.567 |

**Table 5.2.** Silesia Corpus compression results in stored bits per input byte (bpB). Boldface numbers denote the best result of all programs tested, while slanted numbers do the same for STC programs.

Implementations of three different suffix tree based methods, `stc_m1`, `stc_m4` and `stc_u` are compared to three distinct compression methods. Specifically, ZL77, BWC and the Context-Tree Weighting (CTW) [53] methods are used in this comparison. These compression methods are represented by program `gzip` [17], program `bzip2` [45], and program `ctw` [16], respectively. All six programs were compiled using the GNU C/C++ compiler version 3.3.6 and tested under the Gentoo GNU/Linux operating system on a computer with an Athlon 1.3GHz CPU and 768MB DDR RAM.

To get a good picture of the properties our algorithms have, both speed and compression were tested on the following three data sets.

Calgary Corpus
: The Calgary Corpus is a defacto standard lossless data compression test corpus [4]. Its biggest advantage is that there are many results for this corpus available both on-line and in the literature.

Linux 2.4.14
: This dataset is a single file, the TAR archive of the source code of the Linux Kernel version 2.4.14 [50]. It is the largest of all test files in use.

Silesia Corpus
: When compared to the Calgary Corpus, the Silesia Corpus is more recent, much larger and has a wider selection of file types. Another advantage this

| File | bzip2 | ctw | gzip | stc_m1 | stc_m4 | stc_u | lzfg-pm | lzp4 |
|---|---|---|---|---|---|---|---|---|
| BIB | 1.97 | **1.82** | 2.51 | 2.53 | *2.11* | 2.13 | 2.44 | 1.92 |
| BOOK1 | 2.42 | **2.17** | 3.25 | 2.68 | *2.71* | 2.79 | 3.28 | 2.35 |
| BOOK2 | 2.06 | **1.86** | 2.70 | 3.16 | *2.26* | 2.32 | 2.78 | 2.01 |
| GEO | **4.45** | 4.53 | 5.34 | 5.62 | *4.99* | 5.02 | 5.63 | 4.74 |
| NEWS | 2.52 | **2.34** | 3.06 | 2.99 | *2.59* | 2.65 | 3.25 | 2.35 |
| OBJ1 | 4.01 | **3.72** | 3.84 | 4.59 | *4.06* | 4.10 | 4.32 | 3.74 |
| OBJ2 | 2.48 | **2.36** | 2.63 | 2.90 | *2.52* | 2.56 | 3.04 | 2.39 |
| PAPER1 | 2.49 | **2.28** | 2.79 | 2.96 | *2.58* | 2.64 | 2.74 | 2.38 |
| PAPER2 | 2.44 | **2.22** | 2.89 | 3.04 | *2.62* | 2.69 | 2.80 | 2.39 |
| PIC | **0.78** | 0.79 | 0.82 | 1.22 | 0.96 | *0.92* | 0.93 | 0.81 |
| PROGC | 2.53 | **2.33** | 2.68 | 2.96 | *2.60* | 2.66 | 2.74 | 2.39 |
| PROGL | 1.74 | 1.59 | 1.80 | 2.03 | *1.74* | 1.77 | 1.80 | **1.59** |
| PROGP | 1.74 | 1.63 | 1.81 | 1.97 | *1.71* | 1.73 | 1.81 | **1.59** |
| TRANS | 1.53 | 1.39 | 1.61 | 1.71 | *1.46* | 1.46 | 1.67 | **1.34** |
| Avg. | 2.37 | **2.22** | 2.69 | 2.88 | *2.49* | 2.53 | 2.80 | 2.28 |

**Table 5.3.** Calgary Corpus compression results in stored bits per input byte (bpB). Boldface numbers denote the best result of all programs tested, while slanted numbers do the same for STC programs.

corpus has is that results for some of the best methods published so far are available at the corpus site [10].

The name, description and size of all test files are shown in Table 5.1.

Every program was run with its maximum compression settings available enabled. The STC methods were set up to use a sliding window of at most $2^{21}$ bytes, a limit imposed by the memory available for tests.

The compression results for the Silesia Corpus are shown in Table 5.2. Here the `stc_m4` method offers better compression than `stc_u` method, but the difference is less then 0.1 bit stored per byte of input (bpB). However, both of them provide a significantly better compression than the `stc_m1`, as the gap is almost 0.5 bpB. When `ctw` and `bzip2` are added into the comparison, their maturity shows in their results that are 0.263 bpB and 0.155 bpB better than the best STC result, respectively. Finally, `gzip` offers the worst compression and is outperformed by all other compression methods used.

