Symmetric Index Structures

Highly efficient symmetric text indexing

Daniel Bruder

Munich 2012

Symmetric Index Structures

Highly efficient symmetric text indexing

Daniel Bruder

Thesis
for the acquisition of the degree to
Magister Artium (M.A.)

Centrum für Informations- und Sprachverarbeitung (CIS)
Ludwig-Maximilians-Universität (LMU)
München

Daniel Bruder

September 2012

Primary Reviewer: Prof. Dr. Klaus U. Schulz

Contents

1 Preface

> *Treat the index, not the text!*
>
> [Klaus U. Schulz]

The work presented here is the thesis for the acquisition of the degree to "Magister Artium (M.A.)" at the Ludwig-Maximilians-Universität (LMU München) in Computational Linguistics as a major.

It comprises an intensive period of work that started following a visit to the Bulgarian Academy of Sciences in Sofia whose efforts form the ground base of this work.[1]

After the visit to Sofia, proof of concept implementations to acquire familiarity with the variety of intricate algorithms and different types of indexing structures used in this work followed.

Next up, wrapping and extending the existing code base to extend the current library to be a document-indexing automaton followed.

The points presented here summarize the efforts and want to help successive work on the library. To make the understanding of the complex algorithms as easy as possible for new readers in the first steps, the presentation is intentionally kept simple where possible. The goal decidedly is to "get to point across" to as many readers as possible and to stay clear and comprehensible in the presentation. In all places references are given to the original papers, mathematically sound definitions and additional links for further reading and presentations from different perspectives.

This work is mostly based on work by Klaus Schulz et al. ([10]), as will become clear in the following.

[1] please see "Acknowledgements" and "Thanks"

2 Motivation and Goals

The beginnings of the theory of finite state automata essentially date back to the 1940s and 1950s when researchers began work on structures and machines following Turing's preliminary work on the so called "Turing machines" in the 1930s. Although these machines were initially intended to simulate human brain functions, they soon proved to be of great use elsewhere…[2]

Despite less concrete work on the *theory* of finite state automata itself in recent years, the basic concepts of finite state automata theory form the crucial backbones of many of today's technologies (find a compilation of a subset of the uses for finite state automata in Uses for Automata).

Following [14], in today's formal training, the approach to finite state automata is a more pragmatic one. May the situation be as it is, I feel lucky that, in the formal training in Computational Linguistics at the Munich LMU University, I could find a lot more than only pragmatic approaches to this, admittedly, at times perplexing theoretic field through Prof. Klaus U. Schulz and his research assistants.

The theory of finite state automata, for sure, can be a very complex topic at times, but, it needs to be said, that albeit its complexity it nevertheless builds on only a small set of common ground compared to the many tasks where it can successfully be put to use. Thus, the theory of finite state automata draws its immense value from the vast fields where it can be applied successfully in conjunction with the (finite) set of theory as a foundation.

2.1 Uses for Automata

Index structures such as the ones described in this work and in the papers referenced, can be used to effectively store and *index* strings, along with useful additional information that can be of great use in a wide range of applications and a diverse set of mathematical problems on strings.

In order to arrive at the carefully woven index structure, to incorporate, for instance, all subparts of a text, different structures have been studied and algorithms developed.

The different structures that have been developed which can serve a diverse set of purposes and can be distinguished by their different properties and thus prove to be suitable in various degrees for different tasks at hand.

Among the wide range of applications that Finite State Automata (in the following: FSA[3]) in the form of index structures like Suffix Trees / Suffix Tries, Directed Acyclic

[2] See [14, chapter 1, "Automata: The Methods and the Madness"]

[3] also check the Glossary section in the Appendix

Word Graphs (DAWGs), Compacted Directed Acyclic Word Graphs (CDAWGs) can be used for, one can find – at least

- information retrieval
- bioinformatics
- pattern matching / regular expressions
- data compression
- spell checking
- OCR correction

From another level, these structures can be used to give answers to a manyfold of combinatorial problems on strings like[4]

- the "pattern p in text t"–problem, i.e. string search, in $\mathcal{O}(m)$ complexity, where m is the length of the current sub-string of the pattern searched for (with initial $\mathcal{O}(n)$ time required to build the structure for the string)
- the "finding the longest common substring" problem
- the "finding the longest repeated substring" problem
- the "finding the longest palindrome in a string" problem

2.2 A motivating example

To introduce the topic of this thesis and to ensure that all readers are on the same page, a very illustrative example is chosen. Feel free to skip to the next section, if you are familiar with the theory and foundation of FSA – you will find the concrete goals of this work summarized in the next section.

Imagine, the words of our language (the text t) that are to be recognized were $L = \{cocoa, cola\}$, and the pattern p that is searched for were *cola*.

Imagine further, that the words which are recognized as valid words of the language were given as a plain text file with one word per line, sorted alphabetically:

```
cocoa
cola
```

[4] see, for example, [27] for an overview of applications

Now, to decide whether the pattern $p = cola$ is part of our language *without using an automaton*, in the worst case, one would have to compare each and every word in the text file with the pattern. In this case pictured here, this lookup would lead to two comparisons, asking whether the input pattern is equal to the current line of our dictionary. Only in the last of these string comparisons however, do we find that, indeed, the word *cola* is part of our language and thus is "accepted" as a valid word of the language given.

However, if there were a million words in our language and in alphabetical order "cola" was the last word in that file, one million lookups and comparisons would be needed to decide whether that word (pattern) is part of the particular language L.

Of course, the pattern we are looking up could be *not* part of our language, and if so, in the current example, this would mean to make a million lookups and comparisons in the first place – only to find out after the last unsuccessful comparison that the search pattern is not part of the language.

Obviously, an approach like the one described is grossly ineffective. As was said, using automata, the lookup can be carried out in $\mathcal{O}(m)$ time w.r.t. to length of the pattern (and with the initial $\mathcal{O}(n)$ time overhead to build the automaton w.r.t. to the input length of the text t). This means that, if no words with an initial letter c exist in our language, we could stop searching for the pattern *cola* right away in the very first step. Compare that to a million (potentially useless) lookups, or worse, maybe 10 million or more – depending on the language L.

These algorithmic questions are exactly where Finite State Automata come into play. We can use an automaton as an index structure to represent our language L and to execute the search for the pattern p on that text base. If we choose a standard Trie[5] as our structure to represent the language $L = \{cocoa, cola\}$, it would look like the one in figure 1.

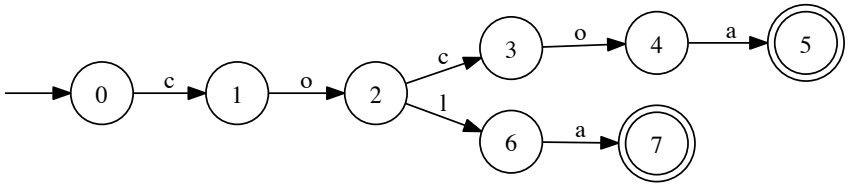

Figure 1: Example Automaton $Trie(L)$ for $L = \{cocoa, cola\}$

[5] a directed tree for text re*trie*val

Recognizing *cola* as the pattern to decide whether it is recognized and thus "part of the language L", we would, for example,

- go from the dedicated *"start state" 0* to state *1* by using the first letter of the pattern, c in one step
- from there, in a second step, move on to state *2* with the second letter o. State *2* does have a transition with the next letter l, so this *transition* is followed to state *6*
- finally, in a last step, the "accepting" *final state 7* is reached (the double circles indicate that this is a final, accepting state). This is the indication that the search pattern was "matched successfully".

Conversely to the example given above, even if there were a million words in our language, the steps needed to recognize a word of that language would still be $\mathcal{O}(m)$, i.e. the matching procedure for the search pattern *cola* would need four steps – or less if it was not part of our language. In effect, only exactly as many steps as the pattern's number of characters are needed to successfully recognize that word. In other words: the matching procedure is made independent of the number of words of that language.

Where this pays off even more is in the second scenario given above: trying to match a word that is *not* part of the language (hence: *"not accepted by the automaton"*). In this toy example, if the pattern was, instead of *cola*, the word *water*, the automaton simply would not have any path to follow the start state and thus recognition is aborted *in the very first step*. In other words again, the pattern *cannot be recognized* and thus it is clear that the pattern p is *not part of the language* L of which the automaton is a representation of.

Now, going one step further: imagine we wanted to match suffixes of the language. Take, for example the suffix *oa* of the word *cocoa*, i.e. let the pattern be *oa*. It is easy to see that *oa* is not part of our language L, $oa \notin L$ *where* $L = \{cocoa, cola\}$. Speaking in terms of the automaton, there is no transition with letter o from the *start state* (as was the case when the search pattern was $p = water$) to begin with, so recognition is abandoned in the first step.

If we were to recognize even subparts of the language given, we would have to build a *superset* of that language first.

Let's look at an example automaton of a superset of L, incorporating the *suffixes* giving us $L_{Suf(cocoa) \cup Suf(cola)} = L_{Suf(cocoa)} \cup L_{Suf(cola)}$ with $L_{Suf(cocoa)} = \{cocoa, ocoa, coa, oa, a\}$ and $L_{Suf(cola)} = \{cola, ola, la, a\}$ resulting in $L = \{cocoa, ocoa, coa, oa, cola, ola, la, a\}$, depicted by figure 2 on the next page.

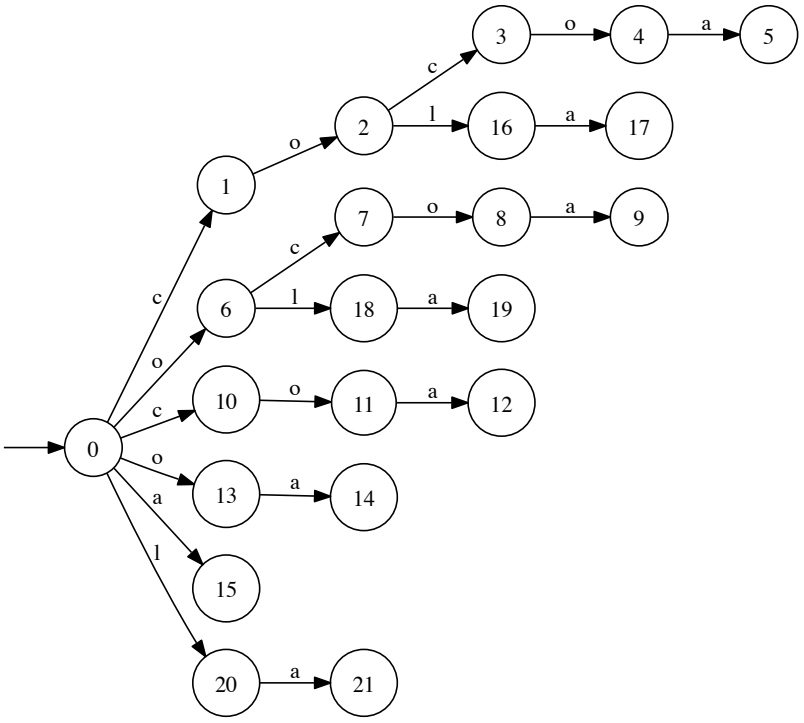

Figure 2: Example Automaton *SuffixTrie*(L) for $L = \{Suf(cocoa), Suf(cola)\}$

It is easy to see, that now, the pattern p = oa is recognized by the automaton, i.e. the *suffix "oa"* part of the language, $p \in L_{Suf(cocoa,cola)}$.

From this standard point in the theory of finite state automata several paths to go further are possible:

- What if we wanted to "walk" with a certain pattern, e.g. *co* and get suggestions on possible extensions of the current pattern? In other words the automaton would respond with all possible extensions giving us all extensions that can still lead to a successful match. For instance, for the beginning *co* the automaton would not only return the obvious extensions "**co***coa*" and "**co***la*" but also the suffix "**co***a*" as a possible extension – since it is part of our language $L_{Suf(cocoa,cola)}$.
- Another possible way to take this further: What if we wanted to add full sentences or whole text bases (like articles, books, complete works) and work on that level? What could suggestions look like and how would the indexing of them, the building of the structure work? Which structure and construction algorithm would prove as the most suitable for this task?
- For the case that we were to add whole texts instead of words only: How could one build a search engine / text lookup and retrieval system in which the user could *a)* enter a certain part of a pattern — *b)* get suggestions — *c)* enter more subparts of the text — *d)* follow a certain part of the given suggestions again — *e)* modify the search pattern (and so forth) until finally reaching an accepting state?
- Additionally: How could the structure be used to store information in order to be able to point the user back to that specific subpart of text that she derived from the previous steps? How can the user be pointed to the occurrence of that specific pattern she chose?
- How can one build the even smaller and more space efficient equivalent representation of the language as in 3 on the following page, cutting down on the number of states from 21 to 4 and on the number of transitions from 21 to 7 only?
- Since most of these question have already been solved and answered in the articles referenced in the bibliography, one thing that has not been done yet: How would suggestions *in both directions* work, i.e. how could the user be provided with additional suggestions not only "to the right" (as is known from search engines and search fields), but also "to the left" and thus get more specific context-like suggestions? What would the index structure suitable for

this kind of task look like and how would one go about constructing and using it in the most efficient way?

- How can one build the smallest possible structure for this task *directly*, i.e. *without the help of intermediate structures*?
- Additionally: How can one do all these steps *symmetrically*, i.e. construct an automaton capable of giving suggestions for both directions *directly* and in the *most compact way possible*?

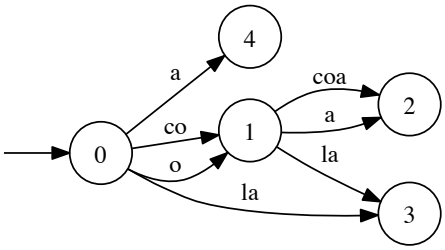

Figure 3: Example Automaton $CDAWG(L)$ for $L_{Suf(cocoa,cola)} = \{cocoa, ocoa, coa, oa, a, cola, ola, la, [a]\}$

Luckily, solutions to all these questions will be given in this work.

2.3 Goals of this work

Ultimately, what we want to achieve in this work, is to build an index structure, that will

- take a set of files containing "running text" (as opposed to words of a dictionary only)
- index all subparts (factors, "infixes") of each text file
- allow pattern search on the infixes in $\mathcal{O}(m)$ complexity, where m is the length of the current sub-string of the search pattern (with the build time being linear $\mathcal{O}(n)$ w.r.t. to the input length of the text t)
- allow the pattern search in *both directions*: in "natural" left-to-right direction as well as a right-to-left direction
- give suggestions on possible *concrete* extensions of the given pattern p to both these directions (as opposed to suggestions classically known from search engines that are *statistical* and only "to the right")

- additionally keep an inverted file index of all these concrete factors keeping track of their respective occurrences and concrete positions in the given text files and return them for the current pattern found
- be the smallest automaton possible
- be constructed in linear time and space complexity w.r.t. to the input length and directly.

To this end I will describe

- how to obtain a Symmetric Index Structure that is suitable to index the entirety of the text base's subparts (in this case, *infixes*)
- that, by its nature, can be used to give suggestions to both sides using the deterministic extensions present in the symmetric automaton
- how to apply Blumer, Blumer et al.'s "complete inverted files" algorithm to the symmetric automaton (in linear time).

2.4 Organization of this work

Chapter 3 will set a baseline for the rest of this work by introducing the basic definitions of Finite State Automata, regular languages, the indexing of words, of suffixes, and supersets of languages.

Chapter 4, "Indexing all subparts of a text directly", will first begin to describe how all suffixes of a text can be indexed in a suffix tree (similar to figure 2 on page 10) *directly* and *on-line* using Ukkonen's clever algorithm from [23].

In the chapter following, it is shown how to use Inenaga et al.'s algorithm that builds the *smallest automaton possible* of the same set of suffixes *directly* (Chapter 5, "Building the smallest automaton possible").

Chapter 6, "Constructing the symmetric automaton" will then go on to describe how to obtain the *symmetric* index structure that incorporates *all suffixes* of the *natural reading direction* of the text and all suffixes in the *reversed direction* directly and on-line.

Furthermore, this section will then describe how to *identify* the resulting equivalent states of both the automata gained in the first step, through the algorithm developed by Schulz et al. ([10]) to build a fairly novel structure, the SCDAWG.

Building on this is chapter 7, in which it is described how to use and apply the ideas by Blumer, Blumer et al. to attach additional "inverted file" information to this smallest possible symmetric structure obtained in the previous step (compare [2]).

Following a code usage example on how to use the current library in chapter 8, chapter 9, "Adapting the existing automata code base" will go into details of the implementation of the algorithms covered and will discuss the refactoring process of the existing code base that this work builds on.

Finally, metrics (chapter 10) are presented as well as ideas and hints for future work (chapter 11).

Please see the Appendix section for definitions and additional information on terminology and structures used throughout the text and for additional indexes to the tables and figures that illustrate this work. Additionally, links to the sources and a detailed installation procedure are given.

3 Index Structures

While there is a wide family of index structures, all of them are composing Finite State Automata theory of some sort with a few differences and thus are intimately inter-related.

The theory of finite state automata is central to each of the index structures described in the following. Hence, we will cover the essential basics first, then progressing on to describing the commonalities and differences between the different variants of index structures and how to make use of them for the task at hand: indexing all factors of a set of running text in a symmetric structure enabling us to give suggestions to the left and to the right of a current search pattern p by having these present in our symmetric structure deterministically.

In order to finally obtain the SCDAWG structure that is needed, we will follow the description of the related automata in the following chain:

SuffixTrie → SuffixTree → DAWG → CDAWG → SCDAWG

Table 1: Progression of index structures

3.1 Technical Preliminaries

Next follow the essential definitions that will be needed in the sequel. These are meant to serve as a common ground where, in the following, only the differences between the structures are presented. As far as the deeper specifics of individual structures are concerned, the presentation will focus on the differences only and pointers to the original references and, at times, more formally tight definitions will be given.

3.1.1 Notation

As far as the notation is concerned, it will be kept close to the notation found in [10, pp. 3].

Let Σ denote a finite alphabet, and words P, U, V, W denote words over Σ^*. The empty word is written as ϵ and single letters of Σ are written as σ, and σ_i; $W = \sigma_1 ... \sigma_n$.

W^{rev} denotes the reversed word $\sigma_n ... \sigma_1$. The length (the number of symbols) of a given word W is given by $|W|$.

The notation UV is used for the concatenation $U \circ V$. A string U is called a *prefix* (resp. *suffix*) of $W \in \Sigma^*$ iff W can be represented in the form $W = U \circ V$ (resp. $W = V \circ U$) for some $V \in \Sigma^*$.

A string is an *infix* of $W \in \Sigma^*$ iff W can be represented in the form $W = U_1 \circ V \circ U_2$ for some $U_1, U_2 \in \Sigma^*$. The set of all strings over Σ is denoted Σ^* and the set of nonempty strings over Σ is denoted Σ^+.

A *lexicon* or *dictionary* is represented by a nonempty collection $\mathcal{D}$ of words, a *text* $\mathcal{T}$ is a concatenation of strings over Σ^* with length $|\mathcal{T}| > 0$, a *text base* $\mathcal{B}$ is a (nonempty) collection of texts $\mathcal{T}$.

The set of all infixes, suffixes, prefixes in $\mathcal{B}$ is $Inf(\mathcal{B})$, $Suf(\mathcal{B})$, $Pref(\mathcal{B})$.

The size of the text base $\mathcal{B}$ is $\|\mathcal{B}\| = \sum_{T \in \mathcal{B}} |\mathcal{T}|$.

3.1.2 Definitions

To begin with, definitions are given with respect to the structures and concepts used in the following. The list here is not exhaustive in the sense that, in the sequel, additional definitions will be given where needed.

Finite State Automaton. Classically, (see, for instance, [16, 17, 10, 14, 24, 3]), a basic Deterministic Finite State Automaton (DFA) is a quintuple, consisting of $\mathcal{A} = \{Q, \Sigma, s, \delta, F\}$ where[6]

- Q is a finite set of states
- Σ is a finite input alphabet
- s is a start state
- δ is the partial transition function $\delta : Q \times \Gamma \to Q$
- $F \subseteq Q$ is the set of final states

Regular language. A Finite State Automaton represents a regular language in an algebraic form (compare, among many, [14, chapter 3]) and thus can "recognize" a language.[7]

A regular language is defined as "a formal language that can be expressed using a regular expression". See, e.g. [26] for the formal definition of a regular language. The interesting part is that

- a regular language satisfies the property (among other properties), that it can be accepted by a deterministic finite state automaton
- a regular language can be expressed through a regular expression

[6] compare Gerdjikov et al. [10], Definition 2.1

[7] For an example of a recognition process, see again, section 2.2, "A motivating example".

- all finite languages are regular languages – to prove that a language is regular, one often uses the "Myhill-Nerode theorem" or the "pumping lemma" among other methods[8].

Relationships between automata and languages. The crucial point here is that, in every case, a regular language can be represented by a Finite State Automaton and a regular expression and vice versa in all directions, compare the table "Relationships".

