Master's thesis

Master's Programme in Computer Science

Fast Implementation of Shortest Common Superstring Approximation with Application to Relative Lempel-Ziv Dictionary Construction

Arttu Kilpinen

May 23, 2022

Faculty of Science

University of Helsinki

Supervisor(s)

Assoc. Prof. Simon Puglisi

Examiner(s)

Prof. Leena Salmela

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Master's Programme in Computer Science

Tekijä — Författare — Author

Arttu Kilpinen

Työn nimi — Arbetets titel — Title

Fast Implementation of Shortest Common Superstring Approximation with Application to Relative Lempel-Ziv Dictionary Construction

Ohjaajat — Handledare — Supervisors

Assoc. Prof. Simon Puglisi

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's thesis	May 23, 2022	68 pages

Tiivistelmä — Referat — Abstract

The objective of the shortest common superstring problem is to find a string of minimum length that contains all keywords in the given input as substrings. Shortest common superstrings have many applications in the fields of data compression and bioinformatics. For example, a common superstring can be seen as a compressed form of the keywords it is generated from.

Since the shortest common superstring problem is NP-hard, we focus on the approximation algorithms that implement a so-called greedy heuristic. It turns out that the actual shortest common superstring is not always needed. Instead, it is often enough to find an approximate solution of sufficient quality.

We provide an implementation of the Ukkonen's linear time algorithm for the greedy heuristic. The practical performance of this implementation is measured by comparing it to another implementation of the same heuristic. We also hypothesize that shortest common superstrings can be potentially used to improve the compression ratio of the Relative Lempel-Ziv data compression algorithm. This hypothesis is examined and shown to be valid.

ACM Computing Classification System (CCS)
Theory of Computation → Design and Analysis of Algorithms → Data Structures Design and Analysis → Data Compression
Theory of Computation → Design and Analysis of Algorithms → Data Structures Design and Analysis → Pattern Matching
Theory of Computation → Design and Analysis of Algorithms → Data Structures Design and Analysis → Sorting and Searching

Avainsanat — Nyckelord — Keywords

shortest common superstring, relative lempel-ziv, implementation

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Algorithms study track

Contents

List of Algorithms

1 Introduction

Textually represented data – text – is one of the most common types of preserved information. It occurs in many forms, e.g., source code, markup languages and plain text. Also, DNA and amino acid sequences are often modeled as text. In stringology, a field of computer science that studies text processing and manipulation, this kind of data is modeled as finite sequences of symbols called strings. Due to the large variety of applications, stringology covers many types of problems. For example, programming and editing text documents, as well as more comprehensive areas such as bioinformatics and data compression, involve some degree of string processing. Stringology is also one of the most studied fields in computer science (Crochemore and Rytter, 2003).

One of the most fundamental and common problems for textual data is the String Matching problem (also known as string searching). That is, given a text string and a pattern the objective is to output every location of the text string that contains the pattern as a substring (Gusfield, 1997). The generalized version of the String Matching Problem is the Multiple String Matching problem where the pattern is replaced with a set of patterns, while the objective is to report the locations for all of them. Since many practical applications require solving string matching problems, it is fortunate that these problems can be solved efficiently. For example, Knuth-Morris-Pratt (Knuth et al., 1977) and Two-Way (Crochemore and Perrin, 1991) algorithms solve the string matching problem in linear time. Many other polynomial-time algorithms also exist, see e.g., (Karp and Rabin, 1987) and (Horspool, 1980). The multiple string matching problem can be efficiently solved, for example, with algorithms presented in (Aho and Corasick, 1975), (Navarro and Raffinot, 2000), (Karp and Rabin, 1987) and (Commentz-Walter, 1979).

Stringology also covers many optimization problems assumed to be intractable, e.g., Closest String, Longest Common Subsequence, Maximal Strip Recovery, and Shortest Common Superstring, to name a few (Bulteau et al., 2014). Although there are many tractable special cases of these problems, they are nevertheless NP-hard, i.e., there are no efficient algorithms known to solve these problems for arbitrary inputs. This means that often suboptimal decisions must be settled for. For many applications, an optimal solution is however not always required. In many cases it is enough to find

a solution of sufficient quality. That can be obtained with so-called approximation algorithms. Depending on the problem, there are efficient heuristics to be used. For example, the greedy heuristic for the Shortest Common Superstring problem, which we focus on this thesis, produces very good results in practice (Romero et al., 2004).