One of the advantages of the Calgary Corpus is that it is a defacto standard and many researchers have published their results for this dataset. This makes it possible to compare the six tested programs with dictionary methods `lzfg-pm` [19] and `lzp4` [5] that are related to the STC family. The results for these

| File | | bzip2 | ctw | gzip | stc_m1 | stc_m4 | stc_u |
|---|---|---|---|---|---|---|---|
| Compression | [bpB] | 1.466 | 1.814 | 1.814 | 1.689 | ***1.398*** | 1.406 |
| Compression | [KiB/s] | 1 392 | 11 | **2 967** | 203 | *205* | 191 |
| Decompression | [KiB/s] | 5 902 | 11 | **63 450** | *213* | 195 | 173 |

**Table 5.4.** Linux 2.4.14 compression and speed results. Boldface numbers denote the best result of all programs tested, while slanted numbers do the same for STC programs.

two methods were added to the results of the six programs tested and put into Table 5.3. Note that results for the six programs tested are very similar to those for the Silesia Corpus. However, this time `stc_m1` came 0.1 bpB closer to other STC methods, but was still worse than `gzip` by 0.19 bpB. The comparison of STC methods with the two related methods reveals that while `stc_u` and `stc_m4` significantly outperform `lzfg-pm` by 0.27 bpB and 0.31 bpB, they lose to `lzp4` by 0.25 bpB and 0.21 bpB.

So far we have concentrated on compression performance, but ignored the compression and decompression speed. This is why the results of the last test contain both, as can be seen in Table 5.4. These tests were performed on the TAR archive of the source code of the Linux Kernel 2.4.14 TAR archive. This test brings a very pleasant surprise, as methods `stc_m4` and `stc_u` win, and even the weakest of STC methods, the `stc_m1`, outperforms both `ctw` and `gzip`. It is likely that the size and structure of this test file favours stable models with slow adaptation and provides enough material for STC methods to learn. Nevertheless, the speed results are much less pleasing. Even though our current implementations of STC methods are almost twenty times faster than `ctw`, they are in turn at least five times slower than both `gzip` and `bzip2` programs.

# 6. Epilogue

This thesis combines suffix graphs with lossless data compression which is not unusual. However, where the traditional approach uses a suffix graph as a tool to implement the required string sorting and searching for selected lossless data compression method, we analyse the suffix graph properties and use them to create new lossless data compression methods. Specifically, the suffix graph construction algorithms and maintenance of suffix graphs in a sliding window are examined thoroughly. The information learnt from this analysis is then used to formulate and study the concept of suffix graph based data compression. Note that selected parts of this work were published previously and are presented here in extended and polished versions [42, 43, 44, 41, 40, 39]. The main contributions of our research may be summarised as follows.

## 6.1 Suffix Graph Construction

As efficient suffix graph construction is required by both a sliding window maintenance algorithm and the suffix graph base data compression, it is examined first. A unified on-line construction algorithm for all four suffix graphs is described and the implicit suffix link simulation is analysed in detail. This analysis yields two original alternatives to the traditional simulation approach that remove a significant portion of branching operations from the construction of suffix tree. Experiments show that as much as sixty six percent of potentially complex and costly branching operations can be removed on real-life data. The same experiments demonstrate that this reduction in branching can result in in up to thirty two percent decrease in execution time needed for suffix tree construction. Nevertheless, both alternatives (`Climb` and `ClimbScan`) require more memory than the traditional method (`ReScan`) when used for construction only. On the other hand, all three techniques need the same amount of space when used in a sliding window setting. Finally, while we show that `ClimbScan` achieves the same linear time complexity as `ReScan`, `Climb` needs at least $\Omega(n^{3/2})$ and at most $O(n^2)$ time in the worst case. Whether or not can `Climb` be forced to a quadratic worst-case time is an open question.

## 6.2 Sliding Window Maintenance

Since the suffix graph based data compression requires maintaining string in a sliding window, an in-depth discussion of sliding window maintenance is needed. This analysis brings the first unified on-line perfect sliding algorithm for all four suffix graphs that includes our original algorithm for the incremental perfect sliding of CDAWG. Moreover, the gaps in previously published proofs on classical percolating update technique are filled, and the suffix tree is verified to be capable of perfect sliding in amortised constant time per one input symbol. On the other hand, the analysis proves that the same cannot be done for CDAWG as it requires time proportional to the sliding window size, like DAWG. This closes the question formulated by Inenaga et al. However, whether there is an incremental algorithm for approximate sliding which requires only amortised constant time per symbol is still an open question.