Regular language $\rightleftarrows$ Regular expression $\rightleftarrows$ Finite State Automaton

Table 2: Relationships

Trie. The trie data structure can be seen as the common ground for all other subsequent tree like data structures.

A trie is an ordered tree structure that is used to store a dynamic set or associative array where the keys are usually strings. It is also known as a "Prefix Tree"[9].

For example, the Trie depicted in figure 2 on page 10 for the lexicon $\mathcal{D} = \{Suf(cocoa), Suf(cola)\}$ is a

> "deterministic finite-state automaton $Trie(\mathcal{D}) = (Q, \Sigma, q_\epsilon, \delta, \{q_U \mid U \in \mathcal{D}\})$ where $Q = \{q_U \mid U \in Pref(\mathcal{D})\}$ is a set of states indexed with the prefixes in $Pref(\mathcal{D})$ and $\delta(q_U, \sigma) = q_{U \circ \sigma}$ for all $U \circ \sigma \in Pref(\mathcal{D})$"[10].

Suffix Trie. A suffix trie is a special trie in the sense that it indexes the set $Suf(\mathcal{D})$ and thus recognizes all *suffixes* of $\mathcal{D}$.

Formally, $Trie(Suf(\mathcal{D})) = STrie(\mathcal{D}) = (Q, \Sigma, q_\epsilon, \delta, F)$ where the set of final (i.e. accepting) states is $F = \{q_U \mid U \in \mathcal{D}\}$.

The size of the suffix trie is bounded by $Suf(|\mathcal{D}|)$ and thus *quadratic* w.r.t $|\mathcal{D}|$; for every $n \in \mathbb{N}$ it is $(n+1)^2$.[11]

Suffix tree. A suffix tree is a specialized suffix trie in the sense that can be seen as the compacted version of the aforementioned Suffix Trie, which requires only linear space w.r.t. $|\mathcal{D}|$.

> "Suffix tree $STree(T)$ of T is a data structure that represents $STrie(T)$ in space linear in the length $|T|$ of T. This is achieved by representing only a subset $Q' \cup \{\perp\}$"[12]

[8] see, for example, Wikipedia [26]
[9] compare Wikipedia [28]
[10] Gerdjikov et al. [10], Definition 2.3
[11] compare Gerdjikov et al. [10], Definition 2.4
[12] Ukkonen [23], p. 6

Note that the additional state $\perp$ is a special "bottom state" that makes the construction easier:

> "Auxiliary state $\perp$ allows us to write the algorithms in the sequel such that an explicit distinction between the empty and the nonempty suffixes (or, between *root* and the other states) can be avoided. State $\perp$ is connected to the trie by $g(\perp, a) = root$ for every $a \in \Sigma$. We leave $f(\perp)$ undefined"
>
> (Compare [23, pp. 3]. Read δ for g and "Suffix Link" *sl* for f).

Compacted Automaton. A compacted automaton represents a compacted version of a "standard" automaton. The accepted language remains identical, but the compacted automaton features a smaller set of states compared to its counterpart. This is achieved by removing explicit states of "out-degree 1" making them *implicit* and introducing a *generalized transition function*.

Compare the compacted Suffix Tree of figure 5 on page 27 with the Suffix Trie of figure 2 on page 10 to get the idea behind the compaction – keep in mind the latter automaton additionally represents the suffixes of the word *cola*, but still, the side-by-side comparison should make the point of compaction clear. Figure 4 on page 25 gives an overview of the relationships of the different types of automata.

Minimized Automaton. A minimized automaton is a deterministic finite state automaton with the least possible states. A deterministic automaton is minimized by identifying non-distinguishable states through their *right-language* and by removing all states that are non-reachable or "trap states" (states from which there is no escape). Compare the minimized equivalent automaton of figure 17 on page 38 with the compacted automaton of figure 5 on page 27. Figure 4 on page 25 gives overview of the relationships of the different types of automata. The definitions needed for the minimized automaton are as follows.

Right-language. Two states can be considered non-distinguishable when their right-language is identical. The right-language of two states is identical when both states have the same "finality" (both are final states or both are non-final states), the same amount of transitions, all transitions have the same symbols, and the same needs to hold true for all states that are reachable from this pair of states.

The right language $\overrightarrow{L}$ of state q is given by (using a *generalized transition function* $\hat{\delta}$):

$\overrightarrow{L}(q) = \{w \in \Sigma^* \mid \hat{\delta}(q, w) \in F\}$

Recursively defined (with a unique solution) it is:

$$\overrightarrow{L}(q) = \{a\overrightarrow{L}(\delta(q,a)) \mid a \in \Sigma \wedge \delta(q,a) \neq \bot\} \cup \begin{cases} \{\epsilon\} & \text{if } q \in F \\ \emptyset & \text{otherwise} \end{cases}$$

Compare these definitions with [7].

Reachability. The next property needed for the minimal automaton is the property of all states being reachable from the *root state* (cf. [7]):

$Reachable(\mathcal{A}) := \forall_{q \in Q} \exists_{w \in \Sigma^*} (\hat{\delta}(q_0, w) = q)$

Minimalitity. A deterministic finite state automaton $\mathcal{A}$ is defined as minimal, if:

$Minimal(\mathcal{A}) := (\forall_{q,q' \in Q} (q \neq q' \Rightarrow \overrightarrow{L}(q) \neq \overrightarrow{L}(q'))) \wedge Reachable(\mathcal{A})$

Thus, all states of the automaton are reachable and no two non-distinguishable states p and p' exist for which the right language is the same.

Compare these definitions with [7].

Explicit state. A branching state, i.e. a state from which there are at least two outgoing transitions.[13]

Implicit state. The term "implicit state" refers to a position on a compacted edge in a deterministic automaton. It is only an intuitive helper but does not actually exist as a dedicated state in the automaton. It is generally referred to using a tuple $(root, c, 1)$, indicating that the point referred to is the point reachable from the (explicit) root state, following the transition beginning with letter c just after the first letter.

One could say that an implicit state is a (fictional) state from which there is only one outgoing transition, a "state of out-degree 1".[14]

Equivalence class. When building equivalence classes, one builds a partition over a set X in which several specific elements x, x' are considered "equivalent" to each other with respect to the equivalence relation R. In our case we build the equivalence classes w.r.t. to the equivalence relations $\overleftarrow{=}$, $\overrightarrow{=}$ and $\overleftrightarrow{=}$ to gain the equivalence classes $[\overleftarrow{X}]$, $[\overrightarrow{X}]$ and $[\overleftrightarrow{X}]$. The special signs # and \$ found throughout this work are classic boundary helper symbols in the theory of finite state automata. They are not part of the Alphabet Σ and, in the following, will be left out where presentative purposes allow to do so. Otherwise, they are considered inherent.

[13] compare Ukkonen [23], pp. 6
[14] compare Ukkonen [23], p. 7

Example of equivalence classes. The equivalence classes that will be built and used in this work will be represented by the states in the minimized automata. For example, the equivalence classes of the automaton of figure 3 on page 12 consist of: $0 = \{\epsilon\}, 1 = \{co, o\}, 2 = \{cocoa, ocoa, oa, a\}, 3 = \{cola, ola, la\}, 4 = \{a\}$ with the reason being the definition of equivalence w.r.t. $\leftrightarrows$:

Two suffixes are considered part of the same equivalence class, when their set of *end-positions* are equal, which for instance, can easily be seen in the equivalence class $3 = \{cola, ola, la\}$. All these suffixes have the same set of end-positions, whereas the suffix a is in an equivalence class of its own, since its set of end-positions contains both: the end-position in *cocoa* and the end-position in *cola*.

For the mathematically sound definitions, please compare [10, Definitions 5.1 and 5.2 and pp. 14-19].

Canonical representative. The canonical representative of an equivalence class as built in the previous example simply is the *longest member of the class*. That is, the canonical representatives concerning the classes given above are: $0 = \epsilon, 1 = co, 2 = cocoa, 3 = cola, 4 = a$. For the mathematically sound definitions, please compare [10, Proposition 5.3 and p. 14] or [16, Definition 3].

DAWG. For a definition of the *Directed Acyclic Word Graph* compare, e.g. [10, Definition 5.8].

CDAWG. Please find the definition for the *Compacted Directed Acyclic Word Graph* in section "5.1 Definition of the CDAWG" on page 38 of this work or consult [10, Definition 5.9].

SCDAWG. Please find the definition for the *Symmetric Compacted Directed Acyclic Word Graph*, the SCDAWG, in section "6.1 Definition of the SCDAWG" on page 46 of this work or confer to [10, Definition 5.10].

Inverted file. Also referred to as "inverted index" – "an index data structure storing a mapping from content, such as words or numbers, to its locations in a database file, or in a document or a set of documents"[15]. The purpose of an inverted index is to allow fast full text searches.

3.2 Indexing all subparts of a text

In order to obtain a symmetric index structure suitable to incorporate all subparts, i.e. *infixes* of a given set of texts $Inf(\mathcal{B})$ (as opposed to all infixes of a given set

[15] cf. Wikipedia [25] for an overview and for a tighter definition: Blumer et al. [3], "Abstract" and "Introduction"

of words, as found in a dictionary $\mathcal{D}$) and giving suggestions in the search/lookup-phase in both directions, one needs to build a Finite State Automaton in form of a *Symmetric Compacted Directed Acyclic Word Graph* (SCDAWG).

The SCDAWG index structure's language symmetrically incorporates all unique infixes of a set of texts $Inf(\mathcal{B})$, such that the states represent the infixes as *equivalence classes* w.r.t. $\leftrightarrows$. This way, *both directions* are covered by indexing the set of all suffixes of the natural left-to-right (LTR) reading direction $Suf(\mathcal{B})$ as well as the set of all suffixes of the right-to-left (RTL) direction $Suf(\mathcal{B}^{rev})$ followed by a bijection function identifying identical states in both these sets.

The index structure can be queried for a pattern p – to finally point the user to the respective locations of the pattern, one needs to additionally save *"inverted file" information* attached to the states corresponding to the offsets of the pattern in the set of text files.

3.2.1 General Overview of Indexing Steps

On the construction-side, the general steps to be taken can essentially be broken down to the following:

(1) *Construct* a suitable one-directional index structure in the natural *left-to-right* (LTR) reading direction for the set of suffixes $Suf(\mathcal{B})$

(2) *Construct* a suitable one-directional index structure in the opposite *right-to-left* (RTL) direction for the set of suffixes $Suf(\mathcal{B}^{rev})$

(3) *Compact* and *minimize* both these automata

(4) *Identify the equivalent states* of both automata

(5) *Attach additional "inverted file" information* to one of the underlying automata (naturally most likely the LTR directed automaton, although it is irrelevant which automaton is chosen)

As far as the user interface side is concerned, to be able to give suggestions in both directions, one will need to

(1) Carry out reading in the current pattern p

(2) user will accept (deterministic) suggestions to the left (!) and to the right of the current pattern or type more parts of her pattern

(3) (recursively follow steps (1) and (2) until user chooses specific match)

(4) point user to all pieces of text where pattern p occurs

Additional points regarding the construction. On the construction side, the separate automata generated in steps (1) and (2) will best be constructed in a compacted manner directly by using an on-line algorithm.

Inenaga et al. ([16]) give a nice description how to perform a so-called *on-line construction* to obtain the *Compacted Directed Acyclic Word Graph* (CDAWG) *directly* based on ideas of Ukkonen's on-line algorithm.

Make sure to check Ukkonen's description ([23]) of the direct construction of suffix trees and the section "Ukkonen's algorithm". Ukkonen gives a nice description of his Suffix Tree construction algorithm that will make understanding the Inenaga algorithm a lot easier.

For the CDAWG construction algorithm check the description in [16] and the section "The Inenaga algorithm for CDAWGs". The algorithm proposed by Inenaga et al. proves to be interesting in that it constructs the smallest automaton possible directly using key ideas that were presented by Ukkonen's direct Suffix Tree construction algorithm.

With step (3) covered, step (4), the identification of equivalent states, will use the algorithm described by Schulz, Mihov et al. ([10]).

It is fascinating and interesting to note already here, that both automata, the *LTR-Compacted Directed Acyclic Word Graph* (gained from step (1)), and the *RTL-Compacted Directed Acyclic Word Graph* (gained from step (2)) which are now combined to form the *Symmetric*-CDAWG, the SCDAWG, have the nice property of having *the same set of identifiable states* – even though they individually indexed $Suf(\mathcal{B})$ and $Suf(\mathcal{B}^{rev})$ respectively.

Finally, for step (5), Blumer & Blumer's idea of attaching "inverted file"-information to the automaton's states (compare [3] and [2]) is used and applied to the symmetric structure obtained through the previous steps.

Also interesting to note in this place already is, that all of these automaton construction steps described can be carried out in linear time and space w.r.t. to the length of the input string(s) (or document(s)) and, as was mentioned before, the resulting automaton will be the smallest automaton possible (compare with [3] and [10]).

Especially in a case where all factors, i.e. supersets over a text base $\mathcal{B}$ are indexed, e.g. for the complete works of an author or the complete set of a newspaper's articles from a range of several years are to be indexed, these are very desirable properties.

3.3 Comparison and highlights of the index structures used in the following

As was mentioned before, the different index structures can be characterized by their different properties and, by this, are suitable for different tasks.

3.3.1 Properties

For our task at hand, and following the comparison of index structures in [16, p. 4], we will add to it the SCDAWG structure with the following properties:

- construction on-line and directly – i.e. without the need to rebuild the whole structure when a new set of texts is added and with the structure being readily constructed up to the current point at all times.
- in linear time and space
- for multi-strings (in this case "running text" from the text base $\mathcal{B}$ as opposed to a dictionary $\mathcal{D}$)
- with additional inverted file information (pointing to all occurrences of the pattern p in the text data base $\mathcal{B}$).

3.3.2 Suffix Tree

The suffix tree construction for $Suf(\mathcal{B})$ as described by [23] has the following properties:

Property	Ukkonen SuffixTree
on-line construction	✓
construction in linear-time	✓
linear-space consumption	✓
smallest possible automaton	×
symmetric structure	×

Table 3: Comparison of properties of Ukkonen's Suffix Tree construction

3.3.3 CDAWG

The description of the CDAWG follows [16]. The CDAWG described by Inenaga et al. for $Suf(\mathcal{B})$ has the following properties:

Property	CDAWG
on-line construction	✓
construction in linear-time	✓
linear-space consumption	✓
smallest possible automaton	✓
symmetric structure	×

Table 4: Comparison of properties of Inenaga et al.'s CDAWG construction

3.3.4 SCDAWG

The full algorithm for the direct construction of an SCDAWG as described in this work was first presented in [10], following [17].

Essentially, this algorithm fills the gap of the outline to a symmetric structure which was given in [17] as a sketch[16].

The indexed factors are $Inf(\mathcal{B})$. Additionally, inverted file information will be added to the resulting structure.

Property	SCDAWG
on-line construction	✓
construction in linear-time	✓
linear-space consumption	✓
smallest possible automaton	✓
symmetric structure	✓

Table 5: Comparison of properties of Schulz / Mihov et al.'s SCDAWG construction

3.3.5 Graphical Overview of structure relationships

The graph in figure 4 on the following page illustrates the relationships between the Suffix Trie, Suffix Tree, DAWG and CDAWG structures, check [16, p. 3] for a more detailed account of the relationships.

[16] compare Gerdjikov et al. [10], p. 17

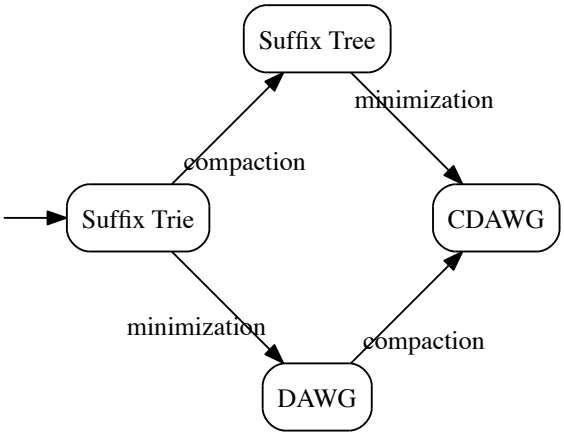

Figure 4: Graphical overview of structure relationships

4 Indexing all subparts of a text directly

As was pointed out briefly in section "General Overview of Indexing Steps", ultimately, to arrive at a symmetric index structure which represents all infixes $Inf(\mathcal{B})$ of a set of texts, in a first step, two independent one-directed Automata are used to form, in a consecutive second step, the *Symmetric Compacted Directed Acyclic Word Graph* Automaton, referred to as the "SCDAWG".

The steps involved in indexing all *infixes* of a text are based on indexing all *suffixes* (i.e. "factors") of a text – one each for both reading directions LTR and RTL –, followed by an identification step afterwards (refer to sections "Building the SCDAWG" and "Algorithm for the bijection" respectively for this identification step).

As was mentioned in section "General Overview of Indexing Steps", Inenaga et al. ([16]) give an on-line algorithm for obtaining a (one-directed) CDAWG index structure directly, based on Ukkonen's on-line algorithm used for the construction of another related index structure, the *Suffix Tree*.

Since Inenaga's algorithm works so closely with the ideas of Ukonnen's algorithm, we will describe Ukkonen's algorithm first and progress on to describing the CDAWG construction algorithm.

Keep in mind, at the end, two such Inenaga-CDAWGs will be combined to form the Symmetric CDAWG, the SCDAWG – find an illustration of the SCDAWG's structure in figure 25 on page 47 and an example SCDAWG automaton in figure 26 on page 48.

4.1 Indexing all suffixes

The suffixes of a word are denoted by $Suf(w)$.

Imagine you had the word `cocoa`, then, the set of suffixes $Suf(cocoa)$ to be indexed would consist of:

```
cocoa
 ocoa
  coa
   oa
    a
```

The easiest and most straight-forward structure to represent this language[17], is a suffix tree / suffix trie that has all these suffixes added to its structure.

[17] cf. the definition of a regular language in the definitions section

As can be told from figure 5, this automaton accepts, as its language, exactly this same set of suffixes. For instance, to recognize the suffix *coa*, it could go to state *3* from the *start state* and from there go on to state *4*, by which the suffix is accepted. Similarly the same holds for the suffix *ocoa*: The automaton would simply progress along the path (State 0 → State 5 → State 2).

Also obvious in this sense should be that, for instance, a suffix *ox* would not lead to a successful recognition. From one perspective, simply because it is not part of the language L that this automaton is a representation of. From another perspective the alleged suffix cannot be recognized, because there is no transition to be taken for the letter x after having progressed with letter o to state *5*. In other words again, simply the alleged suffix *ox* is not part of the set $Suf(cocoa)$.

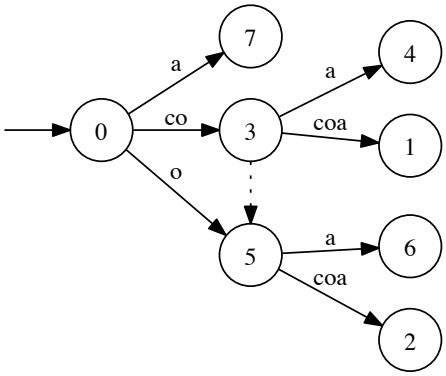

Figure 5: Suffix Tree for $L = \{cocoa, ocoa, coa, oa, a\}$

4.2 Ukkonen's algorithm

What Ukkonen's algorithm achieves is to build the suffix tree index structure in linear-time and directly, and, additionally, as far as the understanding of the algorithm itself is concerned, it does this in a more "natural" left-to-right direction.

Ukkonen's algorithm is a successor to Weiner's method [24] that proceeds right-to-left and adds the suffixes to the structure in increasing order of their length and also to McCreight's algorithm [18], that adds the suffixes to the tree in decreasing order of their length.[18] Both, Weiner's and McCreight's algorithm, do not have the online-ness-property.[19] Essentially this means that all words that are to be added need to be known beforehand. Make sure to confer to [11] for an in-depth comparison of similarities and differences between these three Suffix Tree Construction algorithms.

[18] compare Ukkonen [23], p. 2

[19] compare Inenaga et al. [16], p. 4 and the Glossary section in the Appendix

Although all of Ukkonen's, Weiner's and McCreight's construction algorithms ultimately build one and the same structure, Ukkonen's algorithm tries to give a more "natural" idea by processing the string symbol by symbol from left to right while having the current tree readily available at any time during the construction.

Ukkonen's key idea is to obtain all suffixes of a string T by concatenating t_i to the end of each suffix of $\mathcal{T}^{i-1}$ and by adding an empty suffix[20]:

$$\mathit{suffix}(\mathcal{T}^i) = \mathit{suffix}(\mathcal{T}^{i-1})t_i \cup \{\epsilon\}$$

The essential "key" tricks used in Ukkonen's algorithm are:

- the "active point", keeping track of the location where the next insertion will have to be made from
- a counter called "remainder", keeping track of the remaining suffixes that need insertion at a later step
- implicit states, i.e. "states of out-degree one"[21]
- symbol by symbol adding of suffixes (see above)
- suffix links, as known from the Aho-Corasick algorithm ([1])
- open edges – as defined below
- data and pointers into data: "data" is one long string of the original text base $\mathcal{B}$. Transitions are merely pointers to these memory locations
- edges once created will not "move" again, rather, they will only be split (and thus make the tree structure "finer grained"). Leaf nodes always stay leaf nodes and keep to the rule "once a leaf, always a leaf".