The objective of the Shortest Common Superstring (SCS) problem is to find a string of minimum length that contains all keywords in the given input. One obvious application of the SCS is data compression. Since the SCS is usually shorter than the input keywords, it can be seen as a packed form of the input. To compress a set of keywords, one can compute an SCS and address the keywords as a pair of position and length, thus decreasing the redundancy of the saved data, for example.

Another application of the SCS problem is DNA sequencing, i.e., the construction of a complete model of nucleotides of some biological unit, e.g., a single gene, chromosome or a genome. Direct DNA sequencing (sequencing a whole strand as single piece) methods can only be applied for small nucleotide sequences, while the process is infeasible for larger units. To sequence a longer strand of DNA, a number of its copies are first replicated. The multiple strands of the same sequence are then split from several semi-random places, yielding a set of different subsequencences that overlap each other. These shorter nucleotide strands are then sequenced and modeled as a set of strings. What remains is to map those randomly permutated sequences in order to find their correct natural order. This fragment assembly problem can be simplified to the SCS problem (Peltola et al., 1983).

As the decision version of the SCS problem is NP-complete (J. Gallant et al., 1980), several heuristics have been developed to compute the approximate solution. The quality of these heuristics can be defined in terms of compression, that is, the number of symbols by which the common superstring is shorter than the input keywords combined. For the greedy heuristic, which works by repeatedly merging the two strings with a maximal overlap together, first described in (J. K. Gallant, 1982), the compression has been shown to be at least half of the optimal (Tarhio and Ukkonen, 1988). Another measure for the approximation is the length of the obtained superstring. (Blum et al., 1994) has shown that the approximation ratio is at least 4. Later on, a tighter upper bound of 3.5, has been proven (Kaplan and Shafrir, 2005). A conjencture (Tarhio and Ukkonen, 1988) claiming the approximation ratio to be 2 remains unresolved. Despite this, the greedy heuristic is observed to result in very good approximation ratios, in practice. A theoretical explanation for this is discussed in (Ma, 2008). For other heuristics, the best proven approximation

ratio is $2 + \frac{11}{30}$ (Paluch, 2014).

For the aforementioned data compression and DNA sequencing related problems, approximate solutions of the SCS problem can be used. There is no inevitable requirement to get an actual shortest common superstring. For data compression problems, the length of the superstring is only a matter of efficiency. A good approximation is simply enough.

One alternative to implement the greedy heuristic for the Shortest Common Superstring problem is to solve the All-Pairs Suffix-Prefix (APSP) problem (Lim and Park, 2017) for the input keywords and construct the common superstring repeatedly selecting a pair of strings with a maximal overlap. Tarhio and Ukkonen (1988) describes an algorithm that works in this manner. The APSP problem is solved using a Knut-Morris-Pratt algorithm, giving a time complexity of $O(mn)$, where m is the number of input keywords and n is the total length of the input. This also dominates the time complexity of the whole algorithm. To the best of our knowledge, no publicly available implementation of this algorithm exists. After the publication of (Tarhio and Ukkonen, 1988), more efficient algorithms for APSP problem has been developed (Gusfield et al., 1992). Theoretically, these could be combined with the bookkeeping of (Tarhio and Ukkonen, 1988) to implement the greedy heuristic in time $O(n + m^2)$.

There are also algorithms that compute the greedy heuristic without calculating the APSP problem. For example, (Ukkonen, 1990) uses the so-called Aho-Corasick machine to find the overlaps. In this way, at most $2m$ overlaps need to be examined. This significantly decreases the time complexity of the computation. In fact, depending on the chosen data structures, Ukkonen's algorithm can be implemented to run in linear time. Another algorithm of this kind is described in (Alanko and Norri, 2017). The time complexity of this algorithm is $O(n \cdot \log(|\Sigma|))$, where Σ is the alphabet. This algorithm is implemented using the SDLS library (Gog et al., 2014). The publication also has some theoretical comparisons of the latter two algorithms. No practical comparisons were made since at the time of writing of (Alanko and Norri, 2017), there was no publicly available implementation of (Ukkonen, 1990). It has been stated in (Alanko and Norri, 2017) that it would be interesting to compare these two algorithms in practice.