## 6.3 Suffix Graph Based Data Compression

Building on investigations of construction and sliding window maintenance, we examine the traditional way of using suffix graphs in lossless data compression, where suffix graphs are used like any other string sorting and searching tool. This analysis reveals a surprising connection between the behaviour of suffix graph construction algorithms and two different lossless data compression methods which leads us to design a whole family of compression methods. These new compression methods are based solely on the description of suffix graph construction for the string to be compressed. Some of these methods resemble classical finite context methods like Prediction by Partial Matching or dictionary techniques like ZL77, while other methods appear to be brand-new. To evaluate the practical behaviour of these methods, three of them get proof-of-concept implementations and are tested on standard data corpora. The experiments show that `gzip` and `bzip2` standards outperform our proof-of-concept implementations in terms of speed. However, the results on compression efficiency are more appealing. While the compression ratios reached by our methods on standard corpora are fairly competitive, two of our methods outperform all other tested methods on the largest test file (a Linux kernel source code archive).

## 6.4 Future Work

There are several promising directions for future work on the topics explored in this dissertation. When studying the suffix graphs construction, we designed an alternative approach to implicit suffix link simulation, called `Climb`, which outperforms the traditional solution to this problem in our experiments on real-life data. However, the theoretical analysis reveals that its worst-case time complexity is superlinear. The exact time complexity of the `Climb` procedure is an open problem. So far only upper bounds $O(n^2)$ and $O(n \log(n))$ were obtained on the worst-case time and expected time, respectively (Section 3.2.1.3). Nevertheless, these bounds are not proved to be tight and may be improved upon. Another interesting problem would be to characterise all pathological strings that force Climb to work in a total superlinear time per suffix tree construction, like those in Lemma 3.1.

While investigating the sliding window maintenance, we prove that CDAWG is incapable of perfect sliding in amortised constant time per input symbol, unlike the suffix tree. However, it may be possible to design an approximate incremental algorithm for sliding CDAWG in amortised constant time. As noted in the introduction to Chapter 4: Sliding, there already was an attempt to solve this issue by Inenaga et al. [21], but it was later proved to be invalid by Filip [15].

Finally, when considering suffix data structures, one cannot avoid mentioning the one recently most popular, the suffix array. While it appears to be an ideal tool for our purposes due to space efficiency and simplicity, it is not studied in this work. The reasons are that our data compression applications need to maintain suffix graphs over a text in a sliding window, but the static nature of the suffix array has prevented incremental sliding so far. However, this may change in the future, as there are already attempts to make suffix arrays dynamic, e.g. Salson et al. [37] suggested ways to allow for edit operations on the underlying string.

# Bibliography

[1] ADLER, Mark, Jean-loup GAILLY et al. *Info-ZIP* [online]. 0.8. [cited December 2003]. Available from: `http://www.info-zip.org/`. Version 0.8 was obtained through a private communication with Jean-loup GAILLY.

[2] APOSTOLICO, Alberto and Wojciech SZPANKOWSKI. Self-Alignments in Words and Their Applications. *Journal of Algorithms*. Elsevier, September 1992, volume 13, issue 3, pages 446–467. ISSN 0196-6774. Available from: `doi:10.1016/0196-6774(92)90049-I`.

[3] BALKENHOL, Bernhard, Stefen KURTZ and Yuri M. SHTARKOV. Modifications of the Burrows and Wheeler Data Compression Algorithm. In: STORER, James A. and Martin COHN, editors. *DCC 1999: Data Compression Conference, March 29–March 31, 1999, Snowbird, Utah, [USA]*. Los Alamitos, California: IEEE Computer Society, 1999, pages 188–197. ISBN 0-7695-0096-X. Available from: `doi:10.1109/DCC.1999.755668`.

[4] BELL, Timothy, Ian H. WITTEN and John G. CLEARY. *Calgary Text Compression Corpus* [online]. [cited January 2013]. Available from: `http://corpus.canterbury.ac.nz/descriptions/#calgary`.