4.2.1 Suffix Tree construction

What follows, is a hands-on presentation of this algorithm with illustrations to give a better understanding of each of the steps taken in the construction.[22] Please refer to [23] for the exact details and pseudo code – the approach taken here is to give an intuitive understanding of the steps taken in the construction.

[20] Ukkonen [23], p. 5

[21] compare Definitions

[22] inspired by the excellent answer to the question "Ukkonen's suffix tree algorithm in plain English?" on `http://stackoverflow.com` by "jogojapan", see: stackoverflow/jogojapan [22]

Adding *cocoa* Step 1, adding letter c from position 1 of the input string *cocoa*.

In the first step, the first symbol is added to the structure, in this case, letter c (compare figures 6 and figure 7), beginning from the root state *0*.

Since this is the first step, in the structure, no transition with c is present from the *root state*, and thus, a new edge from the *root state* is created to a newly formed *"sink state"*, state 1.

By using *open edges* and, by this, *pointers into the text*, it is not exactly the letter c, that is added, but merely two pointers saying $[1, \#]$, meaning that this edge covers the string from position 1 in data to the (open) end #. Currently, since we are in *step 1*, the open edge represented by # is $\# = 1$.

Although, up to this point, only the letter c of *cocoa* was added, effectively, still the "rest" of the word is, so-to-say, already present in the structure *implicitly*: an open edge represents the whole rest of the word up to the current point. Ultimately, the current point will move to the last position of $\mathcal{T}$, $t_{|\mathcal{T}|}$, i.e. the whole word will get added and the open edge # will be pushed and increased to $|\mathcal{T}|$.

The open edge based representation of the current structure is illustrated by figure 6:

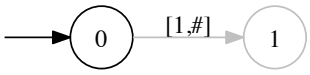

Figure 6: Suffix Tree construction step for $L = \{c(ocoa)\}$ (in open edge representation)

Effectively, the current stage in "string representation" (i.e. with "interpreted edges") is illustrated by figure 7 (the *ocoa*-part in parentheses represents the implicitly-available parts that will ultimately be added by extending the open edge step by step):

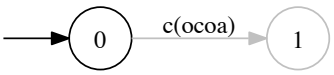

Figure 7: Suffix Tree construction step for $L = \{c(ocoa)\}$

Adding *ocoa* Step 2, adding letter o from position 2 of the input string *cocoa*.

In the next step of the algorithm, depicted by figure 8, the next letter o is added, and thus, through open edge representation, the suffixes up to the current point t_i, *o(coa)* and *co(coa)* are added to the structure.

Since there is no transition that starts with letter o from the root state *0* (to which the algorithm brought us back to in the previous step) which could be followed, a new transition $[2, \#]$ is created, that points to a newly created state, state *2*.

Again, the two different representations are presented by figures 8 and 9.

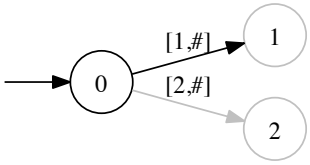

Figure 8: Suffix Tree construction step for $L = \{co(coa), o(coa)\}$ (in open edge representation)

Note that in step *2*, the edge $0 \rightarrow 1$ is automatically extended to represent *co* instead of *c* by virtue of open edges, i.e. by setting the current open edge boundary marker $\# := 2$.

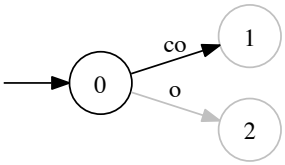

Figure 9: Suffix Tree construction step for $L = \{co(coa), o(coa)\}$

Adding *coa* Step 3, adding letter c from position 3 of the input string *cocoa*.

In the third step, a new phenomenon occurs: Again, starting from the root state, letter c is to be added to the structure. Now, that a transition with c already exists from the *root state*, several different steps from the ones seen before happen:

- we will not create a duplicate transition, but rather, remember that we were to add the suffix *coa* to the tree

- instead of creating such a new (open) edge for the letter c, we follow the already existing transition and thus move forward to the position *right after* the letter c in our current structure to an *implicit state*
- since this *implicit state* carries no state identification number of its own, we refer to the position that the next insertion will have to happen from by using the tuple $(root, c, 1)$. This is actually moving the *active point* away from the *root state* where it used to be in the previous steps.

Again, to remember that we were to add the suffix *coa* and the current point from where the next insertion will have to happen from, we make use of the *active point*: by setting it to the tuple $(root, c, 1)$, we to refer to this *implicit state*. In the next step t_i the check whether there already exists a transition with the letter to be added will be carried out from this (implicit) state.

The tuple $(root, c, 1)$ specifically points to the position starting from root, with letter c, at the position just-past position 1 on that edge.

The variable *remainder*, which used to be 1 in all previous cases (but was not mentioned yet) is increased to 2, since we did not really insert anything in this step, but rather only moved the active point and have one more remaining factor left to insert in one of the next steps.

That is, the insertion of the suffix *coa* is not carried out directly at this point but *postponed to a later step*. Rather than inserting anything in this step, pieces of information are saved for a later step, by moving the active point and increasing the counter *remainder*.

As was done in the previous steps however, the value of the open edge pointer # is set to *3*, giving us the following picture representation (figure 10):

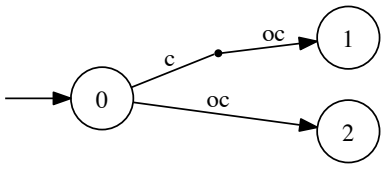

Figure 10: Suffix Tree construction step for $L = \{coc(oa), oc(oa)\}$. Factor $c(oa)$ remaining to be inserted in a later step, the active point on implicit state $(root, c, 1)$ is represented by the dot.

Adding *oa* Step 4, adding letter o from position 4 of the input string.

This time, after having moved the *active point* to the position $(root, c, 1)$, we start the regular insertion process from this implicit state, rather than from *root* (as we have done before, without knowledge about the active point at that time).

Again, similar to the previous step, we see that there is no edge to be added, because there already is an edge that starts with o from the *active point* $(root, c, 1)$.

As before, we carry out several steps to be able to add the suffix oa at a later time:

- we move the active point tge position just past the letter that already exists from the current node, $(root, c, 2)$
- increment *remainder* to the value of *3*
  - to remember that we were about to add a factor of our input string, but did not actually carry out the insertion
  - because a transition with the letter to be added already existed (the fourth letter o of our input string and thus the suffix oa of our input string).
- Again, the open edge pointer is extended to represent the current construction process – we arrive at the current structure as represented by figure 11.

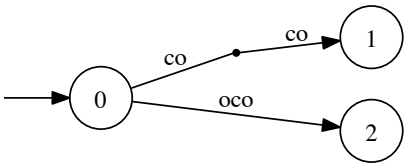

Figure 11: Suffix Tree construction step for $L = \{coco(a), oco(a)\}$ (with factors $co(a), o(a)$ remaining)

Adding *a* Step 5, adding letter a from position 5 of the input string *cocoa*.

In the final position, the suffix a needs to be added and by this, the final letter a needs insertion to the structure.

This last step indeed is an interesting step: Again, we are adding the letter starting from the *current active point*, which is, resulting from the previous step $(root, c, 2)$,

and our remainder is still at *3*, which means that the suffixes that are waiting to be added are *coa*, *oa*, and now, *a*.

Ultimately, since this is the last step, we will have to add all of the remaining suffixes that had been postponed to a later step to the structure (*oa*, and *coa*) at some point before leaving the construction phase.

First, like in all previous steps, we increase the boundary of the open edge, making the structure look as depicted by figure 12:

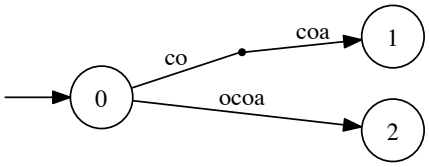

Figure 12: Suffix Tree construction step 1 for $L = \{cocoa, ocoa\}$

Next, we check if there already exists a transition with letter *a* from the active point. There is none, so *a* can be added directly at this point.

However, since our active point is placed on an *implicit state*, we need to `split` the edge first before adding a new transition and thus have to make this implicit state an *explicit state* (remember, explicit states are states with more than one outgoing edge).

In this sense, the *implicit state* where the *active point* is located, is `split` (made explicit) and a new transition with letter *a* is appended to this newly formed state (state *3*), pointing to another newly created state, state *4*.

Compare the current situation of the structure with figure 13 on the following page.

Now that we have added *coa* as one of the remaining suffixes and extended the open edges by setting # to the value *5* – and thus extended all existing edges in only one step to represent *cocoa*, *ocoa* and *coa* respectively – we still need to add the remaining suffixes *oa* and *a*.

In the step that we have just carried out – adding *coa* – we decrement the remainder counter to a value of *2* and move the *active point* to the next position where the adding of the remaining suffixes will have to start from. In this case, the next remaining suffix to be added is *oa*.

The rule for a situation like this is the following: If there is a *suffix link* pointing out from the node we have done the `split` and the insertion of a new edge from,

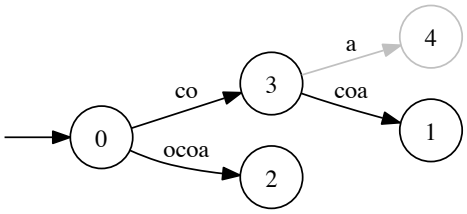

Figure 13: Suffix Tree construction step 2 for $L = \{cocoa, ocoa, coa\}$. Suffixes *oa* and *a* waiting for insertion

we follow this suffix link. If there is no such *suffix link* from the current point, the *active node* is set back to *root*.

Although *suffix links* have been mentioned before in the characteristics section of the Ukkonen algorithm, only now do we first encounter them.

Suffix links are another key point of Ukkonen's algorithm and are very suitable for something that could be described as a sort of "backtracking" into the remaining suffixes to be added from our stack of remaining suffixes[23].

The concept of suffix links is very useful and the rule for suffix links can be summed up like the following [compare [22], and figure 14 on the next page]:

> "If we split an edge and insert a new node, and if that is not the *first node* created during the current step, we connect the previously inserted node and the new node through a special pointer, a **suffix link**." ([22])

In our case at hand, we are still in step *5*, and this is our first `split` and insertion of a new node – so no suffix link had been created before or gets created at this point.

Instead, as a rule, on the first split, the *active node* is reset to *root* and follows the first transition found from there, of the next suffix we want to insert – if there is one such transition to follow.

Since in this current case the suffix to be inserted next from our stack of remaining suffixes is *oa*, the *active node* is set to $(root, o, 1)$ – compare with figure 14 on the following page to see the current location of the active point (indicated by the dot).

From here, again an implicit state, we can `split` the edge, make the active point an explicit node (state *5*), and insert a new transition with the letter *a* of our suffix

[23] also referred to as "failure transitions" in [11] and [17]. Also see the algorithm by [1] and its use of suffix-links in the Unix-classic `fgrep` program.

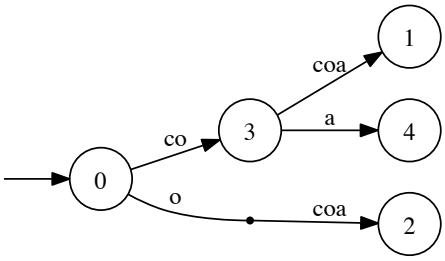

Figure 14: Suffix Tree construction step 3 for $L = \{cocoa, ocoa, coa\}$

to add, *oa* to another newly formed state, state *6*, just as we have done previously for the suffix *coa*. Figure 15 illustrates this step.

Conversely to the previous `split` however, since this is our second `split` in the current step *5*, we have to *insert a suffix link from the previously created node* that was created during a `split` to the newly created node (compare with the rule above). Thus, our structure looks like the following (figure 15):

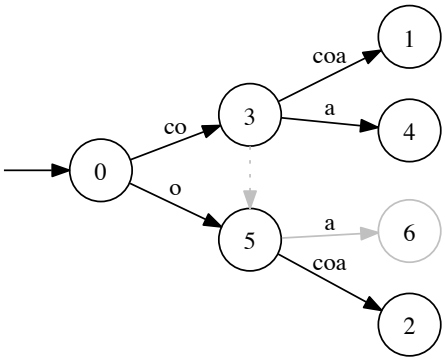

Figure 15: Suffix Tree construction step 4 for $L = \{cocoa, ocoa, coa, oa\}$

If there was a suffix link from the current point of insertion, we would have to follow it and set the active point to it. Since this is not the case, for the last remaining suffix *a*, we will set the *active point* back to the *root node*, and, from there, since no transition starting with *a* exists, we will create a new transition to a newly created state, state *7*.

Finally, the resulting structure looks like the structure represented by figure 16 on the next page and the algorithm is finished with the construction.

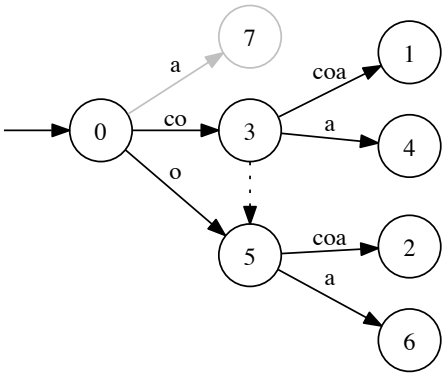

Figure 16: Suffix Tree construction step 5 for $L = \{cocoa, ocoa, coa, oa, a\}$

The points to keep in mind are:

- The moving of the active node across the structure
- The updating of the open edges by setting # to the current point of insertion and how it can guarantee the linear time complexity of the algorithm.
- the pointers into data and how it helps to guarantee the linear space complexity of the algorithm
- How the "once a leaf node, always a leaf node"-rule applies: merely does the tree structure get finer grained but transitions to leaf nodes that have existed will be kept as they are – only splits may occur and leaf nodes once created will stay untouched.

4.2.2 More resources

For descriptions from other perspectives, please confer to these excellent resources, linked to in the bibliography, which are: Ukkonen's own account of the algorithm in [23] and two further online resources, which give more perspectives on this beautiful algorithm: [22] and [20].

5 Building the smallest automaton possible

As the table "Progression of index structures" has pointed out, we will next cover the construction of the CDAWG structure. What we have seen so far was how to reduce the automaton size by building a Suffix Tree instead of a Suffix Trie to index all suffixes of a text.

Compare figures 2 on page 10 and 3 on page 12 to see the reduction in states and transitions needed to represent the same language through two different-sized – but, nevertheless w.r.t. to the language represented, equivalent automata.

In this section we will further reduce the Suffix Tree as seen in the previous chapter to a CDAWG-automaton which is the smallest automaton possible for the same language L.

Looking at the suffix tree's graph of figure 5 on page 27, two important things can be noted:

a) to further reduce the automaton's size to obtain the smallest automaton possible, some states could be "folded", "merged" together to save space without changing the language L of the automaton: for instance, some states having transitions ending in *a* and *coa* in the graph 5 on page 27 could be summed up and be represented through one single state only, since their right-language[24] is the same to form the equivalent automaton of figure 17 on the following page. Effectively, this means a *converging, a "sharing" of common suffixes.*

b) compared to the Suffix Trie of figure 2 on page 10, some states can be saved by making transitions hold multi-letter units instead of single letters and thus making some states *implicit*. In figure 5 on page 27 this was already taken care for through Ukkonen's algorithm for the language $L = \{Suf(cocoa)\}$.

It is important to note, that what step a) involves is a *minimization*, and what step b) accomplishes is a *compaction*.

Consequently, the tree depicted in figure 5 on page 27 already is a *compacted* automaton.

The *minimized and compacted version* of said trie is the so called CDAWG structure and looks like the automaton shown in figure 17 on the following page.

The *minimization and compaction* of the suffix trie is what makes the CDAWG, the *compacted directed acyclic word graph*. The language of all three types of structures is the same and thus the automata are equivalent in terms of the language they represent.

[24] cf. the definition of right-language in the Definitions section.

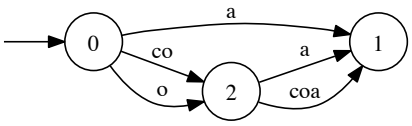

Figure 17: CDAWG for $L = \{cocoa, ocoa, coa, oa, a\}$

Now, to obtain a CDAWG structure from a suffix tree, two obvious ways will form viable paths:

a) First minimizing and then compacting the suffix trie

b) first compacting and then minimizing the suffix trie.

Inenaga et al. ([16, p. 3]) give a nice illustration of these relationships concerning the transformations between the structures – figure 4 on page 25 gives a small example illustration of the same figure.

Finally, in order to keep space requirements linear at all times and not having to construct a potentially quadratic size structure during construction time, Inenaga et al. ([16]) present an on-line, direct algorithm to arrive at the CDAWG structure directly, i.e. without the need for any such intermediate structures.

5.1 Definition of the CDAWG

The CDAWG automaton is the same as defined in [2], [16] and [10] as the tuple $\overleftarrow{C}(\#D\$) := (Q_{\overleftrightarrow{C}}, \Sigma_{\#\$}, \overleftrightarrow{[\epsilon]}, \delta_{\overleftrightarrow{C}}, F_{\overleftrightarrow{C}})$, where

- $Q_{\overleftrightarrow{C}}$ is the set of all equivalence classes $\overleftrightarrow{[V]}$ w.r.t. $\overleftrightarrow{\equiv}$,
- the start state is $\overleftrightarrow{[\epsilon]} \in Q_{\overleftrightarrow{C}}$,
- the (partial) transition function $\delta_{\overleftrightarrow{C}} : Q_{\overleftrightarrow{C}} \times \Sigma^{+}_{\#\$} \to Q_{\overleftrightarrow{C}}$ is defined for a string of the form $\sigma U (\sigma \in \Sigma_{\#\$}, U \in \Sigma^{*}_{\#\$})$ iff $\overrightarrow{\overleftrightarrow{X} \circ \sigma} = \overleftrightarrow{X} \circ \sigma U$. The value is $\delta_{\overleftrightarrow{C}}(\overleftrightarrow{[X]}, \sigma U) = \overleftarrow{[\overleftrightarrow{X} \circ \sigma]}$
- the set of final states is $F_{\overleftrightarrow{C}} := \{\overleftrightarrow{[V]} \mid \overleftrightarrow{[V]} \cap \#D\$ \neq \emptyset\}$.

Compare these definitions here with [10, p. 16] and [2].

5.2 The Inenaga algorithm for CDAWGs

The main difference between Inenaga et al. ([16]) and Crochemore's algorithm ([6]) is the property of *on-line*-ness, i.e.

a) the strings / set of strings does not need to be known beforehand

b) the structure is readily constructed up to the current point at all times

c) in the case of updating the structure with a new suffix or a new string that was not known / had not been known beforehand, there is no need to rebuild the whole structure from scratch (cf. [16, p. 33]).

The main difference of Inenaga et al.'s algorithm to Blumer et al.'s algorithm ([3]) is, that Inenaga et al. ([16]) build the CDAWG *directly*, i.e. without the need of an intermediate DAWG structure that is, only in a second step, shrinked down to a CDAWG by eliminating all out-degree-one nodes from it[25]. The point here is, that, in order to build a space and time economical structure (take the case of combinatorial algorithms on DNA with a huge set of factors) it sometimes is strictly not possible to build a quadratic-size in-between structure to finally obtain from it the desired structure by compacting and minimizing it only in a second step.

The construction algorithm proposed by Inenaga et al. is very similar to Ukkonen's algorithm and draws its main characteristics and ideas from there.

The CDAWG structure is a more space-efficient structure than the DAWG[26], and represents the smallest automaton possible in terms of states and edges[27] – in this case for all suffixes $Suf(\mathcal{B})$ of the input text base $\mathcal{B}$.

For an overview of the properties of the CDAWG structure please refer back to the tables found in the section "Properties".

In this section, I will present the key ideas and modifications of the Inenaga et al. algorithm ([16]) to Ukkonen's Suffix Tree algorithm and give a walk-through of the Inenaga CDAWG construction algorithm's basic ideas. We will use this type of CDAWG construction to serve our final goal, building a *symmetric* index structure for a set of *texts*. For a more detailed and the technical elaboration of the CDAWG construction algorithm, please consult the original paper [16].

[25] compare Inenaga et al. [16], pp. 31
[26] compare Inenaga et al. [16], p. 2
[27] compare Crochemore [5]

The key ideas to this algorithm are

- all edges of an input word lead to the same sink state
- *merging* equivalent factors in the construction by building *equivalence classes* (compare [3])
- to this aim, during construction, edges might need *redirection*
- at times, edges might need *separation*

What Inenaga et al. draw from Ukkonen are the general ideas of the construction process, i.e. *a)* building the structure letter by letter, left to right, directly, and on-line. As was shown before, this makes it possible for the structure to represent at all times the current stage of adding factors. Along with it, *b)* the idea of open edges and pointers into the one place only where the whole string is stored ("data"), *c)* the idea of suffix links and, finally, *d)* the active point are also kept as base for the algorithm. Conversely however, *one and only one* sink state per document will be added to the structure.