Motivated by the above discussion, in this thesis, Ukkonen's linear time approximation algorithm for the greedy heuristic of the Shortest Common Superstring problem is further studied. We provide an implementation of the algorithm to examine its practical capabilities. We compare the runtimes of our implementation to the implementation of (Alanko and Norri, 2017). We also hypothesize that the shortest common superstring

approximation can be used to improve the quality of the Relative Lempel-Ziv (RLZ) data compression algorithm. As RLZ dictionaries for general-purpose data are conventionally constructed by concatenating random samples of the input, in theory, depending on the repetitiveness of the input, the dictionary could be highly compressed. Using this feature, the overall compression could be increased while the dictionary size decreases. This hypothesis is tested and validated in this thesis.

The structure of the thesis is as follows. In Chapter 2 we briefly discuss the preliminaries needed to understand the rest of the thesis. Chapter 3 gives the detailed description of Ukkonen's algorithm for the Shortest Common Superstring problem. Besides the pseudocode, which is discussed in four parts, Chapter 3 includes the formal definition of the greedy heuristic as well as discussion of the Aho-Corasick machine, which is used to implement the heuristic. In addition, we prove the correctness of Ukkonen's algorithm and discuss the time complexities of all of the sections. The concept of dictionary compression is introduced in Chapter 4. We discuss the Lempel-Ziv family of compression algorithms. The Relative Lempel-Ziv method is also defined. In Chapter 5 we detail our contribution to the topic. Section 5.1 comes as a documentation of our implementation of Ukkonen's algorithm. The datasets and experiment environments are described in Sections 5.2 and 5.3, respectively. The results are discussed in Section 5.4. Finally, we give a conclusion of the thesis in Chapter 6.

2 Terminology and Definitions

Since the Shortest Common Superstring problem is an essential part of this thesis, we first define the notations and terminology of strings in order to formally define the problem. The Shortest Common Superstring problem is solved as the Longest Hamiltonian Path problem. Therefore, that problem and its related concepts are also recalled. To make the contents of this chapter more readable, each definition is preceded with a corresponding explanation in natural language.

Regardless of the alphabet, a string is simply a finite sequence of symbols. The length of a string equals the number of symbols it contains. Sequences of symbols (as well as other types of elements) are denoted by writing the elements adjacently. A dot symbol $\cdot$ is also used to emphasize the concatenation of two symbols or strings.

Definition 2.1 (String)**.**
For an alphabet Σ, a string s is a finite sequence of symbols $\sigma_1 \cdots \sigma_n$ where each symbol $\sigma_i \in \Sigma$. The length of a string s denoted by $|s|$ is the number of symbols in the string. An empty string denoted by ε contains zero symbols. An ith symbol of a string s is denoted by $s[i]$.

In the theoretical discussion of Chapter 3 we assume an integer alphabet where any symbol can be stored in $\log(|\Sigma|)$ bits, where $|\Sigma|$ is the size of the alphabet.

For a pair of strings we define the concepts of substring, superstring, prefix, suffix and overlap as follows.

The concepts of a prefix and a suffix are used to define an overlap. A string u is a prefix of a string s if the first $|u|$ symbols of s spells out u. Similarly, a string v is a suffix of a string s if the last $|v|$ symbols of s spells out v. Note that the prefix or the suffix may also be empty.

Definition 2.2 (Prefix)**.**
Let s and u be strings. The string u is a prefix of s if and only if $s = ut$ for some string t.

Definition 2.3 (Suffix)**.**
Let s and v be strings. The string v is a suffix of s if and only if $s = tv$ for some string t.

Now we define the overlap of two strings to be a string that is a prefix of the first string and a suffix of the other. Note the antisymmetry of the definition. The overlap between the first and the second string is not necessarily the same as the overlap between the second and the first string. Trivially, any two strings have an empty overlap.

Definition 2.4 (Overlap and Maximal Overlap)**.**
Let s and t be strings. A string w is an overlap between s and t if and only if w is a prefix of s and a suffix of t. If w is the longest such string, it is said to be maximal overlap.

If a string s contains another string s' we say that the string s' is a substring of the string s and string s is a superstring of s'. By containing a string we mean that there is a sequence of symbols in s that spells out s'.