[5] BLOOM Charles. LZP: A New Data Compression Algorithm. In: STORER, James A. and Martin COHN, editors. *DCC 1996: Data Compression Conference, March 31–April 03, 1996, Snowbird, Utah, [USA]*. Los Alamitos, California: IEEE Computer Society, 1996, page 425. ISBN 0-8186-7358-3. Available from: `doi:10.1109/DCC.1996.488324`. Available from: `http://www.cbloom.com/src/index_lz.html`.

[6] BLUMER, A., J. BLUMER, D. HAUSSLER, A. EHRENFEUCHT, M. T. CHEN and J. SEIFERAS. The Smallest Automaton Recognizing the Subwords of a Text. *Theoretical Computer Science*. Elsevier, 1985, volume 40, pages 31–35. ISSN 0304-3975. Available from: `doi:10.1016/0304-3975(85)90157-4`.

[7] BLUMER, A., J. BLUMER, D. HAUSSLER, R. MCCONNELL and A. EHRENFEUCHT. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*. New York: ACM, July 1987, volume 34, issue 3, pages 578–595. ISSN 0004-5411. Available from: `doi:10.1145/28869.28873`.

[8] BLUMER, Janet A. How Much is That DAWG in the Window? A Moving Window Algorithm for the Directed Acyclic Word Graph. *Journal of*

*Algorithms*. Elsevier, December 1987, volume 8, issue 4, pages 451–469. ISSN 0196-6774. Available from: `doi:10.1016/0196-6774(87)90045-9`.

[9] CROCHEMORE Maxime. Reducing space for index implementation. *Theoretical Computer Science*. Elsevier, 10 January 2003, volume 292, issue 1, pages 185–197. ISSN 0304-3975. Available from: `doi:10.1016/S0304-3975(01)00222-5`.

[10] DEOROWICZ, Sebastian. *Silesia compression corpus* [online]. [cited January 2013]. Available from: `http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia`.

[11] DIESTEL, Reinhard. *Graph Theory* [online]. 4th edition 2010, Corrected reprint 2012. Heidelberg: Springer-Verlag, July 2010, Graduate Texts in Mathematics, volume 173 [cited December 22 2012]. ISBN 978-3-642-14278-9. Available from: `http://diestel-graph-theory.com/`.

[12] FARACH, Martin. Optimal Suffix Tree Construction with Large Alphabets. In: *Foundations of Computer Science: 38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19–22, 1997*. Los Alamitos: IEEE Computer Society, October 1997, pages 137–143. ISBN 0-8186-8197-7. Available from: `doi:10.1109/SFCS.1997.646102`.

[13] FERRAGINA, Paolo and Gonzalo NAVARRO. *Pizza&Chili Corpus:* Compressed Indexes and their Testbeds [online]. [cited January 2013]. Available from: `http://pizzachili.di.unipi.it/`.

[14] FIALA, E. R. and D. H. GREENE. Data compression with finite windows. *Communications of the ACM*. New York: ACM, April 1989, volume 32, issue 4, pages 490–505. ISSN 0001-0782. Available from: `doi:10.1145/63334.63341`.

[15] FILIP, Ondřej. *Kompaktní sufixový automat v posuvném okně [Compact directed acyclic word graph in sliding window]*. Praha, 27. 5. 2011. Bachelors Thesis. Univerzita Karlova v Praze, Matematicko-fyzikální fakulta, Kabinet software a výuky informatiky. Supervisor Tomáš Dvořák. (in Czech).

[16] FRANKEN, Erik, Marcel PEETERS and T. J. TJALKENS. *CTW* [online]. 0.1b2. [cited December 04 2003]. Available from: `http://www.ele.tue.nl/ctw/`.

[17] GAILLY Jean-loup and Mark ADLER. *gzip* [online]. 1.2.4. [cited November 2005]. Available from: `http://www.gzip.org/`.

[18] GIEGERICH, R. and S. KURTZ. From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction. *Algorithmica*. New York: Springer-Verlag, 1997, volume 19, issue 3, pages 331–353. ISSN 0178-4617. Available from: `doi:10.1007/PL00009177`.

[19] HOANG, Dzung T., Philip M. LONG and Jeffrey Scott VITTER. Dictionary Selection Using Partial Matching. *Information Sciences*. Elsevier, October 1999, volume 119, issue 1–2, pages 57–72. ISSN 0020-0255. Available from: doi:10.1016/S0020-0255(99)00060-2.