5.2.1 Construction example

In this section I would like to present a casual, lightweight walk-through of the Inenaga algorithm. Please consult [16] for deeper technical details and the formal description of the algorithm. The description here is intended to present a nice entry point for readers coming to the subject, showing how to arrive at the structure as shown in figure 17 on page 38.

The example will, as did the example for Ukkonen's algorithm in the previous chapter, add the word *cocoa* to the structure. Having the word *cocoa* as input is a very suitable example for it is short enough to walk through the entire example and it additionally has the nice property of having the repeated substring *co*. This example word will show us again how the adding of particular suffixes can be postponed and how the active point is moving during the construction phase.

Step 1 Adding the first letter *c* of the input *cocoa*.

As was already mentioned, the CDAWG structure has exactly one *sink state* per input string. Since we are adding the first letter of our string, we have to construct this *sink state* first and subsequently *direct all edges of the current input document to it*.

In a first step, we create an open edge (as with Ukkonen) from the *start state* to this newly formed *sink state*, and obtain the structure as depicted by figure 18 on the next page (as with Ukkonen, the additional "bottom state" is left out for presentiveness purposes, as well as the special delimiters # and $).

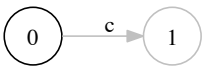

Figure 18: CDAWG construction step for $L = \{c(ocoa)\}$

Step 2 Adding the second letter o (and thus the suffixes up to the current point, co and o)

In the second step, the suffix o, and by this, the suffixes of the current point co and o are added to the structure. Now, since we have one sink state only, the newly created edge is directed to the sink state for this string, i.e $(root, o, sink)$ – compare with figure 19.

As in Ukkonen's algorithm, the open edge c is extended to co, i.e. by setting the open edge marker $\# = 1$ to $\# := 2$. Again, as before, extending all existing edges at once by only one operation can guarantee linear time construction.

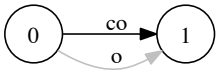

Figure 19: CDAWG construction step for $L = \{co(coa), o(coa)\}$

Step 3 Adding the third letter c and moving the active point.

In step 3, while adding the third letter c in order to add the factors coc, oc, c, another phenomenon occurs that is already familiar from Ukkonen's algorithm: since a transition with c already exists from the root node – which we have returned to in the previous steps without explicit mention – instead of creating another node beginning with same letter, the *active point* is moved.

It is worth noting that the active point represents the current *LRS*, the *longest repeated suffix* of the structure. In this case here, the longest repeated suffix up to the current point is c. The active node is moved to the (implicit) position $(root, c, 1)$, i.e. the position just past the first letter on the edge that starts with letter c. Figure 20 on the next page gives an illustration.

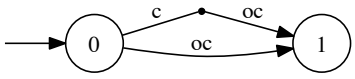

Figure 20: CDAWG construction step for $L = \{coc(oa), oc(oa)\}$ with the factor $c(oa)$ waiting for insertion

Step 4 Adding the fourth letter o and moving the active point again.

This step is similar to step 3 and also known from the description of Ukkonen's algorithm. Again, as in every step, the open edges are automatically extended, and, since there already exists a transition with the current letter o from the current active point where the next insertion would be made from, we move this active point $(root, c, 1)$ to $(root, c, 2)$ and increase counter $remainder := 2$.

Note that we obtain a new longest repeated suffix (LRS) co by doing so – figure 21 gives an illustration.

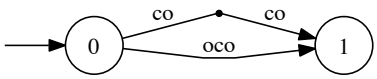

Figure 21: CDAWG construction step for $L = \{coco(a), oco(a)\}$ – factors up to the current point, $co(a)$, $o(a)$ postponed

Step 5 Adding the final letter a of string cocoa.

This final step is an interesting step and will have a few sub-steps.

To begin with, as in every step, all existing open edges are extended to represent the current stage of adding suffixes (compare figure 22 on the next page). The active point is on $(root, c, 2)$ and the counter $remainder$ is currently set to 3, for the currently remaining suffixes are coa, oa and a.

Next, it is checked, whether there is a transition with a from the current point, the *active point* $(root, c, 2)$. Since there is none, a transition with that letter needs to be created – remember, in this algorithm, it will be directed to the dedicated sink state for the current document. As with Ukkonen, since the current point is an implicit state, it needs to be made explicit (or, seen from another perspective the current implicit state *is made explicit* by raising its *out-degree* > 1); consult figure 23 on the following page.

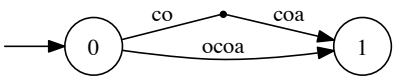

Figure 22: CDAWG construction step 1 for $L = \{cocoa, ocoa\}$. Suffixes *coa*, *oa*, and *a* left for insertion

With this step accomplished, we have made the remaining suffix *coa* part of the automaton. The counter *remainder* gets decremented to $remainder := 2$, the suffixes remaining to be inserted are *oa* and *a*.

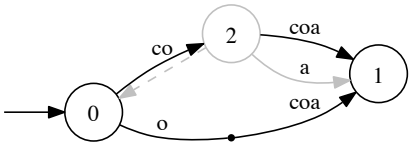

Figure 23: CDAWG construction step 2 for $L = \{cocoa, ocoa, coa\}$. Suffixes *oa*, *a* left for insertion

As for the suffix link, remember that with Ukkonen, it was said that no *suffix link* needs to be set to another state on the first `split`. However, in this case here we need to set this *suffix link* (why this is the case will be clear in a minute).

Additionally, since this was the first `split` in this step, the new current *active point* is set back to *root* for now. From there, since *oa* is the next suffix waiting to be inserted, the *active point* is moved to $(root, o, 1)$ (also represented in figure 23).

As is already known from Ukkonen's algorithm, the *implicit state* where the next insertion will have to happen from (indicated by the *active point*), would be `split` and a new transition with letter *a* of the suffix *oa* would be created from this newly created state *3* – and directed to the *sink state* for this word.

However, in this algorithm this case is handled differently:

Since the resulting automaton is supposed to be the smallest possible ("minimized") automaton, built in a direct-construction algorithm, the state that would be created (state *3*) would be equal to the already existing state *2* in its right-language. Thus it is *not created at all* but rather the edge $(root, o, 1)$ is *redirected* to state *2*.

Essentially, one can imagine the creation of a state *3* with the out-going edges *coa* and *a*, followed by an immediate merge with the already existing state *2*, but in fact,

no new state needs creation.

Still, since this was the second `split` – speaking with Ukkonen – a *suffix link* needs to be set. However, it cannot be set to the state where the previous `split` happened: the previous `split` had created state *2* – exactly the state which the last edge had been *redirected* to. Thus we can use the suffix link that was set before from state *2* to *root*.

Thus the active point is set back to the root state and a transition with a is created from the *root state* to the dedicated sink state for this "document" and the construction is finished for all suffixes of the word *cocoa* (compare figure 24).

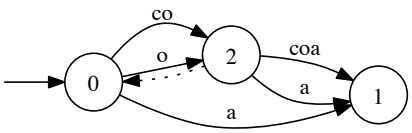

Figure 24: CDAWG construction step 4 for $L = \{cocoa, ocoa, coa, oa, a\}$

Additional Step 6 The above walkthrough made most of the construction clear. However, what has not been covered yet is another main addition to Ukkonen's algorithm: the separation of nodes.

Imagine we were adding the second word (and all its suffixes of course), *cola*. As was shown in the example of equivalence classes in the Definitions section, and as the automaton in figure 3 on page 12 clearly shows, at some point, the suffix a will be represented in an equivalence class of its own – state *4* in the example automaton.

Recalling the definition of equivalence classes, it is clear that the suffix a will need to be in an equivalence class of its own, since it occurs in two different places: as the last letter of *cocoa* and as the last letter of *cola*. Thus, its set of *end-positions* is not the same as the (individual) sets of *end-positions* of the same letter in the respective words.

What this means in terms of the construction is, that, for the second word *cola*, the construction will be carried out similarly to the above example: as the second document being added to the structure, it will be granted a dedicated sink state and all edges will be directed towards it. The active node will move in situations where transitions with the current letter to be added already exist. Finally however, for the last letter a, a call to the function `separate_node` (as called by Inenaga et al.) will happen and state *4* will be created for the new equivalence class that a is part of.

The points to keep in mind are:

- The close similarities to Ukkonen's algorithm: using pointers into data and extension of the open edges using the current point boundary #, the moving of the active node, as well as the order in which the suffixes get added to the structure.
- The "one dedicated sink state" approach and how the edges of one document get directed to it
- The use of equivalence classes and how these are represented as states
- The merging and separation of nodes

6 Constructing the symmetric automaton

The SCDAWG index structures presented in this work basically build on the classic Finite State Automata structures and theory as described in chapter "Index Structures".

6.1 Definition of the SCDAWG

Instead of being a five-tuple, like the CDAWG, an SCDAWG structure is a *six-tuple composed of two generalized CDAWG automata*, with an additional delta function $\delta_{\overrightarrow{C}}$ for extensions "to the left".

> The bidirectional *symmetric compact directed acyclic word graph* (SCDAWG) for $\#D\$$ is the tuple $\overleftrightarrow{C}(\#D\$) := (Q_{\overleftrightarrow{C}}, \Sigma^{+}_{\#\$}, \overleftrightarrow{[\epsilon]}, \delta_{\overleftarrow{C}}, \delta_{\overrightarrow{C}}, F_{\overleftrightarrow{C}})$ where $Q_{\overleftrightarrow{C}} := Q_{\overleftarrow{C}} = Q_{\overrightarrow{C}}$ and $F_{\overleftrightarrow{C}} := F_{\overleftarrow{C}} = F_{\overrightarrow{C}}$
> [10, Definition 5.10, p. 16]

Regarding the definition, two details should be noted:

- In our case, the automaton will hold the text base $\#\mathcal{B}\$$ instead of the dictionary $\#\mathcal{D}\$$. As such, we will add a seventh element to the tuple definition of the SCADWG to form a *document-indexing* SCDAWG, the *location-relation* $P : Q_{\overleftrightarrow{C}} \rightarrow pos_{\#B\$}(q)$ which maps an equivalence class' occurrences to the positions in the text base B. (See "Adding "inverted file" information to the SCDAWG" for a description of how this information is generated.)
- Again, the special markers # and \$ denote that each of the elements of $\mathcal{B}$ is prepended (resp. appended) with # (resp. \$) that are not part of the input alphabet Σ.

The two composing CDAWG structures are two generalized CDAWG automata as defined in "Definition of the CDAWG".

6.2 SCDAWG structure overview

With the definitions given, the structure of the SCDAWG can be pictured like the following (compare figure 25 on the following page).

6.3 SCDAWG Example

Figure 26 on page 48 gives an example of an SCDAWG structure for the language $L = \{Inf(cocoa), Inf(cola)\}$ with additional boundary markers # and \$. The dashed transitions represent the additional delta function $\delta_{\overrightarrow{C}}$ for extensions "to the left".

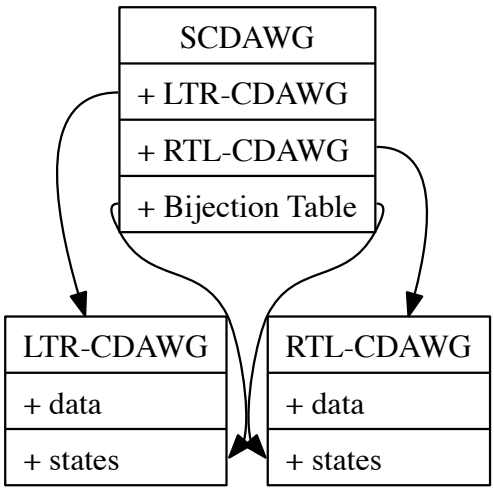

Figure 25: SCDAWG structure overview

6.4 Building the SCDAWG

Since the two generalized CDAWG structures which form the SCDAWG are isomorphic[28], we need a recursive bijection function that will run in linear time and identify the equivalent states in the LTR-CDAWG and the RTL-CDAWG. The isomorphism of the two underlying structures is guaranteed through the properties of the equivalence relation with an equivalence relation being *reflexive*, *transitive* and *symmetric* and thus the states of the two independent structures can be identified as identical and the structures themselves synchronized.

If we were not to build to CDAWGs independently and afterwards identify the equivalence classes / states, and instead built the SCDAWG directly, in the worst case the construction time could be quadratic w.r.t. to the input length n.[29]

6.5 Algorithm for the bijection

Synchronization of the two independent, but nonetheless isomorphic CDAWG structures works by using a bijection function b. The input words are added to the CDAWGs individually (in LTR and RTL direction). Following this step, every new state is identified with its "counterpart" in the other structure by the recursive *FindRightState* function.

[28] compare Gerdjikov et al. [10], p. 17, Proposition 5.13
[29] again, Gerdjikov et al. [10], p. 13, Proposition 5.13

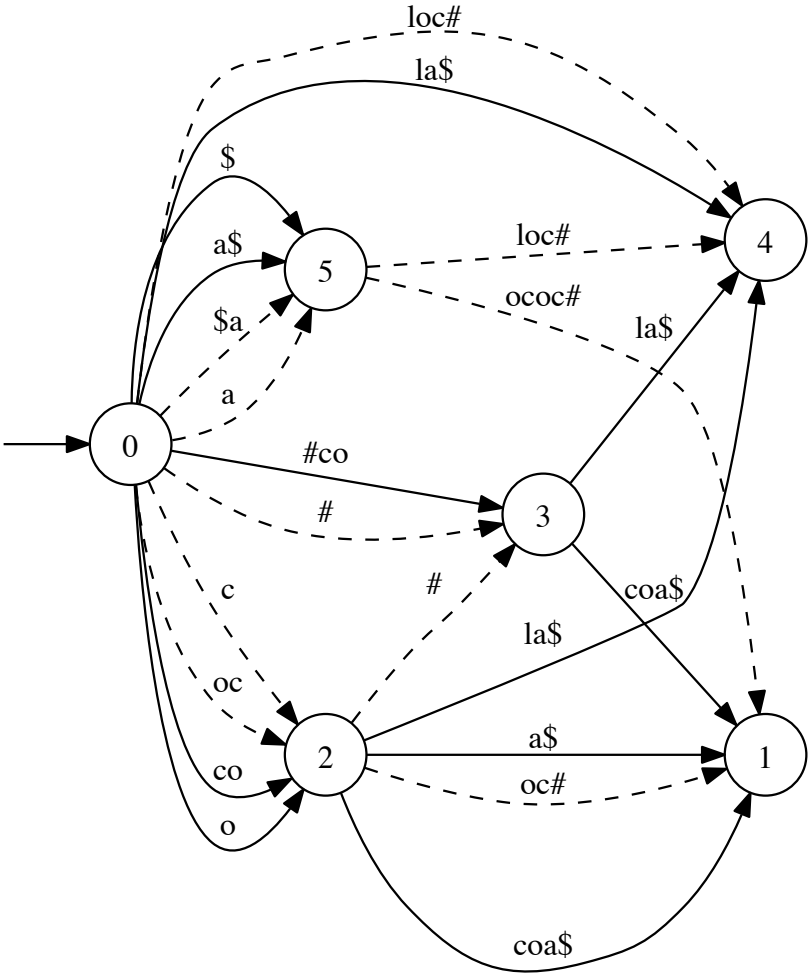

Figure 26: SCDAWG for $L_{\#\$} = \{Inf(cocoa), Inf(cola)\}$

6.5.1 Intuitive description of the algorithm

The algorithm for the bijection works along the following lines:

First, the independent CDAWG structures (created and initialized in lines 2–3 of function *BuildSCDAWG* in the pseudo code 27 on the following page) are filled with the input strings (in forward and reverse order – compare lines 6–7 in the pseudo code). By this, they create new states (which are consecutive integer numbers, with the root state being *state 0*). For each new state, an empty slot in the bijection table b is created (line 10).

Next, the recursive identification of states is carried out using the recursive *FindRightState* function. Since the adding of new states is stable w.r.t. to the previously calculated values of b, only the newly added states need identification[30].

The recursive function first descends back to a known identification of two states using the *suffix links* (the bottom of the recursive function is guaranteed through the identification of the root states of both automata in line 12 of *BuildSCDAWG*).

From there, the information used to identify two states is the position $end(i)$ in the concatenated `data` variable and the *length* of the *canonical representative* of this equivalence class (i.e. of this state) – compare line 6 of the *FindRightState* function in the pseudo code. Once the begin of the canonical representative of this equivalence class in `data` is known ($end - length = j$), there must be exactly one transition in the other respective structure with this letter σ at position j in `data` – which leads to the state that needs identification in the other structure; see line 7 of *FindRightState*.

For the proof and mathematically tighter description of the algorithm, please consult [10, p. 18, Proposition 5.15].

6.5.2 Pseudo-Code

The pseudo code to the intuitive description is given in figure 27 on the next page[31].

6.5.3 Actual implementation

Keep in mind that for the bijection to work, since it builds upon the chain of suffix links in the tree (which are located in the `BuildHelp` structure), the identification step must be carried in `PRE-CLOSED` stage (compare section "Implementation specifics: Staging system" in the sequel).

[30] compare Gerdjikov et al. [10], p. 18, Proposition 5.15
[31] (almost) verbatim from [10, p. 20]

```
1   BuildSCDAWG(#D$){
2       Q = {0}; δ = 0; F = 0; Q' = {0}; δ' = 0; F' = 0;
3       ←C = (Q, Σ_{♯$}, 0, δ, F); ←C' = (Q', Σ_{♯$}, 0, δ', F'); b = 0; b^{-1} = 0;
4       for(#W$ ∈ #D$){
5           n = |Q|;
6           AddStringInCDAWG(←C, #W$);
7           AddStringInCDAWG(←C', $W^{rev}#);
8           i = n;
9           while(i < |Q|){
10              b(i) = nil; i = i + 1;
11          }
12          b(0) = 0; b^{-1}(0) = 0; i = n;
13          while(i < |Q|){
14              if(b(i) == nil)
15                  b(i) = FindRightState(←C, ←C', b, b^{-1}, i);
16                  b^{-1}(b(i)) = i;
17              }
18              i = i + 1;
19          }
20      }
21      return(←C, ←C', b, b^{-1});
22  }

1   FindRightState(←C, ←C', b, b^{-1}, i){
2       s = suffixlink(i);
3       if(b(s) == nil){
4           b(s) = FindRightState(←C, ←C', b, b^{-1}, i); b^{-1}(b(s)) = s;
5       }
6       q' = b(s); j = end(i) − length(s); σ = D_j;
7       let δ'(q', σU) = p' be the σ-transition from q' in ←C';
8       return p';
9   }
```

Figure 27: Online construction of a representation of SCDAWG for $\#\mathcal{D}\$$

7 Adding "inverted file" information to the SCDAWG

To finally obtain a *document-indexing* SCDAWG structure, one last element is missing: adding positional information to the equivalence classes – i.e. the states of the automaton.

Following the previously described bijection and identification step of the isomorphic states in the LTR- and RTL-CDAWGs, this chapter will describe Blumer, Blumer et al.'s idea of attaching inverted file information to CDAWGs, as described in [2].

Just as the previous step, this step again runs in linear $\mathcal{O}(n)$ time. With this final step running in linear time again, and thus *all* steps running in linear time, an overall complexity of $\mathcal{O}(n)$ can be guaranteed.

Please note again, that as such, the construction described here and in [10] is the first algorithm to build a *symmetric* structure with document-indexing / inverted file information that can be built directly and with linear complexity in space and time.

The chapter given here will describe how to apply Blumer, Blumer et al.'s algorithm (given in [2]) for CDAWGs to our structure at hand, the SCDAWG.

Recall, that the states of both underlying CDAWGs can be identified since *the equivalence relation* $\overleftrightarrow{\equiv}$ *is the transitive closure of* $\overrightarrow{\equiv}$ *and* $\overleftarrow{\equiv}$ and for this reason the equivalence classes of both these automata are *identical*, since the equivalence closure $\overleftarrow{\equiv}$ is *symmetric, reflexive, and transitive*[32].

Now, in order to be able to locate patterns in the text base $\#\mathcal{B}\$$, attaching location information of the occurrences of the equivalence classes $Q_{\overleftrightarrow{\mathcal{C}}}$ w.r.t. to the positions in $\mathcal{B}_{\#\$}$ is needed. This way, each state will hold information of its occurrences in the text base $\mathcal{B}$. To this end, this final step following the bijection algorithm closes the document indexing process.

7.1 Adapting Blumer, Blumer et al.'s idea to SCDAWGs

After having given all the needed steps in the previous sections, this last step, adapting Blumer, Blumer et al.'s algorithm based on CDAWGs to SCDAWGs can be described as a rather straightforward approach.