Definition 2.5 (Substring and Superstring)**.**
Let there be two strings s and s'. The string s' is a substring of s if and only if s can be written as $u \cdot s' \cdot v$, where u and v are strings. Any of the strings u and v may be empty. If u and v are both empty then $s = s'$ which is a substring of itself. An empty string is trivially a substring of every other string. The string s' is a substring of s if and only if s is a superstring of s'. A substring of s with length k starting at index i is denoted as $s[i...k + i - 1]$.

Note that by Definition 2.5, prefixes and suffixes are also substrings.

Let us now define a common superstring and a shortest common superstring for a set of strings using the previous definitions. If a string contains (i.e., is a superstring of) each string in a set of strings, it is said to be a common superstring of the set.

Definition 2.6 (Common superstring)**.**
Let $R = \{s_1, ..., s_m\}$ be a set of strings and s' be a string. The string s' is a common superstring of R if and only if s' is a superstring for each s_i such that $i \in 1, ..., m$.

Trivially, a common superstring of a set of strings can be constructed by concatenating all of the strings of the set together. This kind of string has a length that equals the lengths of all the substrings added together.

Example 2.7:

Let $R =$ { “baa”, “baba”, “abab”, “aab”} be a set of strings. The string “baababaababaab” is a common superstring of the set R.

In many cases, there exists one or more common superstrings that are shorter than the common superstring produced by this naive method. If a common superstring has the shortest length possible, it is said to be a shortest common superstring (SCS).

> **Example 2.8:**
>
> Let $R =$ { "baa", "baba", "abab", "aab"} be a set of strings. A string "aababaa" is a shortest common superstring of the set R.

Definition 2.9 (Shortest Common Superstring)**.**
Let string s be a common superstring of a set of strings R. The string s is a shortest common superstring of R if and only if there does not exist a common superstring of R which is shorter than s.

A reduced set of strings is a set of strings such that no string in the set is a substring of another string in that set. This property is important since the later discussion of the reduction between the Shortest Common Superstring and Longest Hamiltonian Path problems only works for reduced sets of strings. The SCS of the reduced set of strings is the same as the SCS of the same set of strings that is not reduced.

Definition 2.10 (Reduced Set of Strings)**.**
Let $R = \{s_1, s_2, ..., s_m\}$ be a set of strings. The set R is called reduced if and only if s_i is not a substring of s_j for all $i, j = 1, ..., m, i \neq j$.

Lemma 2.11. Let R be a set of strings such that $s, s' \in R$ and s' is a substring of s. The shortest common superstrings of R and $R\backslash\{s'\}$ are the same.

Proof. Let s_{scs} be a shortest common superstring of a set $R\backslash\{s'\}$. Since, $s \in R\backslash\{s'\}$ and s' is a substring of s, the string s_{scs} is a superstring of s'. Adding a string to a set can not decrease the length of its corresponding shortest common superstring. Therefore s_{scs} is also a shortest common superstring of the set R. □

Now we have defined the necessary string-related concepts. Next the Shortest Common Superstring problem is defined as follows

Definition 2.12 (Shortest Common Superstring Problem)**.**
Given a set of strings R, the shortest common superstring problem asks to find a shortest common superstring of a set R. If there are multiple such strings, any of them can be returned.

The algorithm presented in Chapter 3 solves the approximation of the Shortest Common Superstring problem by reducing it to the Longest Hamiltonian Path problem in a weighted digraph. For this reason, we will also define the necessary terminology to understand the latter problem. The proof of the reducibility is given in Chapter 3.

For any given graph, a Hamiltonian path in a graph is a path (sequence of connected vertices) that contains every vertex in the graph exactly once.

Definition 2.13 (Hamiltonian Path)**.**
Let $G = (V, E)$ be a graph. A Hamiltonian path in G is a $|V|$ sized sequence of vertices that contains each vertex in V exactly once and for each consecutive vertices v_i and v_j there is an edge $(v_i, v_j) \in E$.

For a weighted graph, a longest Hamiltonian path is a Hamiltonian path of maximum length. i.e., the sum of the weights of the edges is maximal.

Definition 2.14 (Longest Hamiltonian Path)**.**
Let $G = (V, E, w)$ be a weighted graph. A longest Hamiltonian path in G is a Hamiltonian path $v_1 \cdots v_{|V|}$ such that $w((v_1, v_2)) + \cdots + w((v_{|V|-1}, v_{|V|}))$ is maximized.

Finally, we define the Longest Hamiltonian Path problem as follows.