[20] HUNT, Ela, Malcolm P. ATKINSON and Robert W. IRVING. A Database Index to Large Biological Sequences. In: APERS, Peter M. G., Paolo ATZENI, Stefano CERI, Stefano PARABOSCHI, Kotagiri RAMAMOHANARAO and Richard T. SNODGRASS (Eds.). *VLDB 2001: Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy* [online]. (San Fransisco): Morgan Kaufmann, 2001, pages 139–148 [cited April 24 2013]. ISBN 1-55860-804-4. Available from: http://www.vldb.org/conf/2001/P139.pdf.

[21] INENAGA, Shunsuke, Ayumi SHINOHARA, Masayuki TAKEDA, Setsuo ARIKAWA. Compact directed acyclic word graphs for a sliding window. *Journal of Discrete Algorithms*. Elsevier, March 2004, volume 2, issue 1, pages 33–51. ISSN 1570-8667. Available from: doi:10.1016/S1570-8667(03)00064-9.

[22] INENAGA, Shunsuke, Hiromasa HOSHINO, Ayumi SHINOHARA, Masayuki TAKEDA, Setsuo ARIKAWA, Giancarlo MAURI and Giulio PAVESI. On-line construction of compact directed acyclic word graphs. *Discrete Applied Mathematics*. Elsevier, 2005, volume 146, issue 2, pages 156–179. ISSN 0166-218X. Available from: doi:10.1016/j.dam.2004.04.012.

[23] KÄRKKÄINEN, Juha. Suffix Cactus: A Cross between Suffix Tree and Suffix Array. In: GALIL, Zvi and Esko UKKONEN (Eds.). *Combinatorial Pattern Matching: 6th Annual Symposium, CPM 95, Espoo, Finland, July 5–7, 1995, Proceedings*. Berlin Heidelberg: Springer-Verlag, 1995, Lecture Notes in Computer Science, volume 937, pages 191–204. ISBN 978-3-540-60044-2. Available from: doi:10.1007/3-540-60044-2_43.

[24] KURTZ, Stefan, Adam PHILLIPPY, Art DELCHER, and Steven SALZBERG. *MUMMER 3+:* Ultra-fast alignment of large-scale DNA and protein sequences [online]. 3.22. [cited August 8 2011]. Available from: http://mummer.sourceforge.net/.

[25] KURTZ, Stephan. Reducing the Space Requirement of Suffix Trees. *Journal of: Software: Practice and Experience*. John Wiley & Sons, Ltd., November 1999, volume 29, issue 13, pages 1149–1171. ISSN 0038-0644. Available from: doi:10.1002/(SICI)1097-024X(199911)29:13<1149::AID-SPE274>3.0.CO;2-O.

[26] LARSSON, N. Jesper. Extended Application of Suffix Trees to Data Compression. In: STORER, James A. and Martin COHN, editors. *DCC 1996: Data Compression Conference, March 31–April 03, 1996, Snowbird, Utah, [USA]*. Los

Alamitos, California: IEEE Computer Society, 1996, pages 190–199. ISBN 0-8186-7358-3. Available from: doi:10.1109/DCC.1996.488324.

[27] LARSSON, N. Jesper. *Structures of String Matching and Data Compression*. Lund, Sweden, September 1999. Ph.D. thesis. Department of Computer Science, Lund University. Available from: http://www.larsson.dogma.net/thesis.pdf.

[28] MANBER, Udi and Gene MYERS. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*. Society for Industrial and Applied Mathematics, 1993, volume 22, issue 5, pages 935–948. ISSN 0097-5397. Available from: doi:10.1137/0222058.

[29] MANSOUR, Essam, Amin ALLAM, Spiros SKIADOPOULOS and Panos KALNIS. ERA: Efficient Serial and Parallel Suffix Tree Construction for Very Long Strings. *Proceedings of the VLDB Endowment* [online]. VLDB Endowment, September 2011, volume 5, issue 1, pages 49–60 [cited May 22 2013]. ISSN 2150-8097. Available from: http://www.vldb.org/pvldb/vol5/p049_essammansour_vldb2012.pdf.

[30] MANZINI, Giovanni and Paolo FERRAGINA. *A Lightweight Suffix Array and BWT Construction Algorithm* [online]. [cited January 2013]. Available from: http://people.unipmn.it/manzini/lightweight/.