Recall the composition of the SCDAWG: the SCDAWG is based on two independent – but nevertheless isomorphic – CDAWGs and the additional identification step that followed.[33] Through the isomorphism of the two underlying CDAWG structures, it is perfectly enough to apply the algorithm given by Blumer, Blumer et al. to *one of*

[32] compare [2], [16, p. 13] and also [10]

[33] cf. "Constructing the symmetric automaton" and figure 25 on page 47.

the underlying CDAWGs only and thus attaching of inverted file information to the states of only one of the CDAWGs, preferably the LTR-CDAWG.

7.2 Attaching inverted file information to SCDAWG's states

Since Blumer, Blumer et al.'s algorithm will make use of suffix links, again, as before, this step is to be carried out in `PRE-CLOSED` stage[34].

We will gain from it, a new method called `Index()` in a (templated) class that we will write for it, `DocumentIndexingAutomaton<SCDAWG>`[35].

In the following, a hands-on-description of the algorithm is given. Please consult [2] for the full, in-depth description.

7.3 The algorithm

The document indexing algorithm has two sides to it: Obviously, in a first step, *a)* the *indexing step*, the structure needs to be prepared accordingly to hold the information bits and pieces that will be needed in the second step, *b)* the *retrieval step*.

The first step, taking care of the indexing, consists of a modified `AddDocuments()` method and the `Index()` method. Since this is such a crucial step, it also finds a representation as a dedicated "stage" of the automaton: `INDEXED` stage.[36]

The retrieval step, is represented by the method `findall()`. It will locate all occurrences of the current pattern in the indexed text files $t \in \mathcal{B}$. `findall()` will (internally) call two other methods: `find()` and `findRec()`.

7.3.1 Overview of the algorithm steps

Figure 28 on the next page illustrates the indexing steps.

For the retrieval side, figure 29 on page 54 summarizes the steps involved.

7.3.2 Intuitive description of the algorithm

As far as the indexing side of the algorithm is concerned one will:

- Add all documents, each as one long string, append the string to `data` with delimiters # and $
- grant each document its own, dedicated sinkstate
- save information document $\leftrightarrow$ sinkstates

[34] again, compare section "Implementation specifics: Staging system" in the sequel
[35] compare "Implementation specifics: the templated DocumentIndexingAutomaton" in the sequel
[36] compare "Implementation specifics: Staging system"

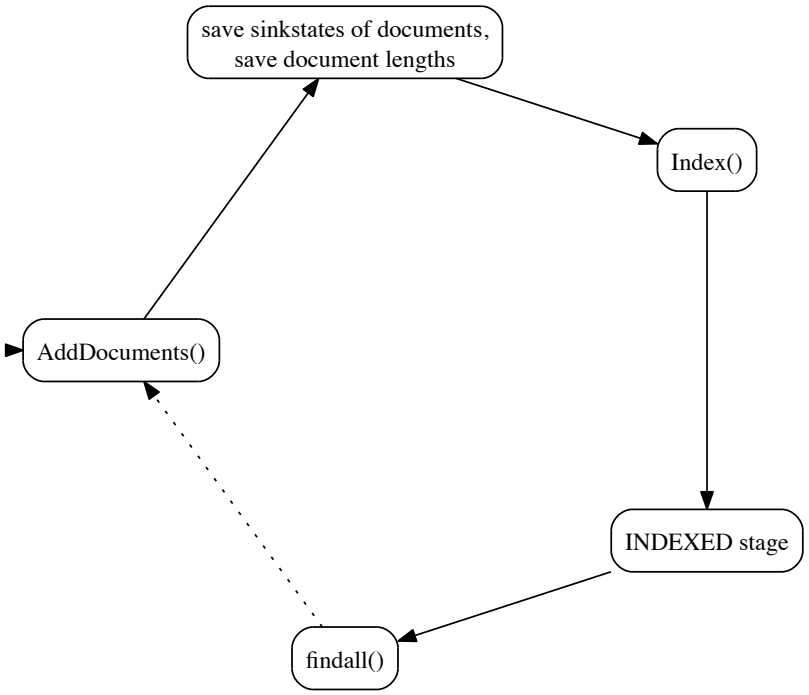

Figure 28: Graphical overview of document indexing steps

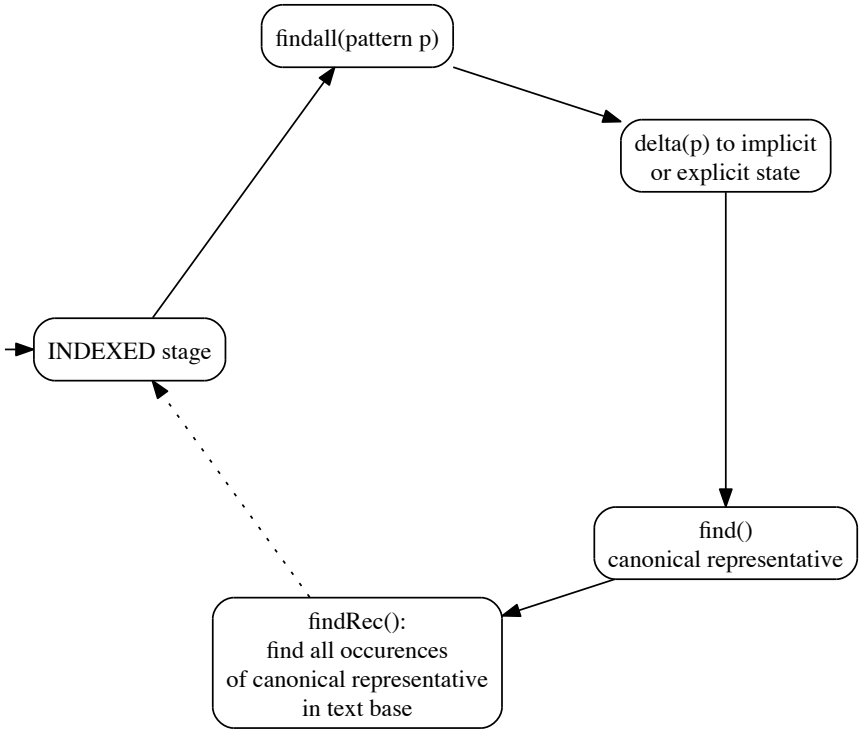

Figure 29: Graphical overview of pattern retrieval steps

- save information document $\leftrightarrow$ documentlength
- in a final step, for each document that was added:
  - starting from the document's dedicated sink state
  - walk up the tree on the suffix links until the start state is reached
  - tell every state visited on this path, that it is attached to the current document

For the location procedure and pattern p,

- delta with pattern p to a state s
- state s may be an implicit one
  - in that case search for the canonical representative of the current pattern p and adjust the length information of the modified pattern accordingly
- once at an explicit state, use the length information of the pattern
- use this length information of the pattern and documentlength to calculate all the positions in each of the documents that are attached to that state to calculate the positions of occurrences of the pattern in the documents.

7.3.3 Actual implementation of adding of documents

The code examples given here are based on the templated C++-implementation and present a version that was simplified to serve as a kind of pseudo-code implementation to grasp the essential bits of the algorithm.

In the `DocumentIndexingAutomaton`, the method `AddDocument()` is similar to the previously existing methods to add documents to the structures. However, in this case, it additionally keeps track of the association sinkstate $\leftrightarrow$ document and the information of the document's length.

The class template `DocumentIndexingAutomaton<AutomatonType>` introduces `AddDocuments()` as a public member method and consequently will call `AutomatonType::Add()`.

This means, if it is instantiated as `DocumentIndexingAutomaton<CDAWG>`, it will call `CDAWG::Add()` etc.

`AddDocuments()` takes care of adding the documents, saving the corresponding sinkstates and document lengths and sets the automaton back to `UNSORTED` stage in the staging system (more on the idea of the staging system in the sequel).

```
AddDocuments( const std::vector<DocumentName>& documents ) {

    /* Suffix links are needed */
    assert( ManagedAutomatonStage_ < CLOSED );

    unsigned int total_length = 0;
    unsigned int number_of_docs_read = 0;

    /* Save the document names in a member variable */
    document_names_ = documents;

    // Add documents
    for (auto& doc: document_names_) {
        number_of_docs_read++;

        // read the whole document
        std::string str = read_document(doc);

        // call to template's Add()
        // (i.e. CDAWG::Add or SCDAWG::Add)
        // Add() will return the sinkstate
        // which was created for this document
        State sinkstate = AutomatonType::Add( str );

        // We need the sinkstates later on
        // to follow the suffix links back up to source.
        // In order to associate states and documents
        // that relate to each other
        document_sinkstate_[doc] = sinkstate;

        // adjust document's length
        // (it is prepended and appended with
        //  '#' and '$')
        DocumentPosition pos = str.length()
                             + total_length
                             + number_of_docs_read*2
                             - 1;

        document_length_[doc] = pos;
```

```
        // - 2 to take '#' and '$' into account...
        total_length += str.length() - 2;
    }

    /* Automaton has newly added documents
       So it will need to be sorted and
       indexed
    */
    ManagedAutomatonStage_ = UNSORTED;
}
```

7.3.4 Actual implementation of indexing step

Of course, the `Index()`-ing step will have to happen before any `findall()` steps and will set the automaton to `INDEXED` stage.

The `Index()` method expects the automaton to at least be filled, and not yet closed (since the suffix links are needed for the indexing step and would not be available in `CLOSED` stage).

For the case that the automaton has not been sorted yet, the `AutomatonType`'s respective `SortTransitions()` method is called.

Essentially, what the `Index()` method does, is to follow the chain of suffix links back up to the source state for each document, starting from the document's sink state. Every state that is "visited" on this path is attached with the information that it is associated with this particular document.

Finally, the automaton's stage is set to `INDEXED` stage.

```
Index() {

    assert( ManagedAutomatonStage_ > EMPTY &&
            ManagedAutomatonStage_ < CLOSED );

    if ( ManagedAutomatonStage_ < SORTED ) {
        AutomatonType::SortTransitions();
    }

    assert( ManagedAutomatonStage_ >= SORTED );

    if ( ManagedAutomatonStage_ < INDEXED ) {
        State source = 0;
```

```
        State state  = 0;

        for (auto doc: document_names_) {
            state = document_sinkstate_[doc];

            while(state != source){
                // associate document back to state...
                states_documents_[state].push_back(doc);

                // ...and follow suffix link up to source
                state = suffixLink(state);
            }
        }
    }
    ManagedAutomatonStage_ = INDEXED;
}
```

On the retrieval side, the method `findall()` is the the only public method visible to the user. As its arguments, it accepts a string and starts the search process. It will call `find()` and `findRec()` and return a container holding the positions where the pattern p occurs in the text base $\mathcal{B}$.

7.3.5 Locating patterns and positions in the SCDAWG

```
// find all occurrences of a given string W
findall( const std::string& w ) {

    // Transitions need to be sorted (internally)
    if ( ManagedAutomatonStage_ < SORTED ) {
        AutomatonType::SortTransitions();
    }

    // Indexing step must have been executed before
    if ( ManagedAutomatonStage_ < INDEXED ) {
        AutomatonType::Index();
    }


    State state = GetStartState();
    State nextState;
```

```
    // Delta() with each character of the pattern
    for( size_t i = 0; i < w.length(); i++ ){
        nextState = Delta( &state, w.at(i) );
        if( nextState.index == NO ){
            /* no occurences, break */
            return; // empty results
        }
        state = nextState;
    }

    /* call find() */
    return find( state, w.length() );
}
```

The method `find()` in the retrieval chain is a private method, not visible to the user. It takes care of adjusting the state and length information if the state reached with the pattern is an implicit state. In the latter case, the next explicit state will be searched for and the "length" of the pattern is adjusted. (Recall that the next explicit state is, in other words, the canonical representative of the current equivalence class that the implicit state is a part of. The canonical representative is, by definition, longer than the actual pattern and thus the length needs to be adjusted).

`find()` will call `findRec()` which will then collect the concrete positions of the pattern p in $\mathcal{B}$.

```
// the find method
find( const State state, unsigned int length ) {
    // check excplicit-ness of the state:
    // if state is not explicit yet, "move forward"
    // (i.e. find the canonical representative)
    // and adjust length
    if( ! state.explicit ) {
        length += state.implicit.nextExplicitState.endPoint
                - state.implicit.ownRealStartPoint + 1;
        state = state.implicit.nextExplicitState;
    }
    return findRec(state, length);
}
```

7.3.6 Collecting the positions in the text base

`findRec()` is the last callee in the chain `findall()` – `find()` – `findRec()`. It recursively collects all occurrences of the pattern across the text base $\mathcal{B}$ and returns a container holding the respective positions.

The recursive `findRec()` method works along the following lines:

(1) if the current state `s` is associated with a document, it will insert the pattern's position `pos` in the document(s) into the container that will be returned back up along the caller-chain

(2) in most cases however, the loop over the transitions will be used to recursively call `findRec()` on the states that the original state transitions to until a state of category (1), i.e. a state that has been "visited" by `Index()` is found. Of course, the state that was reached by `findall()` and passed in to `findRec()` by "walking" with the pattern p is part of some document (otherwise `findall()` would not have reached any state and would have returned zero occurrences) – nevertheless this does not necessarily mean that the state that was reached by `delta()`-ing to it had been visited before by `Index()` on the chain of suffix links.

Along the way, obviously, the length information needs adjusting. In other words, the method reaches out from the current state in the tree until it reaches a state that had been visited and associated with documents in the `Index()` method.

From the state that is reached by recursively stretching out into the deeper tree structure, `findRec()` can, using the document's length information and the length of the pattern, determine the position(s) of the pattern in the document(s).

Again, this is a simplified version of the method and special precautions need to be taken to get the *real* position. Recall the delimiters # and $ and keep in mind that all documents are concatenated to one big `data`-string.

In fact, three different positions can be distinguished:

i) the relative position in the respective document,

ii) the absolute position in the concatenation of documents in `data`, and finally,

iii) the actual position in `data` taking into account the delimiting symbols on each document in the `data`-concatenation.

For the understanding of the algorithm however, the method shown here is sufficient.

```
// the recursive collector method
findRec( const State s, const unsigned int length ) {

    // Positions collector to be returned
    static DocumentIndexingAutomatonFindResults results_acc;

    // state s had been visited by Index():
    // in many cases, if there is no association,
    // this loop will not start and we have to stretch
    // out into the tree using the next loop below
    for (auto doc: states_documents_[s]) {
        DocumentPosition pos = document_length_[doc] - length;

        results_acc.insert_position(doc, pos);
    }

    // stretch out into the tree until we hit
    // states that had been visited and thus
    // associated with a document by Index()
    for( auto transition: states_transitions_[s] ) {
        findRec(
          transition.toState,
          length + transition.toState.endPoint
                 - transition.startPoint
                 + 1
        );
    }

    // return collected positions up along the chain:
    // findall() -- find() -- findRec()
    return
      std::forward<DocumentIndexingAutomatonFindResults>(
        results_acc
    );
}
```

7.4 Overall Time and space complexity

As was noted in the beginning of this chapter, the complete construction of the `DocumentIndexingAutomaton` is carried out in linear time.

Since each of the algorithms described so far runs in linear time, the construction of the document indexing structure as a whole will run in linear time and use linear space only w.r.t. to the input length n of $\|B\|$.

Namely, the linear-complexity of the algorithms is proven individually for:

- The construction of the CDAWG: [16]
- the construction of the SCDAWG structure (identification of equivalent states): [10]
- attaching the inverted file information: [2]
- linear size of the CDAWG automaton: [3]

8 Code Usage Example

In this chapter, a quick introduction is given to the usage of the existing code base.

8.1 C Automata Code Base Usage Example

The C Automata Code Base has one executable: `caut`. This target was built by the original library and is still built after introducing `cmake` to the compilation workflow[37].

Usage of this executable is along the following lines:

The `caut`-executable is the entry point to a list of features and functionality provided by the original library[38]. It accepts several different commands as its first argument (similar to tools like `svn` or `git`), or, for the case that no arguments are given, goes into interactive mode, whereby the the information that would be supplied in form of arguments is stepped through. The executable's first-hand commands are:

1. `reverse_and_sort`
2. `build_suffix_tree`
3. `generate_suffixes`
4. `compressed_automaton_lang`
5. `build_cdawg`
6. `compressed_automaton_stat`
7. `build_scdawg`
8. `scdawg_left_lang`
9. `compressed_automaton_gv`
10. `scdawg_stat`
11. `scdawg_gv`

For a description of the individual arguments, please consult the interactive mode. Although most commands should be self-explanatory, as a rule of thumb, all commands ending in `_gv` are used to dump graphical visualizations using the "`graphviz dot`" tool while all commands beginning in `build_` are used to build the respective structures and serialize them to disk in binary format.

[37] cf. "Transition to cmake build system"
[38] cf. "Structure of this project"

These first hand commands are followed by additional arguments and again, should be self-explanatory:

- `file`
- `PLAIN / UTF-8`
- `TarjanTable yes/no`

8.2 `msgrep` implementation example

In this section, an example is given how a tool similar to `fgrep`, named `msgrep` can be implemented for the task of locating patterns in text bases. Compare, e.g. Aho-Corasick ([1]) and other implementations of the Unix-classic `grep` program.

`fgrep` is used as a pattern search tool. This means, that it is given a set of texts and a pattern of which to locate all occurrences in the given set of texts (may the text base be served as files or piped input on the terminal). On successful locating of the pattern, its location information, i.e. the *offsets* where the pattern occurs in the files are presented.

The `DocumentIndexingAutomaton<AutomatonType>` can be used to implement a similar feature.

It is worth knowing that the Aho-Corasick ([1]) `fgrep` program works in a similar fashion as our program will be: The given texts are represented in a suitable index structure with additional "inverted file" information to execute pattern searches on the text base.

`fgrep` is a variant of the classic `grep` tool, alongside several other variations of the `grep` tool like `egrep` and `agrep`, where `agrep` is based on the algorithm in [@WuManber92] and supports approximative matching. `fgrep` is used for exact pattern search that builds a trie like structure with additional suffix links as "failure transitions", as was seen before.

In our case, the `msgrep` tool will be based on the SCDAWG symmetric index structure and will work in the same way, that the usual `grep`-derivates work: transforming the text base into an index structure and locate the pattern p to search for while adding additional inverted file information.

Here is an example with additional comments of how the presented code base can be used to implement `msgrep` with different underlying types of automata (CDAWG, SCDAWG, SuffixTree, …) using the templated class `DocumentIndexingAutomaton`. See "Implementation specifics: the templated DocumentIndexingAutomaton" for an in-depth description of the `DocumentIndexingAutomaton` and the ideas behind it.

```cpp
#include <iostream>
#include <vector>
#include <string>

#include "DocumentIndexingAutomaton.hpp"
#include "SCDAWGAdapter.hpp"

/* Get into namespace */
using std::vector;
using std::string;
using std::cout;
using std::endl; /* is "\n" */

using lmu::cis::sis::DocumentIndexingAutomaton;
using lmu::cis::sis::SCDAWGAdapter;

int main() {
  vector<string> filenames = { "file1", "file2" };

  /* change to CDAWGAdapter, SuffixTree if needed */
  DocumentIndexingAutomaton< SCDAWGAdapter > aut;

  aut.AddDocuments(filenames);

  aut.Index();

  /* Search for all occurrences of the pattern "cocoa" */
  string pattern("cocoa");

  auto results = aut.findall(pattern);

  /* List matches */
  for (auto r: results) {
    cout << "Pattern " << pattern << " found in "
      << results.document_name(r)
      << " at "
      << results.relative_pos(r)
      << endl;
  }

```

```
41    /* Concordance view,
42       with 30 chars to the left and to the right
43    */
44    for (auto r: results) {
45      cout << results.lr_context(r, 30) << endl;
46    }
47
48    return 0;
49  }
```

For the correctness of the argument, this "`msgrep`-implementation" does not support search using *regular expressions* as the original `grep`. Nonetheless, it is similar (and, by its symmetricity, more powerful) than `fgrep`. Compared to `agrep`, this structure can be used as an excellent ground for approximate search without the so called "wall effect", that, in many cases, makes fuzzy matching slow (or even unreliable as far as the results are concerned).

For a description of what the "wall effect" means and the details of the algorithm circumventing it using a SCDAWG index structure, please consult [10]. For information on how to provide this functionality to the current code base, please see the description of possible areas of improvement in the closing section of this work, the descriptions on the implementation specifics in the next chapter, and, make sure to find the technical documentation linked to in the final chapters.