[31] MANZINI, Giovanni and Paolo FERRAGINA. Engineering a Lightweight Suffix Array Construction Algorithm. *Algorithmica*. New York: Springer-Verlag, September 2004, volume 40, issue 1, pages 33–50. ISSN 0178-4617. Available from: doi:10.1007/s00453-004-1094-1.

[32] MCCREIGHT, Edward M.. A Space-Economical Suffix Tree Construction Algorithm. *Journal of the Association for Computing Machinery*. New York: Association for Computing Machinery, April 1976, volume 23, issue 2, pages 262–272. ISSN 0004-5411. Available from: doi:10.1145/321941.321946.

[33] MOFFAT, Alistair. An Improved Data Structure for Cumulative Probability Tables. *Journal of: Software: Practice and Experience*. John Wiley & Sons, Ltd., June 1999, volume 29, issue 7, pages 647–659. ISSN 1097-024X. Available from: doi:10.1002/(SICI)1097-024X(199906)29:7<647::AID-SPE252>3.0.CO;2-5.

[34] MOFFAT, Alistair, Radford M. NEAL and Ian H. WITTEN. Arithmetic Coding Revisited. *ACM Transactions on Information Systems*. New York: ACM, July 1998, volume 16, issue 3, pages 256–294. ISSN 1046-8188. Available from: doi:10.1145/290159.290162.

[35] PHOOPHAKDEE, Benjarath and Mohammed J. ZAKI. Genome-scale disk-based suffix tree indexing. In: *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. New York: Association for Com-

puting Machinery, 2007, pages 833–844. ISBN 978-1-59593-686-8. Available from: doi:10.1145/1247480.1247572.

[36] PUGLISI, Simon J., W. F. SMYTH and Andrew H. TURPIN. A Taxonomy of Suffix Array Construction Algorithms. *ACM Computing Surveys*. New York: ACM, July 2007, volume 39, issue 2. ISSN 0360-0300. Available from: doi:10.1145/1242471.1242472. Note: Article No.: 4.

[37] SALSON, M., T. LECROQ, M. LÉONARD and L. MOUCHARD. Dynamic extended suffix arrays. *Journal of Discrete Algorithms*. Elsevier, June 2010, volume 8, issue 2, pages 241–257. ISSN 1570-8667. Available from: doi: 10.1016/j.jda.2009.02.007.

[38] SAYOOD, Khalid. *Introduction to DATA COMPRESSION*. Fourth edition. Waltham: Morgan Kaufmann, 2012. ISBN 978-0-12-415796-5.

[39] SENFT, Martin and Tomáš DVOŘÁK. On-line suffix tree construction with reduced branching. *Journal of Discrete Algorithms* [online]. Elsevier, 2012, volume 12, pages 48–60 [cited January 14 2012]. ISSN 1570-8667. Available from: doi:10.1016/j.jda.2012.01.001.

[40] SENFT, Martin and Tomáš DVOŘÁK. Sliding CDAWG Perfection. In: AMIR, Amihood, Andrew TURPIN and Alistair MOFFAT (Eds.). *String Processing and Information Retrieval: 15th International Symposium, SPIRE 2008, Melbourne, Australia, November 10–12, 2008, Proceedings*. Berlin Heidelberg: Springer-Verlag, 2008, Lecture Notes in Computer Science, volume 5280, pages 109–120. ISBN 978-3-540-89096-6. Available from: doi: 10.1007/978-3-540-89097-3_12.

[41] SENFT Martin. Compressed by the Suffix Tree. In: STORER, James A. and Martin COHN, editors. *DCC 2006: Data Compression Conference, March 28–30, 2006, Snowbird, Utah, [USA]*. Los Alamitos, California: IEEE Computer Society, 2006, pages 183–192. ISBN 978-0-7695-2545-8. Available from: doi:10.1109/DCC.2006.11.

[42] SENFT, Martin. *Bezztrátová komprese dat pomocí sufixových stromů [Lossless Data Compression using Suffix Trees]*. Praha, 12. 12. 2003. Master's thesis. Univerzita Karlova v Praze, Matematicko-fyzikální fakulta, Kabinet software a výuky informatiky. Supervisor Tomáš Dvořák. (in Czech).

[43] SENFT, Martin. Suffix Tree Based Data Compression. In: VOJTÁŠ, Peter, Mária BIELIKOVÁ, Bernadette CHARRON-BOST and Ondrej SÝKORA (Eds.). *SOFSEM 2005: Theory and Practice of Computer Science: 31st Conference on Current Trends in Theory and Practice of Computer Science, Liptovský Ján, Slovakia, January 22–28, 2005, Proceedings*. Berlin Heidelberg: Springer-Verlag, January 2005, Lecture Notes in Computer Science, volume 3381,

pages 350–359. ISBN 3-540-24302-X. Available from: doi:10.1007/978-3-540-30577-4_38.

[44] SENFT, M. Suffix Tree for a Sliding Window: An Overview. In: ŠAFRÁNKOVÁ, J., editor. *WDS'05: Mathematics and Computer Sciences*. Praha: MATFYZPRESS, 2005, pages 41–46. ISBN 80-86732-59-2.

[45] SEWARD, Julian. *bzip2* [online]. 1.0.3. [cited November 2005]. Available from: http://www.bzip.org/.

[46] SCHINDLER, Michael. A Fast Renormalisation for Arithmetic Coding. In: STORER, James A. and Martin COHN, editors. *DCC 1998: Data Compression Conference, March 30–April 1, 1998, Snowbird, Utah, [USA]*. Los Alamitos, California: IEEE Computer Society, 1998, page 572. ISBN 0-8186-8406-2. Available from: doi:10.1109/DCC.1998.672121.

[47] SMYTH, Bill. *Computing Patterns in Strings*. Harlow: Pearson, Addison Wesley, 2003. ISBN 0-201-39839-7.

[48] STUIVER, Lang and Alistair MOFFAT. Piecewise Integer Mapping for Arithmetic Coding. In: STORER, James A. and Martin COHN, editors. *DCC 1998: Data Compression Conference, March 30–April 1, 1998, Snowbird, Utah, [USA]*. Los Alamitos, California: IEEE Computer Society, 1998, pages 3–12. ISBN 0-8186-8406-2. Available from: doi:10.1109/DCC.1998.672121.

[49] TATA, Sandeep, Richard A. HANKINS and Jignesh M. PATEL. Practical Suffix Tree Construction. In: NASCIMENTO, Mario A., M. Tamer ÖZSU, Donald KOSSMANN, Renée J. MILLER, José A. BLAKELEY and K. Bernhard SCHIEFER (Eds.). *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004* [online]. (San Fransisco): Morgan Kaufmann, 2004, pages 36–47 [cited April 24 2013]. ISBN 0-12-088469-0. Available from: http://www.vldb.org/conf/2004/RS1P3.PDF.

[50] TORVALDS, Linus et al. *Linux* [online]. 2.4.14. [cited January 2013]. Available from: http://www.kernel.org/pub/linux/kernel/v2.4/linux-2.4.14.tar.bz2.

[51] UKKONEN, E. On-Line Construction of Suffix Trees. *Algorithmica*. New York: Springer-Verlag, September 1995, volume 14, issue 3, pages 249–260. ISSN 0178-4617. Available from: doi:10.1007/BF01206331.

[52] WEINER, Peter. Linear Pattern Matching Algorithms. In: *SWAT 1973: 14th Annual Symposium on Switching & Automata Theory*. Waltham: IEEE Computer Society, 1973, pages 1–11. Available from: `doi:10.1109/SWAT.1973.13`.

[53] WILLEMS, F. M. J., Y. M. SHTARKOV, and T. J. TJALKENS. The Context-Tree Weighting Method: Basic Properties. *IEEE Transactions on Information Theory*. IEEE Information Theory Society, May 1995, volume 41, issue 3, pages 653–664. ISSN 0018-9448. Available from: `doi:10.1109/18.382012`.