8.2.1 Sample output

To be able to see the concordance view in action, here is some sample output of a part of philosopher Friedrich Nietzsche's works (for now, the sample files have been "brought into shape" very roughly). This data was used for the "heavy load tests" in the Metrics section, cf. "Automaton Size metrics" for numbers and figures on the automaton's size, execution times, etc.

```
Pattern und found in ../etc/N/MAI_clean.txt -- 668 -- 668 -- 669
Pattern und found in ../etc/N/MAI_clean.txt -- 895 -- 895 -- 896
Pattern und found in ../etc/N/MAI_clean.txt -- 1237 -- 1237 -- 1238
Pattern und found in ../etc/N/MAI_clean.txt -- 1276 -- 1276 -- 1277
[...]
Concordance view:   gar nur die streit und redelustigen Schaa
Concordance view:  muss geschehen sein und die einzige Statue
Concordance view:  an welcher man Sinn und Zweck jener grosse
Concordance view:  ch sollt ihr mir Freunde geworden sein und
Concordance view:  Brueder, auf jede Stunde, wo euer Geist in
[...]
Matches: 4097
```

9 Adapting the existing automata code base

In order to create a usable library with safety and to establish a solid ground work which to continuously build on in the future, several adaptation steps to the existing code base were needed.

The main goals for this work and including future work on the existing library consisted of establishing maintainability and adaptability for new use cases, such as this one.

On the one hand side, the existing `cautomata`-library composes the essential parts of the algorithms described in the previous chapters, on the other hand was in a state where it was designed for one specific use case and thus needed adaptation to be able to meet different needs that might arise in the future.

Clearly, by lowering the heights of the initial steps of getting to grips on the existing library, collaborative future efforts are leveraged.

9.1 Extending, Reusing and Refactoring of the existing implementation

Since a lot of good work has been put into implementation of the algorithms and automata described, the document indexing automata presented in this work are closely coupled to the already existing implementations by Petar Mitankin[39].

Not only does the existing code base already implement many of the intricate algorithms and index structures described in this paper in a well-thought out and highly efficient manner, moreover does an adaptation process meet the tight time constraints for this work and also reflects a standard situation in commercial work processes.

Simply put, re-implementing a large library and thus "re-inventing the wheel", in most cases, is not affordable. Please consult the section code metrics for estimations on lines of code, estimated time schedules and a rough estimation on money resources, etc. to prove this argument.

Consequently, reusing and maturing the existing code base through refactoring and adaptation was a rational decision in light of the time (and, were it a commercial application, money constraints) given. Especially does this hold true for the proof of concept implementations used for the metrics in the paper [10] that were already in good shape as far as the speed metrics of the implementation are concerned.

In a nutshell, these pre-existing implementations, as a starting point, gave way to the document indexing automata presented in the preceding sections.

[39] Petar Mitankin, researcher at the Bulgarian Academy of Sciences with Stoyan Mihov is the author of the underlying C implementation of most index structures upon which the indexing and suggestions-giving automata of this work were built. He is co-author of [10].

The following sections will reflect on the refactoring and adaptation process, will elaborate on the rationale of design decisions and approaches taken, and, finally, will present highlights of the maturing of the existing code and finally give an outline of the advantages that can be gained gained on a long term basis.

9.1.1 Refactoring guidelines and Approaches

The indexing automata presented here use several techniques and approaches and mainly follow these guidelines, principles and common refactoring practices in the adaptation process:

- Adapting the existing pure C codebase using the *Adapter / Wrapper Design Pattern*, keeping the original implementations untouched (as described in section "Wrapping C")
- Tightly coupled to the previous point is the *Layering System* consisting of a C++-layer covering the underlying (wrapped) C-implementation-layer. Please consult the description of advantages of this approach in section "Implementation Specifics: Layer System Description"
- Along with the advantages that the *object-oriented programming* approach gives us – explicitly stated inheritances and interfaces, and, moreover, inheritability of classes as a ground base for future work, while keeping the sheer speed of the original C implementation through the adapter pattern (compare with section "Adapter Classes") – *generative programming* approaches were used for describing the document indexing automata's behavior. See section "Implementation specifics: the templated DocumentIndexingAutomaton" for the details and more information on the rationale behind this decision and the resulting advantages.
- Less on the concrete implementation, but more on the distribution side, leveraging future work on the existing code base, basic ground work was done to ease implementation of new features for coders coming to the project by adding the basic grounds for *test driven development* – see the sections on "Transition to cmake build system" and "Unit tests" for a description of this approach.
- Additionally, `cmake` was brought into the project to help with the classic GNU-`make`-process in building and compiling the library on different platforms

9.2 Wrapping C

A first step to "conquer" the existing the library was to wrap the most essential C-style structs into full object oriented classes.

The combination of a lower-level C-layer and a higher-level C++-layer brings in a "best of both worlds" notion, in that the lower-level C-layer guarantees maximum efficiency and the higher-level C++-layer maximizes flexibility, safety and fast results and long-time implementation gains in terms of time and cost reduction.

In this section I will highlight the first steps of adapting the existing code base, namely through *Adapter Classes*.

9.2.1 Adapter Classes

The C++-adapters used here are full C++-classes wrapped around their counterpart C-structs – which can be (and often are) interpreted as something "equivalent" to classes with the only difference that "everything is public". This is a short-sighted argument, for C-structs still lack all gains that can be drawn from a full fledged object-oriented class (just to name a few: inheritance, interfaces, encapsulation, etc. – more on that to follow in the sequel).

The so called "Adapter pattern" which was used is also known as the "Wrapper pattern".[40]

As the "Gang of Four" point out, the Adapter Pattern's applicability is given by the following criteria:

> "*Use the Adapter pattern when*
>
> - you want to use an existing class, and its interface does not match the one you need.
> - you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.
> - (object adapter only) you need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class."[41]

As far as "Collaborations" are concerned, [9] have to say the following:

> "Clients call operations on an Adapter instance. In turn, the adapter calls Adaptee operations that carry out the request."[42]

This approach of adapting the underlying C-structs is fruitful for a variety of reasons:

[40] For a detailed description of this Object Oriented Design Pattern, see the "Gang of Four" (also known as the "GoF") in [9, pp. 139].

[41] Gamma et al. [9], pp. 139

[42] Gamma et al. [9], p. 141. The "GoF" has much more to say about Adapter pattern's consequences, also see pp. 142

Advantages of Adapter Classes. Compared to their C-structs, full C++ adapter classes define "proper" constructors, destructors, interfaces, state their inheritance, are inheritable and template-able, meaning that

- the objects are "easier to handle" on the user side: easier *instantiation of objects* with automatic memory allocation and multiple constructors for different situations
- Easy to implement: mainly these full-feature adapter classes created for this work do a simple *delegation* ("pass-through") of their own methods to the struct's original functions, while making them more type-safe
- The C++ adapter classes of this work can be reused in many different situations through inheritance (for instance confer to "DocumentIndexingAutomaton as a template")
- Since the adapter pattern makes it possible to generate compatible interfaces for other classes, in our case here, a crucial advantage can be found in that the existing code base need not be rewritten, but stays as it is. Imagine a vendor class interface with compiled objects, that you don't have direct access to, except for the documented interface. In all these cases the adapter pattern proves to be ideal.

In a nutshell, full-fledged object-oriented classes are

- easier to instantiate / work with
- make inheritance, interfaces and composition explicit
- use the language's own object building facilities (rather than "rolling your own" object system, or, for that matter, use pointers and references *as if they were real objects* and leaving inheritance, templatization and specialization implicit and/or to dangerous macros)
- therefore minimize risks
- while still holding up the speed of the original C implementation.

The adapter pattern used here is inherently very tightly coupled to the LayeredSystem introduced in this work. For a full description, cf. the section on "Implementation Specifics: Layer System Description".

9.2.2 General Description of Adapter Classes

Adapter classes are relatively easy to implement in that the adapting class merely keeps a pointer to the adapted class (also referred to as the "adaptee") and, in general, mainly forwards method calls to the adaptee's functions.

However, adapter classes are capable of much more. Not only can adapter classes specifically *override* some (or all) of the adaptee's behavior, in order to make incompatible interfaces compatible to each other [cf. 9, pp. 139] – adapters can also *add behavior* to the class adapted or specifically choose to *hide* certain functionality.

In our specific case here, not only do the adapter classes wrap the original C-structs, but also give them, by making them full C++-classes, *additional properties*, like inheritability and present the user with a variety of overlaoded methods that make working with these classes easier in different circumstances.

Conventions for adapter classes in this distribution. The adapter classes defined in this distribution follow these conventions:

- The `struct SCDAWG`'s adapter can be found as the `class SCDAWGAdapter`
- for instance, a function like `SCDAWGAdd(SCDAWG* scdawg, VoidSequence* input)` will turn into the method `SCDAWGAdapter::Add(VoidSequence* input)`
  - oftentimes, `SCDAWGAdapter::Add(VoidSequence* input)` will be overloaded so that different input methods can be used to add data to the automaton, e.g.
    * `SCDAWGAdapter::Add(const std::string input&)`
    * `SCDAWGAdapter::Add(const char* input)`
    * etc.
- adapted methods/functions are spelled with a capital letter
- newly added methods are spelled regularly using lower case letters
- Following the adapter class design pattern, the adapting class holds a pointer to its adaptee:
  - Class `SCDAWGAdapter` will hold a pointer to the `struct SCDAWG` from the C-Layer as `SCDAWG* C_SCDAWG` as a member.
- all adapter classes end in `...Adapter` as a naming convention.

Construction and Destruction Handling in Adapter Classes. Construction and Destruction are handled transparently in the following way:

- An adapter class holds a private member:
  - `VoidSequenceAdapter` will hold a private member `C_VoidSequence` which will point to the `struct tVoidSequence`
  - (Ideally) exactly only one constructor per adapter class, will call `VoidSequenceInitialize()` to do a proper initialization.
    * The other constructors available to the adapter class are merely to make configuration easier on the user's end and will internally do *Constructor-Forwarding* as needed
  - The destructor `~VoidSequenceAdapter()` will make a call to `VoidSequenceFree(C_VoidSequence)` in order to release the pointer and to tear down the struct properly.

9.3 Implementation Specifics: Layer System Description

As was mentioned in the section "Adapter Classes", the library presented here is inherently tightly coupled to a *layered design*.

The already existing code base (in pure and fast ANSI C) comprises implementations for Compacted Automata (i.e. automata with implicit states, named `CompressedAutomaton`), Ukkonen's and Inenaga's CDAWG algorithms, and the algorithm for constructing SCDAWGs as described in [10].

This pure-C-base presents the grounds for the adaptation of the automata described to function as Document-Indexing Automata.

To make implementation of such document-indexing automata easier and safer to use, the latter is achieved by wrapping ("adapting") the pure-C base and thus leading to a "layered system". Find an overview illustration in figure 30 on the next page.

Please also check the UML graphs in the technical documentation generated by `doxygen` (cf. Doxygen integration on page 84).

9.3.1 Advantages of the layered system

Many of the points mentioned in the section "Advantages of Adapter Classes" are equally valid in the description of the layered system:

- The layered systems keeps on to the immense *speed* of the pure C implementation

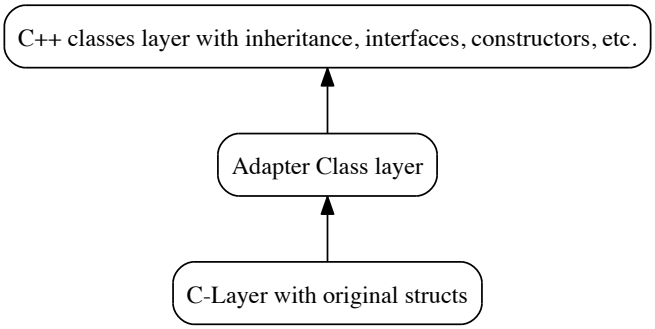

Figure 30: Layered system overview

- the layered system makes both, usage and implementation *easier* and *safer* (see again the Description of Advantages of the Adapter Pattern)
- Through the layered/adapted system, *clear inheritance*, *composition* and *templatization* are made possible.
- In many respects a document-indexing CDAWG and SCDAWG pretty much behave the same. This let's us abstract even a little more, and through the use of *generative programming* allows us to have the appropriate code generated for us through *templatization* (see more on templatization in "Implementation specifics: the templated DocumentIndexingAutomaton").

9.3.2 Conventions used in the Layered System

- You will find the adapted pure C code base in `src/bas/lml` (Bulgarian Academy of Sciences / Linguistic Modelling Department).
- Keep in mind, that for this whole approach none of this code had to be changed.
- Check `src/lmu/cis/sis/adapter` for the corresponding adapter classes (Ludwig-Maximilian-Universität München (LMU) / Centrum für Informations- und Sprachverarbeitung (CIS) / Symmetric Index Structures).
- Check `src/lmu/cis/sis/indexer` for the indexer classes building on the adapter classes.

9.4 Implementation specifics: Staging system

This section describes the automaton's different stages during construction and while working on building a text retrieval system and with the automaton itself.

9.4.1 Staging system idea

During construction, and before and while processing and indexing text, the automaton, at any time, can find itself in exactly one specific stage.

The set of different stages is the following, in progressive order:

- `EMPTY`: automaton is empty and has not been filled yet.
- `UNSORTED`: text has been read-in (or new text has been added) – the transitions are not sorted yet.
- `SORTED`: after having read-in new text, the transitions have been sorted (this is an internal, but nevertheless crucial operation.)
- `SHRUNK`: the index of transitions has been shrunk (internal, optional)
- `TARJANTABLE`: a tarjantable has been added to the automaton to reduce size and improve lookups (optional).
- `INDEXED`: the automaton has been indexed, i.e. bookkeeping was done in order to be able to do text-retrieval
- `CLOSED`: the automaton has been closed, i.e. the additional `buildhelp`-structures needed during construction have been freed (note: this action is un-recoverable)

These stages are *consecutive* in the sense that each of the steps depends on the former as a minimum base. Some of the stages are optional (see above) and some of these stages are *un-recoverable* and can not be taken back (this is especially true for `CLOSED`) – for the latter, please compare with the dotted lines in figure 31 on page 76.

All stages however will need to be re-done, if an action of a previous stage was executed in the meantime.

Example: `SORTED` / `UNSORTED` stage: You can add additional text, but eventually you will have to re-sort the transitions.

In this sense, specific methods can only be executed after specific needed actions have been executed beforehand (e.g. sorting, indexing).

In simpler words, for certain methods, the automaton needs to be in a specific stage – figure 31 on page 76 should explain this.

For instance, `Index()` needs to happen

- right before `Close()` (if to be followed by text retrieval operations)
- some time after `SortTransitions()`

Assertions (at run-time) are available to test these conditions when certain methods, like the following are called:

- `Write(FILE * file)`
- `Close()`
- `findall(string pattern)`
- `AddTarjanTable()`

For all these methods you will find additional information in the class's documentation as to which stage they belong to and which stage they incur on the automaton (consult the section "Doxygen integration" for pointers to the documentation and hints on generating the documentation yourself).

In some cases, the appropriate actions might be taken for you as a convenience, however, you should not rely on them, since they can have undesirable effects which are not possible to recover from, so control is mostly up to you (actions of the latter form would not be taken, so you should not rely on them being executed for you).

Anyway, keep in mind that many of these actions can at times be expensive operations and, since this is C-World, usually, "you don't pay for what you don't use".

However, to be on the safe side, as was mentioned, you will get checks done on the current stage of the automaton and the executability of your desired action wherever possible.

9.4.2 In-depth description of stages

To guarantee robustness of the automaton at all times, the current stage is saved and enforced using assertions where possible.

Note that derived classes will have to make sure to properly work with these conditions (respect them and set them).

The general rationale behind stages is that

- some operations are essential to certain other operations, and
- some operations can be expensive and unnecessary for certain other tasks.

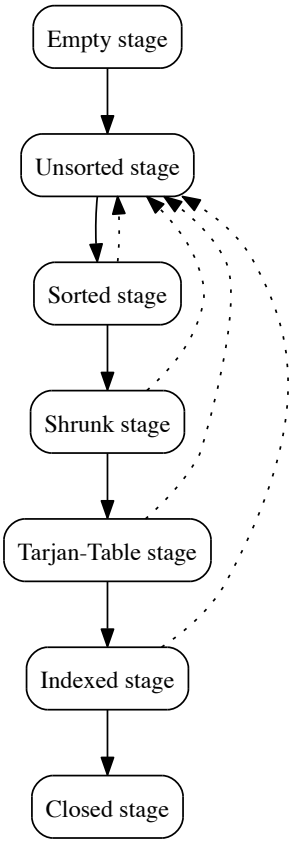

Figure 31: Staging system overview

In other words (and other contexts) one would say that these are "lazy" operations, i.e. certain operations will only be executed at the "latest time possible" in order to cut down on potentially unneeded and expensive computation/run-time.

Description of stages: `EMPTY` stage.

Right after construction, the automaton finds itself in `EMPTY` stage.

In most cases, at this time, new documents will be added, which sets the automaton to `UNSORTED` stage.

Description of stages: `UNSORTED` stage.

After new documents have been added to the automaton (whether it be from `EMPTY` stage or, say, `SORTED` stage), the automaton is set to / set back to `UNSORTED` stage.

For consecutive actions, the automaton will have to be `SORTED`, i.e. a call to the specific class' method `SortTransitions()` will be necessary.

Description of stages: `SORTED` stage.

After calling `SortTransitions()` the automaton is in `SORTED` stage.

This is a stage necessary to (internally) clean up pointers of transitions in order to be able to get to the following stages described below.

Externally speaking, this is a crucial stage needed for all following stages.

Most likely, `INDEXED` or `CLOSED` stage will be a favorable next stage for document retrieval and serialization of the automaton.

Description of stages: `SHRUNK` stage.

Between `SORTED`, `INDEXED` and `CLOSED` stage, `SHRUNK` stage exists, in order to reduce on the memory footprint of the final automaton.

It will cut down on unneeded pointers, transitions and memory allocations which are waiting for potential new documents to be added.

This stage is especially useful for `TARJANTABLE` stage and reduction of memory allocation.

Note however, that this stage is optional to the main stages `UNSORTED`, `SORTED`, `INDEXED` and `CLOSED` and might be considered optional as well as "internal" (but nevertheless "useful").

Description of stages: `TARJANTABLE` stage.

At this stage, a TarjanTable[43] is added to the automaton to give it "sparse table compression".

[43] a sparse table – cf. the Glossary

This is an extremly powerful method to reduce the size of huge, but sparse, arrays and can save a lot of memory needed to save the automaton's transitions.

Note again, that this is a stage optional to the main stages `UNSORTED`, `SORTED`, `INDEXED` and `CLOSED`.

Description of stages: `INDEXED` stage.

For all document retrieval tasks, `INDEXED` stage is the necessary, crucial stage.

`Index()`-ing of the automaton is an operation which only makes sense at a certain point and may be expensive (although it is implemented to run in $\mathcal{O}(n)$, linear time).

The `Index()` method is in charge of associating the automaton's internal states with the documents added to make retrieval of patterns possible.

Description of stages: `CLOSED` stage.

During construction and for most stages before the `Close()`-ing step, the automaton carries extra information in its class-specific `Buildhelp` struct in its underlying C-listayer.

After construction and during the `Close()`-ing step, this struct is used to "finalize" the automaton, but will be deleted in the closing step to cut down on memory needed, since it is not needed for any retrieval steps following after indexing.

For this reason, setting the automaton to `CLOSED` stage, again reduces on the memory footprint of the automaton by `free()`-ing and releasing the internal `BuildHelp`-structure(s).

Note that this is a non-recoverable step. Adding new documents to a `CLOSED` automaton will not be possible and the automaton would have to be rebuilt from scratch in order to add new documents to a `CLOSED` automaton.

Note as well, that some methods can only be invoked in post-`CLOSED` state, see "Post-Closed Methods" in the respective class' technical documentation for a comprehensive list.

For example, `Write(FILE* fp)`-like serialization methods expect the automaton to be in `CLOSED` stage, since `Read(FILE* fp)`-like de-serialization methods expect to read a `CLOSED` stage automaton.

9.5 Implementation specifics: the templated DocumentIndexingAutomaton

As proven in sections "Wrapping C", "Adapter Classes" and, also, "Transition to cmake build system", the approach of adapting the existing pure C code base through writing adapter classes around the structs, not only gave way to

- easier and safer handling of the structures
- explicitely stated relations between the structures and inhertiability for future classes
- possibility of adding unit-, integration- and regression-tests

but also the existence of the structures as full C++ classes has yet another, compelling advantage: *templatibility*.

9.5.1 DocumentIndexingAutomaton as a template

For the task at hand of making the existing index structures additionally carry inverted file information, i.e. turning a simple SCDAWG into a document-indexing SCDAWG, there generally exist two distinct routes to take:

- *Inherit* `SCDAWG` and give it additional functionality through a new class `DocumentIndexingSCDAWG`.
- Write a *template class* `DocumentIndexingAutomaton`.

Although, it might seem, at first sight, less straight-forward, writing a templated class `DocumentIndexingAutomaton` has one major advantage:

Imagine, we had 10 different index structures which all were to become extended to be document-indexing automata. Now, of course, is it possible to extend each and every class through inheritance and adding the new desired behavior as a `DocumentIndexingStructureX`, `DocumentIndexingStructureY`, `DocumentIndexingStructureZ`, and so forth (keep in mind, this would produce 10 more classes in the library which, in turn, need individual maintenance).