# List of Algorithms

# List of Figures

# List of Tables

# List of Notation

**Strings**

| | |
|---|---|
| $a,\ b,\ \ldots,\ x$ | alphabet symbols |
| $\alpha,\ \beta,\ \ldots,\ \omega$ | strings |
| $\lambda$ | the empty string |
| $\lvert\mu\rvert$ | the length of string $\mu$ |
| $\Sigma$ | alphabet |
| $\Sigma^*$ | the set of all strings on alphabet $\Sigma$ |
| $\Sigma^+$ | the set of all non-empty strings on alphabet $\Sigma$ |
| $\alpha\beta$ | the concatenation of strings $\alpha$ and $\beta$ |
| $\alpha^k$ | the $k$-th power of $\alpha$ |
| $\mathrm{Prefix}(\mu)$ | the set of all prefixes of string $\mu$ |
| $\mathrm{Factor}(\mu)$ | the set of all factors of string $\mu$ |
| $\mathrm{Suffix}(\mu)$ | the set of all suffixes of string $\mu$ |
| $\mu[i]$ | the $i$-th symbol of string $\mu$ |
| $\mu[i\,..\,j]$ | the factor of $\mu$ consisting of symbols $\mu[i]\ \mu[i+1]\ldots\ \mu[j]$ |
| $\mathrm{Occur}^{\mathrm{L}}_{\mu}(\alpha)$ | the set of all positions of left occurrences of $\alpha$ in $\mu$ |
| $\mathrm{Occur}^{\mathrm{R}}_{\mu}(\alpha)$ | the set of all positions of right occurrences of $\alpha$ in $\mu$ |
| $\mathrm{Unique}(\mu)$ | the set of all unique factors of $\mu$ |
| $\mathrm{UniquePrefix}(\mu)$ | the set of all unique prefixes |
| $\mathrm{UniqueSuffix}(\mu)$ | the set of all unique suffixes |
| $\mathrm{LNUP}(\mu)$ | the longest non-unique prefix of $\mu$ |
| $\mathrm{LNUS}(\mu)$ | the longest non-unique suffix of $\mu$ |

|  |  |
|---|---|
| $\mathrm{Context}^{\mathrm{L}}_{\mu}(\alpha)$ | the set of all left contexts of occurrences of $\alpha$ in $\mu$ |
| $\mathrm{Context}^{\mathrm{R}}_{\mu}(\alpha)$ | the set of all right contexts of occurrences of $\alpha$ in $\mu$ |
| $\mathrm{Branch}^{\mathrm{L}}(\mu)$ | the set of all left branching factors of string $\mu$ |
| $\mathrm{Branch}^{\mathrm{R}}(\mu)$ | the sets of all right branching factors of string $\mu$ |
| $\mathrm{Explicit}(\mu)$ | the set of all explicit factors of $\mu$ |
| $\langle\alpha\rangle^{\mathrm{R}}_{\mu}$ | the right extension of factor $\alpha$ in string $\mu$ |
| $\equiv^{\mathrm{R}}_{\mu}$ | the right end equivalence on string $\mu$ |
| $[\alpha]^{\mathrm{R}}_{\mu}$ | the class of right end equivalence on $\mu$ that contains $\alpha$ |
| $(\alpha)^{\mathrm{R}}_{\mu}$ | the longest member of class $[\alpha]^{\mathrm{R}}_{\mu}$ |
| $\mathrm{DC}^{\mathrm{R}}_{\mu}$ | the degenerated class of right end equivalence on $\mu$ |

**Graphs**

|  |  |
|---|---|
| $G = (V, E, L)$ | graph with vertex set $V$, edges set $E$ and suffix link set $L$ |
| $V(G)$ | the set of vertices of graph $G$ |
| $E(G)$ | the set of edges of graph $G$ |
| $L(G)$ | the set of suffix links of graph $G$ |
| $\mathrm{SuffixTrie}(\mu)$ | the suffix trie for string $\mu$ |
| $\mathrm{SuffixTree}(\mu)$ | the suffix tree for string $\mu$ |
| $\mathrm{DAWG}(\mu)$ | the directed acyclic word graph for string $\mu$ |
| $\mathrm{CDAWG}(\mu)$ | the compact directed acyclic word graph for string $\mu$ |

**Other**

|  |  |
|---|---|
| $\dot{\cup}$ | the union of two sets that have an empty intersection |

# List of Abbreviations

| | |
|---|---|
| BWC | the Burrows-Wheeler Compression method |
| BWT | the Burrows-Wheeler Transform |
| CDAWG | the Compact Directed Acyclic Word Graph |
| DAWG | the Directed Acyclic Word Graph |
| PPM | the Prediction by Partial Matching compression method |
| ZL77 | the Ziv-Lempel'77 compression method |

# Companion CD Contents

This thesis is accompanied by a CD with the following contents:

| | |
|---|---|
| `ISLS/` | the source code used for experiments with Implicit Suffix Link Simulation discussed in Section 3.2 |
| `STC/` | the source code used for experiments with Suffix Tree Based Data Compression explained in Chapter 5: Compression |
| `README` | a README file explaining the contents of this companion CD |
| `Thesis.pdf` | a PDF version of this thesis |