The other approach, achieving the same goal through *one and only one* template class for the whole *family of interrelated structures* in one go not only has the advantage of being

- more general
- less error-prone (in the long run)
- more “DRY”-ly (“don’t repeat yourself”)

but it simply extends, as said, a *whole family* of automata with the exact same behavior (while still staying open to specializations of individual automata should they need such extra treatment in some cases) through one *class template*.

Code example. Imagine we were taking the first approach, extending class `SCDAWG` by inheriting from it and forming a new class `DocumentIndexingSCDAWG` with additional `InvertedFileInformation`:

```
/* The simple inheritance approach */
class DocumentIndexingSCDAWG : public SCDAWG {
    protected:   /* Protected so others can inherit from us */
      InvertedFileInformation info_; /* Adding a new member */
};
```

The same would be done individually for `CDAWG`s, `SuffixTree`s, `SuffixTrie`s, and so on…

Albeit less easily readable on first sight, the template class to extend all of the named automata in one go is not very much different:

```
/* The class template approach */
template<typename AutomatonType>
class DocumentIndexing<AutomatonType> : public AutomatonType {
    protected:   /* Protected so others can inherit from us */
      InvertedFileInformation info_; /* Same as before here */
};
```

A template class really is nothing more than a class that is generated for us, hence the term *"generative programming"*.

Admittedly, in some places templated classes are not as easy and straight-forward as the respective one-by-one approach, but instead of writing a dedicated class `DocumentIndexingSCDAWG` we obtain it simply by asking for one such getting generated through a call to `DocumentIndexing<SCDAWG>`.

Keep in mind though, that for obtaining a `DocumentIndexing<InenagaCDAWG>`, we only need to instantiate it and no `DocumentIndexingInenagaCDAWG` class needs explicit declaration and definition. This approach keeps the code base small, maintenance to a minimum and generality between the individual classes coherent.

The compiler essentially is doing a copy-paste-like replacement, and as such is writing a class definition on its own. Of course, the same effect could be achieved by using `#define`s and such, but, since this is C++, this approach really is safe in that all code generated is passed on to regular compilation and thus, errors are spotted easily by the compiler.

9.6 Specific new features of C++11

This section is an outline and a small introduction to (only a very tiny part) of the new features of the newly released standard of the C++ language, named "C++11" that were used in this work.

9.6.1 auto keyword

In C++11, the `auto` keyword is one of the first and frontmost nice features that is able to not only ease the software development process itself but also is of great benefit to the "end user" when using the automaton library.

As a small example of its power, former ways to iterate over the elements of a container can be reduced to the following through a process called "*type inference*":

```
/* The "old" way to iterate the elements of a container */
for(std::vector<std::pair<std::string,int> >::const_iterator it =
        freqs.begin(); it != freqs.end(); ++it)
    {
      std::cout << it->first << std::endl;
    }

/* Put the auto keyword to practice
   let the compiler infer the type of the iterator */
for (auto it = freqs.begin(); it != freqs.end(); ++it) {
    std::cout << it->first << std::endl;
}
```

Better yet, the new standard – and through the powers of the `auto` keyword – makes the latter loop even easier to write and boils down the aforementioned code fragment examples to the following:

```
for (auto it: freqs) {
    std::cout << it->first << std::endl;
}
```

9.6.2 Constructor delegation

Unfortunately, *Constructor delegation* was not yet existent at the time of this writing with gcc 4.7.0 but would have been helpful especially for the setting of the members `C_symbolsize` through the cascade of interrelated interface classes. Somehow, when passing on information in constructors in this way, this information can (still) get lost

and special precautions need to be taken to avoid loss of infomation in instantiation – this was taken care of but can be improved in the future.

Whatever the situation is on constructor delegation, one point that deserves special mention at this point as far as instantiation of objects is concerned, is the changed rule on object instantiation in C++11:

In C++11 an object is existent as soon, as the *first* constructor has finished and returned – compared to the object being constructed only when the *last* constructor has finished, as in the previous standard of the language:

> "*In klassischem C++ ist ein Objekt fertig konstruiert, wenn sein Konstruktor ausgeführt wurde. Dies ändert sich mit C++11. Hier gilt: Sobald der erste Konstruktor fertig ausgeführt wurde, ist das Objekt fertig konstruiert. Das bedeutet natürlich, dass jeder weitere Konstruktor auf einem fertig konstruierten Objekt agiert.*" ("In classic C++ an object is readily constructed when its constructor returns. This is changing in C++11: The rule is, as soon as the first constructor was executed and has returned, the object is readily constructed. This of course means that all following constructors can operate on a readily constructed object.")
>
> [12, pp. 147, especially p.151 "fertig konstruiertes Objekt" (German source)]

9.6.3 Compilers

Obviously, and as seen in the section constructor delegation, compilers still need time to implement and support the whole set of features of the new standard.

Users will have to check on the availability of the new set of features in their respective compilers.

This is a classical trade-off situation that is to be decided on the following questions:

- Should one avoid features of the new standard for the moment?
- Where will my library be deployed?
- Will I deliver pre-(cross-)compiled source code?
- Which features does my compiler support?
- How fast will the new compilers be available on the target machines:
  - How fast are system administrators going to provide them?
  - Will the users of the library be able to install a new compiler?

- Can I get enough support in case of problems?
- Are the new features going to help the development process enough to justify using the new standard?

Under the assumption that the new standard of the language is going to leverage the development process and that the library in its current state will not (yet!) be used by a wide audience of users and needs additional work by a few dedicated contributors, the question whether to use the new standard was decided in favor of actually using it.

In combination with a powerful build system (see the next section "Transition to cmake build system"), good documentation (see the section doxygen support) and a version controlled code base (through `git`), contribution is eased and the development process is opened to more contributors by lowering the entry bars considerably.

9.7 Transition to cmake build system

Transitioning the current code base to the *cmake* build system is guided by the principles of modern software development. Among its many advantages, the following points need special mention:

- cross-platform support and deployment
- vast IDE support
- easy setup and maintenance (vs. the classic GNU build system[44])
- integrated testing facilities (unit testing, integration testing)
- integrated cross-platform packaging

9.7.1 CMake build framework

The open source build framework `cmake` by Kitware is transforming into a de-facto standard of modern C/C++-software build system[45] in software development processes.

[44] sometimes also referred to "autohell"

[45] For a list of organizations using cmake, cf. `http://www.cmake.org/cmake/project/success.html`: Among the most presitigous and biggest, KDE needs mentioning.

9.7.2 Testing

In all brevity, *Test driven development (TDD)* has key advantages, such as:

- avoiding regressions in future development
- thus stabilizing the whole code base
- new coders to the project can immediately see if and how their changes to the code affect the code base

9.7.3 Unit tests

Central to all programming efforts of these days is the existence of an automated testing framework.

In light of the complexity of the existing code, tests were needed in order to confirm the efficiency of changes to the existing code while reducing regressions.

`cmake` makes it easy to write and provide tests to the library.

9.7.4 CPack packaging

Along with the many benefits that `cmake` offers, transparent and cross-platform packaging is a blessing.

This way, installers for different platforms can be created while not having to care about each of them individually. Find more information on CPack at http://cmake.org/.

9.8 Doxygen integration

Along with wrapping the current code base into C++ adapter classes, adding tests and packaging facilities through the cmake build framework, *doxygen* was used for commenting the newly written code.

Doxygen is – like cmake –, the state-of-the-art documentation system which produces nicely formatted and intelligent source code documentation, and additionally is able to produce UML graphs of the class designs, of callers and callees, in many different formats (html, LaTeX and many other documentation formats) among much else.

The current source code documentation can be found at the address http://www.cip.ifi.lmu.de/~bruder/sis/documentation/html/index.html for the html version and http://www.cip.ifi.lmu.de/~bruder/sis/documentation/latex for the LaTeX (book) version of the documentation. Please check the appendix section for additional pointers concerning the sources and availability.

10 Metrics

This chapter will present key figures on the code base itself, on the sizes of different automata and additionally will give execution times and measures on the indexing steps involved.

10.1 Code Metrics

In order to estimate the code base size, `sloccount`[46], is used as a tool[47] to (roughly) measure and quickly grasp code base sizes and implementation efforts.

As far as `sloccount`'s accuracy is concerned, it proves to be very exact by estimating the newly written code base's development efforts at about 4.5 months of work which gets very close to the time that was available for working on the code.

10.1.1 Adapted Code base size metrics

As for the adapted code base `sloccount` the numbers show the following:

```
SLOC    Directory   SLOC-by-Language (Sorted)
3638    bas             ansic=3638


Totals grouped by language (dominant language first):
ansic:         3638 (100.00%)


Total Physical Source Lines of Code (SLOC)                = 3,638
Development Effort Estim., Person-Years (Person-Months)   = 0.78 (9.31)
(Basic COCOMO model, Person-Months = 2.4 * (KSLOC**1.05))
Schedule Estimate, Years (Months)                         = 0.49 (5.84)
(Basic COCOMO model, Months = 2.5 * (person-months**0.38))
Estimated Average Number of Developers (Effort/Schedule)  = 1.60
Total Estimated Cost to Develop                           = $ 104,845
(average salary = $56,286/year, overhead = 2.40).
```

10.1.2 Newly written indexing automaton code base

As for the newly written indexing automaton and adapter classes code base `sloccount`'s output:

```
SLOC    Directory   SLOC-by-Language (Sorted)
1712    lmu             cpp=1573,ansic=139
```

[46] "sloc" is a common acronym for Single Lines Of Code. Compare with "kloc" (1000 lines of code), etc. Also cf. the glossary.

[47] the figures presented here are generated using David A. Wheeler's 'SLOCCount' http://www.dwheeler.com/sloccount/.

```
Totals grouped by language (dominant language first):
cpp:           1573 (91.88%)
ansic:          139 (8.12%)


Total Physical Source Lines of Code (SLOC)                = 1,712
Development Effort Estim., Person-Years (Person-Months)   = 0.35 (4.22)
 (Basic COCOMO model, Person-Months = 2.4 * (KSLOC**1.05))
Schedule Estimate, Years (Months)                         = 0.36 (4.32)
 (Basic COCOMO model, Months = 2.5 * (person-months**0.38))
Estimated Average Number of Developers (Effort/Schedule)  = 0.98
Total Estimated Cost to Develop                           = $ 47,514
 (average salary = $56,286/year, overhead = 2.40).
```

10.1.3 Whole code base size metrics

Adding in to the pre-existing code and the newly written indexing automaton code base the test cases written and the cmake build system files, `sloccount` gives us the following final measure on the code:

```
SLOC    Directory   SLOC-by-Language (Sorted)
3638    bas             ansic=3638
1444    lmu             cpp=1444
268     CMakeFiles      ansic=139,cpp=129
248     t               cpp=248
162     top_dir         cpp=162


Totals grouped by language (dominant language first):
ansic:         3777 (65.57%)
cpp:           1983 (34.43%)


Total Physical Source Lines of Code (SLOC)                = 5,760
Development Effort Estim., Person-Years (Person-Months)   = 1.26 (15.09)
 (Basic COCOMO model, Person-Months = 2.4 * (KSLOC**1.05))
Schedule Estimate, Years (Months)                         = 0.58 (7.01)
 (Basic COCOMO model, Months = 2.5 * (person-months**0.38))
Estimated Average Number of Developers (Effort/Schedule)  = 2.15
Total Estimated Cost to Develop                           = $ 169,858
(average salary = $56,286/year, overhead = 2.40).
```

10.2 Indexing time metrics

All test times have been measured on shell-level using the unix tool `time`.

In this series of tests, works of philosopher Friedrich Nietzsche have been used as data[48] on three different machines

- Apple MacBook Air (in the following: MBA)[49].

[48] Nietzsches "Morgenröthe" and "Menschliches, Allzumenschliches" part I, from [21]

[49] 13-inch, Mid 2011, 1.7 GHz Intel Core i5 (Processor), 4 GB 1333 MHz DDR3 (Memory), 1 Processor, 2 physical Cores, L2 Cache (per Core): 256KB, L3 Cache 3MB

- Apple iMac (referred to as "iMac" in the following)[50].
- "Calculus"[51]

Since all machines have about the same processor speed, the times are very close as expected. Although already blazingly fast (thanks to the underlying C-layer) at roughly 30 words / millisecond, these times can at least be cut down by 50% (when using two cores to fill the two CDAWGs individually) by implementing the ideas on multithreading pointed out in "Possible areas of improvement". More speed could be gained by intelligently applying multithreading techniques to the individual steps that are carried out.

In this series of texts, the files are 1) read, 2) added to the structure, the 3) transitions sorted, 4) indexed and 5) the automaton gets closed.[52]

The number of words is a rough figure from the Unix program `wc` (word count) to give some sense on the data that is processed.

All tests were run 25 times to derive more accurate run times. As an interpretable key performance indicator, the "number of words" (as rough as it is) that can be indexed in one millisecond is given for each machine.

Machine	Bytes	Words	Median Time (sec)	w/ms
MBA	697,033	104,176	3,940	26.44
iMac	697,033	104,176	3,322	31.36
Calculus	697,033	104,176	3,381	30.81

Table 6: Indexing 697033 Bytes (0.697 MB) of text in a document-indexing CDAWG.

The same test is carried out for a document-indexing SCDAWG. As expected, the construction time grows linearly, since the SCDAWG structure is composed of two CDAWG structures.[53]

[50] 27-inch, Mid 2011, 2.7 GHz Intel Core i5 (Processor), 16GB 1333 MHz DDR (Memory), 1 Processor, 4 Cores, L2 (per Core) 256KB, L3 Cache 6MB

[51] Fujitsu RX600S5, 19-inch 4HU, 2011, Intel(R) Xeon(R) CPU X7560 @ 2.27GHz, 4x8=32 physical Cores, 1024GB RAM 1033MHz DDR3 (PC3-8500)

[52] cf. "In-depth description of stages"

[53] cf. "SCDAWG structure overview"; also compare "Multithreading support"

Machine	Bytes	Words	Median Time (sec)	w/ms
MBA	697,033	104,176	6,120	17.02
iMac	697,033	104,176	5,306	19.63
Calculus	697,033	104,176	5,110	20.37

Table 7: Indexing 697033 Bytes (0.697 MB) of text in a document-indexing SCDAWG.

10.3 Automaton Size metrics

The next set of figures is aimed at observing the structure growth size. Again, these tests are carried out for both document-indexing CDAWGs and SCDAWGs. As expected, the figures grow with linear complexity, with the SCDAWG counting in at two times the CDAWG's size.

An interesting point for discussion is the last value given, the size-growth factor. Since the $\mathcal{O}(n)$ complexity measure only states the linearity of growth w.r.t. the input length it nevertheless says nothing about the factor involved. Therefore this factor proves to be an interesting figure.

In (bytes)	In (MB)	Out (bytes)	Out (MB)	Factor
697,033	0.66	8,162,382	7.8	11.71

Table 8: Input/Output size metrics for CDAWGs

In (bytes)	In (MB)	Out (bytes)	Out (MB)	Factor
697,033	0.66	15,422,406	15	22.12

Table 9: Input/Output size metrics for SCDAWGs

Next, to arrive at clearer metrics of index structures, even bigger parts of philosopher F. Nietzsches works are indexed in an SCDAWG structure. The input data is weighing in at 12.77 MB, or roughly 2 million words. The drop of words that are indexed per millisecond is both, interesting and startling at the same time. It definitely shows the need for "heavy load" tests and demands for further investigation. The reasons do not seem clear at this stage, although memory swapping and/or repeated memory allocations and de-allocations (on machine-level) and the internal `SortTransitions()`[54] routine are suspected to cause this drop at this point.

[54] compare "In-depth description of stages"

In (MB)	Out (MB)	Factor	t (sec)	Words	w/ms
12.77	275	21.55	326	1,942,903	6.00

Table 10: Heavy load statistics for SCDAWG

The figures presented below are from the same test set of text, involving about 1.9 million words.

number of data symbols	18,596,902
number of states (per SCDAWG)	3,501,505
number of transitions	11,338,695
total number of transitions	23,245,921

Table 11: Heavy load statistics for SCDAWG (internal figures)

11 Summary

11.1 Wrap-Up

After establishing the basics to indexing texts with Finite State Automata in section "A motivating example" and covering the basic definitions in chapter 3, chapter 4 presented the Ukkonen algorithm to indexing all suffixes of a text base in form of a Suffix Tree.

Following these descriptions, chapter 5, "Building the smallest automaton possible" showed how to obtain the CDAWG structure for a text base $\mathcal{B}$ through the Inenaga construction algorithm. Figure 32 again illustrates the Inenaga construction algorithm's derivation from the Ukkonen construction algorithm for Suffix Trees.

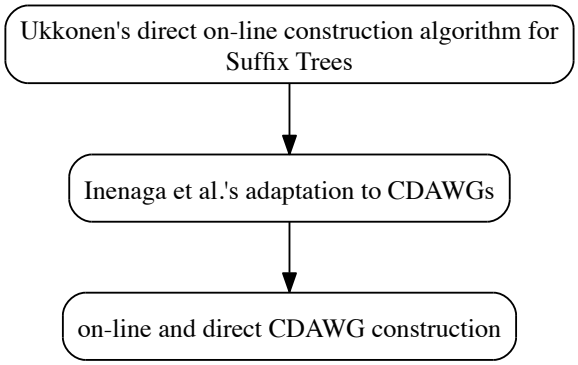

Figure 32: Graphical Overview of CDAWG construction algorithm evolution

Subsequently, chapters 6 and 7 built on the CDAWG structure to build the SCDAWG structure with document indexing abilities. This is again summarized by figure 33 on the next page.

Chapter 9 then went into clarifying how the existing code base was adapted and how the new class `DocumentIndexingAutomaton<AutomatonType>` was created. These points are illustrated and summarized again in figure 34 on the following page.

11.2 A few words on the current State of the Art

Due to the tight time constraints given for this work, several nice features and a lot of what originally was on the roadmap had to be left for future work.

Two things slowed down the highly-set goals considerably:

- getting to grips with the complex algorithms

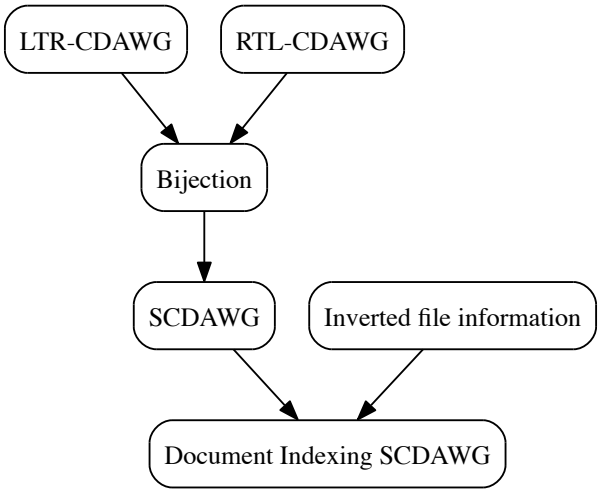

Figure 33: Graphical Overview of document-indexing SCDAWG construction

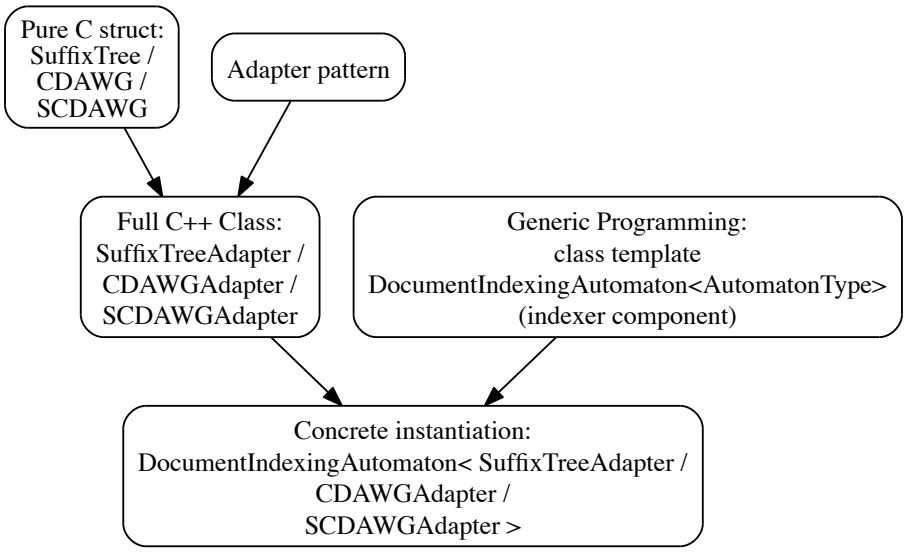

Figure 34: Graphical Overview document-indexing SCDAWG construction (implementation side)

- adapting the, at the time, existing code base and getting it into shape for the tasks at hand while leveraging future efforts.

Especially the latter proved to be more error prone than initially expected. Although the code base was in good shape, as soon as one starts to work with it in a different way, things usually start to break. In the case here, as a first step, the focus was shifted to bringing the code base into a shape that will leverage future work on the automata.

In the following, I will give hints and explanations for such future work and hope to successfully lower the entry bars for everyone willing to contribute.

11.3 Possible areas of improvement

As far as areas of improvement are concerned, a few points spring to mind:

11.3.1 Accessibility improvements

Language bindings. Giving bindings to other languages might be a useful idea. For example, bindings to Perl are possible and generally done through `XS` (eXternal Subroutines). One could think of a perl module to make calls to the cautomata library. Having the C/C++-library "available" from an "easier to write" language such as `perl`, certainly can help make the library more accessible.

Documentation improvements. As always, the *documentation* is at the heart of future development. Although most of the concepts and methodologies described in the section Adapting the existing automata code base can be found in the *"Related Pages"* section of the doxygen-documentation, having contributors read and improve the documentation always is of great help to future contributors.

Several improvements on the code base itself would be of great use and calls for additional contributors. Part of this work was to make contribution easier by lowering the bars of getting "in medias res" and from this base, the following ideas should be kept in mind when looking for improvements. For those interested in contributing, please see the following sections for first ideas on areas of improvement as far as the current code base is concerned.

11.3.2 Multithreading support

With the code base being adapted and with the new standard of the C++-language being established in the current code base, the facilities for multithreading that the new standard supports are getting more and more attractive. Indeed, the first areas that can benefit from multithreading support would be the following:

Multithreading support in filling the separate CDAWGs. Since the two independent LTR and RTL CDAWG structures are filled in separate steps (and only in a later step their respective isomorphic states are identified), this step seems ideal for multithreading support: While the first automaton is indexing the LTR directed text, the RTL-CDAWG can independently index the RTL-directed text from the same file (only in reversed order) without breaking any structures whatsoever or having to do additional bookkeeping / mutexing / locking of specific states for the adding of transitions.

Multithreading support in pre-reading the files to be added. Once this previously mentioned step is established, one can think of having a separate process for the slow reading of the files from the hard disk and keeping the next file "in stock" for the other two processes described above, taking care of the adding of the contents to the separate structures. Once one structure is finished it could immediately start adding the next file's content without having to first go into the (relatively) slow reading of the next file.

Multithreading support in adding to the structure with Ukkonen's algorithm possible? A theoretically very interesting idea for further going research can be the following: Is the Inenaga algorithm capable and agnostic to when and how different parts are added to the CDAWG structure? Is it possible to add two files at the same time to the very same CDAWG structure using the Inenaga algorithm? In other words, since one of the principles in Ukkonen's algorithm tells us that "a leaf is always a leaf" and since in the Inenaga algorithm each document that is added is granted its very own sinkstate (read: "leaf") and adding additional information will in all cases only lead to a finer-grained branching of the current tree, is it possible to add, say, two files to the same structure at the same time without breaking the language of the automaton?

The question is a rather tricky one and to answer the question whether all this is possible without locking states and doing additional bookkeeping will ultimately be decided upon the moving of the so called "active point" which is crucial for the decision of where the next branching will occur.

Surely, this is a very interesting field for further theoretical work as far as Ukkonen's and the Inenaga algorithm are concerned. (Recall Hopcroft et al.'s words from the introductory sections in [14], that theoretical work in the field of finite state automata was rather neglected for more pragmatic approaches to this very complex world…)

11.3.3 General improvement ideas

Improvements on the sorting algorithm. One more idea of improving the existing current code base is directed at the underlying C implementation: After adding new documents to the automata, in a next step the transitions need to get sorted. The current sort algorithm can be implemented differently to run in faster time, with a better complexity in "big O-terms". Currently it is not linear.

Since the sorting step is a) crucial to all work with the index structures and b) lots of speed can be gained in this step, it seems like a good place for speed improvements. Looking at the output of a code metrics analyzer like `valgrind` might bring clarity as to whether `SortTransitions()` is a "bottleneck"-procedure and can benefit from improvements in terms of execution speed.

Bit-shifting the tarjan table ("sparse table"). As far as the (optional) tarjan table is concerned: one idea might be to represent a state and its transitions for a fixed alphabet (potentially only sparsely filled) through bit vectors. Each state has an array of fixed size for each letter of the alphabet, pointing to the state that the transition with this specific letter leads to. In the sparse table compression where two such arrays are shifted against each other to find a possible matching, one can imagine using bit-shift operations for this step and thus, by being even closer to the machine, speeding up the sparse table compression as a whole.

Discussions with Petar Mitankin[55] show that this approach seems worthwhile trying.

However, it is true that one will have to take special care of the following: Since the number of transitions of a state exceeds the boundaries of a machine word when represented as a vector of binary information, one needs to take extra special care in terms of the correctness of the bit-shifts.

Luckily, C++ features a specially designed class to take care of a scenario like this one: `std::bitset`, defined in the header file `<bitset>`.

Serialization. However, what's needed in an earlier (and easier) step, will be serialization support for reading in and writing to disk for the newly created generic index structure `DocumentIndexingAutomaton<AutomatonType>` that carries the additional "inverted file" information of each state.

In order to keep implementation time minimal, one idea is to keep the existing serialization and deserialization routines of the current underlying C layer, the adaptee, and in the wrapping step, have the adaptee read/write his/her own respective bytes, and, following the adaptee's steps, read/write additional bytes to the files from the adapter layer.

[55] cf. "Thanks" and "Acknowledgements"

11.3.4 User Interface improvements

Last but definitely not least will be improvements to the user facing interface.

First of all, a good idea will be to have separate executable programs for the following tasks:

Indexer executable: `msgrep-index`. Pre-Index a set of documents (files) in a directory and serialize the generated index to a (hidden) file in that same directory for repeated and independent lookups and use, and thus making sure that the next time `msgrep` is called, it can directly work on that pre-generated index. Obviously, checks will need to be made whether all files that have been indexed are still available in that directory or warnings need to be emitted otherwise.

Lookup executable: `msgrep-interactive`. Unfortunately the implementation and adaptation of the existing automata library proved to be more time-consuming than initially expected and the interactive user-facing interface parts giving suggestions, accepting choices and generally helping the user "drill down" to the text-parts she is interested in, had to be shifted to a later time.

Here lies an interesting field not only for researchers of the field of finite state automata, but also interaction designers and researchers from the field of "human–machine interaction":

How can suggestions to the left best be designed and offered to the user? How can the user most effectively choose suggestions and new results and how can these suggestions "to the left" and "to the right" best be presented? It really is a pity that this work could not be taken up to this point, but it definitely still is a very attractive field for further research, also in terms of inter-disciplinary research projects.

Commandline interface / Web interface. Of equal importance are the user-facing parts in terms of commandline interface and keeping to the standards established in the Unix world. A consistent and more user friendly commandline interface definitely is one of the very next steps where improvements are needed. Following a good commandline interface, a web interface will be a next step.

Different walking and search behaviors. In hindsight of the time constraints for the work presented here, several obvious next-step features had to be left omitted.

First of all, on the user-interface and user-friendliness level, a case insensitive search for the pattern p will be the next feature to implement in `msgrep`.

On the academic level, different "walking behaviors" could be implemented, like the following:

- walking is possible by locating the pattern p in the index structure (using `delta`) and from there, by recursively following outgoing transitions up to a certain depth. Keep in mind, for the SCDAWG, this can be done in both directions.
- while recursively collecting possible extensions, the output can be tweaked as to give markers where transitions have been taken. Effectively, this means to indicate where and how *equivalence classes are spread* in small text bases up to large text bases. From there, several more fields of interest open up for further going linguistic research as far as recurring patterns in terms of a lower, finer-grained in-word level are concerned.
- walking is possible by locating the correct position in the `data_`-store and extending to the left and to the right from there. Although this walking behavior already is implemented, for concordance view, cleanups to the output are essential. For instance, all whitespace is reduced to a single space in order to keep the concordance view clean. A lot of further going user interface improvements can be made here.

11.3.5 Functional improvements

For now, the approximate search functionality described in [10] is not yet implemented, i.e. the automaton of the `msgrep` example of chapter "Code Usage Example" is not yet fully functional in the sense that it could do approximate search as well. One can find the description of the algorithm in [10].

A Compilation and Installation procedure

All source code of this project is contained in the `src/` directory in the tree structure of this project. The source code also contains the technical documentation which can be generated using `doxygen` (cf. Requirements).

As far as the source code is concerned, three main directories can be distinguished:

- `src/bas/`, containing the original C code base, from the Bulgarian Academy of Sciences by Petar Mitankin
- `src/lmu/`, containing the main project files to this project, i.e. the wrapper classes in `adapter/` and the document indexing classes in `indexer/`.
- another directory of great use and, especially when looking for use-case examples is `t/`, the test directory.

After getting the sources, and fulfilling the Requirements, the sources can be compiled and installed.

A.1 Getting the sources

The sources are available, a) either on the CD shipped with this printed work, or b) from http://www.cip.ifi.lmu.de/~bruder/sis/ from where the version controlled sources can be downloaded by

issuing the following command

```
wget http://www.cip.ifi.lmu.de/~bruder/sis/sis-latest.tar.gz
```

followed by the unpacking command:

```
tar xvzf sis-latest.tar.gz
```

A.2 Structure of this project

```
.                                   # root of the project
|-- CMakeLists.txt                  # base CMakeLists.txt file
|-- Doxyfile                        # generated from Doxyfile.in
|-- Doxyfile.in                     # Will be used by cmake
|-- build                           # Directory to build from
|-- config.cmake                    # Basic configuration for *.in files
|-- etc                             # text files to test automata, etc.
|   |-- N                           # text files containing Nietzsche works
```

```
|   |   `-- [...]
|   |-- N2
|   |   `-- [...]
|   `-- [...]
|-- lib                             # libraries generated from src/
|   |-- libcautomata.a
|   |-- libcppautomataadapter.a
|   `-- libcppindexer.a
|-- src                             # sources
|   |-- CMakeLists.txt
|   |-- bas                         # Bulgarian Academy of Sciences sources
|   |   |-- CMakeLists.txt
|   |   `-- lml
|   |       `-- [...]
|   |-- config                      # Files used to configure the sources
|   |   |-- sis_config.hpp
|   |   `-- sis_config.hpp.in
|   |-- lmu                         # Sources written for this work
|   |   `-- cis
|   |       `-- sis
|   |           |-- CMakeLists.txt
|   |           |-- adapter         # Adapter classes to bas/lml
|   |           |   `-- [...]
|   |           |-- cppbase.hpp
|   |           |-- indexer         # Document indexing automata
|   |           |   `-- [...]
|   |           `-- utility.hpp
|   |-- main.cpp
|   `-- t                           # Unit test sources
|       `-- [...]
`-- var                             # folder for various files:
    `-- t3.stat                     # logs, stats, etc.
```

A.3 Requirements

For a successful compilation of this project the following requirements need to be fulfilled:

- the GNU compilers `gcc` and `g++` with `g++` > 4.7.0 (or any suitable compiler with C++11 support)[56]
- `cmake` > 2.8.3 (to build the `Makefile`)[57]

[56] http://gcc.gnu.org/
[57] http://www.cmake.org/

- `doxygen` > 1.7.5.1 (optionally, to build the technical documentation)[58]

A.4 Compilation

Compilation of the sources is done in a multi-step process. Generally, one should do so-called "out-of-source-builds" with `cmake`, i.e. one should not compile the sources from within the source tree. What this means is, that one will create a `build`-directory from where all `cmake`, `make` and compilation steps will be carried out.

To this end, the steps in successfully building the project follow along these lines (more detailed descriptions will follow right after):

```
cd ~/sis; \                          # Change to project's directory
rm -rf build; \                      # delete (old) build directory
mkdir build; \                       # Create a clean build directory
cd build; \                          # switch to it
cmake \                              # create Makefiles by running cmake
  -DCMAKE_C_COMPILER=/usr/bin/gcc \     # set specific c and c++ compilers
  -DCMAKE_CXX_COMPILER=/usr/local/bin/g++-4.7 \
  -DCMAKE_BUILD_TYPE=Release .. ; \     # set the build type
make; \                              # run make (compile)
make test; \                         # run tests (optional)
bin/caut                             # run (any executable from build/bin/)
```

It is important to note the following points:

- it is essential to set a suitable compiler for the C++-files. It needs to be capable of compiling according to the new C++11 standard. It is set using `-DCMAKE_CXX_COMPILER=`.
- All files will be built in the `build/` directory. You will find the executables in `build/bin/`
- `make doc` offers a separate target in the Makefile to build the technical documentation using `doxygen`

[58] http://doxygen.org/

B Glossary

Alphabet Finite set of symbols

Canonical Representative the longest member of an equivalence class

CLI Command-line Interface

Compacted Automaton an automaton with compacted transitions, i.e. not all states need to be explicit.

Compressed Automaton Petar Mitankin's term in the C base implementation for "Compacted Automaton"

DAWG Directed Acyclic Word Graph:

- all states are accepting
- "Crochemore has pointed out that with a different assignment of accepting states, this DFA is the smallest automaton for the set of all suffixes of w."[3]

DFA Deterministic Finite (State) Automaton

FSA Finite State Automata consist of *states* and *transitions* between states that react to input. They are useful to a variety of software applications, for example the lexical analysis components of compilers and text retrieval programs[Compare 14, end of chapter 1].

KLOC abbreviation for 1K = 1000 Lines Of Code, compare with SLOC

Language a (potentially infinite) set of *strings*, whose symbols are drawn from one and the same *alphabet* [following 14, end of chapter 1].

LTR left-to-right (reading direction, automaton)

LRS longest repeated suffix (of a word w)

Partial DFA "Let a partial deterministic finite automaton be a DFA in which each state need not have a transition edge for each letter of the alphabet." [3, p. 31].

Regular Expressions a structural notation to describe patterns which can be represented through a finite state automaton [following 14, end of chapter 1].

RTL right-to-left (reading direction, automaton)

SLOC abbreviation for Single Lines Of Code, compare with KLOC

String a finite set of symbols [following 14, end of chapter 1].

Suffix Tree a.k.a. "'compact position trees', earlier as 'PATRICIA trees'"[3, p. 31.]

On-line algorithm "in the strong sense"[3] "At each stage of our construction, the automaton will be correct for the prefix of w that has been processed."[3, p. 39]

C Colophon

This work was written in TextMate 1.5.11 on a Mac OSX 10.7.3 using the excellent markup language `pandoc` 1.9.1.2 with inline graphviz and from there, converted to LaTeX and typeset using a custom shell script. The whole typesetting process took about half a minute to render the final pdf. `pandoc` also makes it possible to render presentations and websites from the same sources without changing the input. It builds on Markdown and MultiMarkdown and makes writing technical documentation and scientific works a breeze.

pandoc: http://www.johnmacfarlane.net/pandoc/.

D Acknowledgements

This work heavily builds on the work by Klaus U. Schulz, Stoyan Mihov, Petar Mitankin and Stefan Gerdjikoff and their to-be-published paper [10].

The existing code base was written by Petar Mitankin and was adapted and extended as described in this paper.

Many discussion with Klaus Schulz and Petar Mitankin and my fellow colleagues Estelle Perez, Florian Fink, David Kaumanns and Patrick Seebauer tremendously helped to deeply understand the algorithms themselves as well as the occasional edge cases.

E Thanks

In my final words I would like to express my sincerest thanks to the people mentioned here and whose input has been immensely helpful in covering the difficult topic of this thesis:

- Front and foremost, I would like to thank Prof. Klaus U. Schulz not only for providing me with this interesting topic for a thesis from the field of Finite State Automata, but most for his invaluable expertise, his patience and thoughtfulness and his unmistakeable capability of breaking down the hardest-to-cover topics into easily understandable examples and pictures. I value these wholeheartedly.
- Second, Petar Mitankin and his incredibly well-thought out and cleverly designed base implementation of many of the algorithms to build on that comprised the aforementioned basic C-layer of this work. Before reading his code, I thought I was proficient in reading and writing C. Now I really am.
- Estelle Perez, my fellow student colleague, for her insightfulness, patience and eager to understand even the hardest of algorithms, while still having her own thesis to write and for all the tracing back Petar's Code in places where I was lost. Not to forget the invaluable discussions on the design of this work. Thank you, thank you, thank you, Estelle!
- Dr. Max Hadersbeck, my academic mentor who not only made computational linguistics a hobby for me, but who is a friend as good as you can imagine: even in hard times, he was always there for me, a friend and mentor you can count on.
- My fellow students Patrick Seebauer, David Kaumanns and Florian Fink for their discussions to help clear things up when I was lost or in search for new ideas as well as for their dedicated proof reading and valuable input. Thank you guys, you rock!
- My long time friend Claudia Iberle for her extensive proof reading.
- My friends and family, foremost my beloved sister Franciska, again, for all your patience during my absence while writing the thesis. I couldn't imagine what I would be doing without you.

List of Tables

List of Figures

References

[1] Alfred V. Aho and Margaret J. Corasick. “Efficient string matching: an aid to bibliographic search”. In: *Commun. ACM* 18 (6 1975), pp. 333–340.

[2] A. Blumer et al. “Complete inverted files for efficient text retrieval and analysis”. In: *Journal of the Association for Computing Machinery* 34.3 (1987), pp. 578–595.

[3] A. Blumer et al. “The smallest automation recognizing the subwords of a text”. In: *Theoretical Computer Science* 40 (1985). Eleventh International Colloquium on Automata, Languages and Programming, pp. 31–55.

[4] Marshall Cline. *C++ FAQ*. Aug. 2012. URL: http://www.parashift.com/c++-faq/index.html.

[5] Maxime Crochemore. “Transducers and repetitions”. In: *Theoretical Computer Science* 45.0 (1986), pp. 63–86.

[6] Maxime Crochemore and Renaud Vérin. “On compact directed acyclic word graphs”. In: *Structures in Logic and Computer Science*. Ed. by Jan Mycielski, Grzegorz Rozenberg, and Arto Salomaa. Vol. 1261. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1997, pp. 192–211.

[7] Jan Daciuk et al. “Incremental construction of minimal acyclic finite-state automata”. In: *Computational Linguistics* 26.1 (2000), pp. 3–16.

[8] Paul J. Deitel. *C++ How to Program (6th Edition)*. 6th ed. Prentice Hall, Aug. 2007.

[9] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st ed. Addison-Wesley Professional, Nov. 1994.

[10] Stefan Gerdjikov et al. “Good parts first – a new algorithm for approximate search in lexica”. To appear. 2012.

[11] Robert Giegerich and Stefan Kurtz. “From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction”. In: *Algorithmica* 19 (1997), pp. 331–353.

[12] R. Grimm. *C++11: Der Leitfaden für Programmierer zum neuen Standard*. Programmer’s Choice. Addison Wesley, 2011.

[13] Max Hadersbeck. *Programmierung mit C++ für Computerlinguisten*. CIS Internal Book, 2010.

[14] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to automata theory, languages, and computation*. Pearson/Addison Wesley, 2007.

[15] Andrew Hunt and David Thomas. *Der Pragmatische Programmierer*. Hanser Fachbuch, Mar. 2003.

[16] Shunsuke Inenaga et al. “On-Line Construction of Compact Directed Acyclic Word Graphs”. In: *Word Journal Of The International Linguistic Association* 146.2 (2005), pp. 1–12.

[17] Shunsuke Inenaga et al. “On-Line Construction of Symmetric Compact Directed Acyclic Word Graphs”. In: *Proceedings of the 8th International Symposium on String Processing and Information Retrieval (SPIRE'01)*. IEEE Computer Society, 2001, pp. 96–110.

[18] Edward M. McCreight. “A Space-Economical Suffix Tree Construction Algorithm”. In: *Journal of the Association for Computing Machinery* 23.2 (1976), pp. 262–272.

[19] Scott Meyers. *Effektiv C++ programmieren*. Addison Wesley Verlag, Feb. 1998.

[20] Mark Nelson. *Fast String Searching With Suffix Trees*. 1996. URL: http://marknelson.us/1996/08/01/suffix-trees/.

[21] Friedrich Nietzsche. *Friedrich Nietzsche, Digital critical edition of the complete works and letters, based on the critical text by G. Colli and M. Montinari, Berlin/New York, de Gruyter 1967—, edited by Paolo D'Iorio*. 2012. URL: http://nietzschesource.org/.

[22] stackoverflow/jogojapan. *[Answer to] Ukkonen's suffix tree algorithm in plain English?* [without concrete date]. URL: http://stackoverflow.com/a/9513423.

[23] Esko Ukkonen. “On-line construction of suffix-trees”. In: *Algorithmica* 14.3 (1995), pp. 249–260.

[24] Peter Weiner. “Linear pattern matching algorithms”. In: *Proceedings of 14th IEEE Annual Symposium on Switching and Automata Theory*. 1973, pp. 1–11.

[25] Wikipedia. *Inverted index*. 2012. URL: http://en.wikipedia.org/wiki/Inverted_file.

[26] Wikipedia. *Regular language*. 2012. URL: http://en.wikipedia.org/wiki/Regular_language.

[27] Wikipedia. *Suffix tree*. 2012. URL: https://en.wikipedia.org/wiki/Suffix_tree.

[28] Wikipedia. *Trie*. 2012. URL: http://en.wikipedia.org/wiki/Trie.