Definition 2.15 (Longest Hamiltonian Path Problem)**.**
Given a weighted graph G, the longest Hamiltonian path problem asks to find a longest Hamiltonian path of G. If there are multiple such paths, any of them can be returned.

3 Ukkonen's Algorithm

In this chapter, we describe Ukkonen's linear time approximation algorithm (Ukkonen, 1990) for Shortest Common Superstring problem in detail. First, we define the heuristic that the algorithm uses in terms of strings. Then we show that the problem can be reduced to the problem of finding a longest Hamiltonian path in a specific overlap graph that encodes the same SCS problem. After the problems are proven to be equivalent, we define the same heuristic for the reduced problem. The equivalence of the two approaches are necessary, since the bounds of the achieved compression are proven using the Longest Hamiltonian Path problem. Conceptually, Ukkonen's algorithm solves the approximate SCS problem by constructing the approximate longest Hamiltonian path in a corresponding overlap graph.

Further, we define the Aho-Corasick (AC for short) machine, as it is used in the algorithm for finding the maximal overlaps between the set of input keywords. The time complexity of the construction of the AC-machine is also discussed. We also prove that the AC-machine can be used to find the maximal pairwise overlaps between every input keyword.

This chapter contains four pseudocode blocks, which essentially form the algorithm as a whole. For each of those algorithms, we first discuss their principles on a higher level. After this, the pseudocode is presented and discussed in more detail. Finally, we prove their time complexities. The algorithms represented here are slightly changed and contain a few corrections from the form they were originally presented. Each such modification is added to footnotes to emphasize the differences between the algorithms presented here versus the original ones.

3.1 The Greedy Heuristic

Let $R = \{s_1, ..., s_m\}$ be a set of m strings. A naive method for constructing a common superstring for R is simply to concatenate all the strings in the set. For the set R, the resulting superstring would then be $s_1 s_2 \cdots s_m$. The following greedy heuristic can be used to construct an approximation of a shortest common superstring:

1. Examine the set and remove two strings s_i and s_j that have a longest overlap among all pairs of strings (s_i, s_j) with $i \neq j$. Note that a longest overlap may be an empty string. If there are multiple pairs of strings with a longest overlap, the decision can be made arbitrarily.

2. Merge the strings together such that they are maximally overlapped to form a shortest common superstring of the set $\{s_i, s_j\}$.

3. Add the new string back to the set R and repeat until there is only one string left.

It turns out that the Shortest Common Superstring problem for a reduced set of strings is analogous with a special case of the Longest Hamiltonian Path problem, that is, every instance of the SCS problem can be encoded as an instance of the Longest Hamiltonian Path problem. The encoding works by creating a weighted complete digraph from the set of strings as follows. The graph $G_R = (V_R, E_R, w)$, where V_R is a set of vertices, E_R is a set of edges and w is a weight function defined for each edge in E_R, is constructed from the set of strings R. Let there be a vertex $v_i = s_i$ for each string in R. Moreover, we define the start vertex v_{start} and the end vertex v_{end}. Each vertex v_i has a directed weighted edge to each vertex v_j with $i \neq j$. The start vertex has an edge to each v_i and each v_i has an edge to the end vertex. The weight $w(v_{start}, v_i) = 0$ and the weight $w(v_i, v_{end}) = 0$ for all $i = 1, ..., m$. The weight of every other edge (v_i, v_j) is the length of the maximum overlap between s_i and s_j. More formally,

$G_R = (V_R, E_R, w)$ with

$$V_R = \{v_i = s_i \in R\} \cup \{v_{start}, v_{end}\},$$
$$E_R = \{(v_{start}, v_i), (v_i, v_{end}), (v_i, v_j) \mid i = 1, ..., m, j = 1, ..., m, i \neq j\},$$

and

$$w((v_s, v_d)) = \begin{cases} \text{length of the longest overlap between } v_s \text{ and } v_d & \text{when } v_s \in R \text{ and } v_d \in R, \\ 0 & \text{when } v_s = v_{start} \text{ and } v_d \in R, \\ 0 & \text{when } v_s \in R \text{ and } v_d = v_{end}. \end{cases}$$

Since the subgraph consisting of vertices $v_i, i = 1, ..., m$ is a complete graph, there exists a Hamiltonian path or paths. Given any of the Hamiltonian paths H from the graph G_R, we can construct a common superstring of R by overlapping the strings representing the vertices in the same order they appear on the path in the following way.

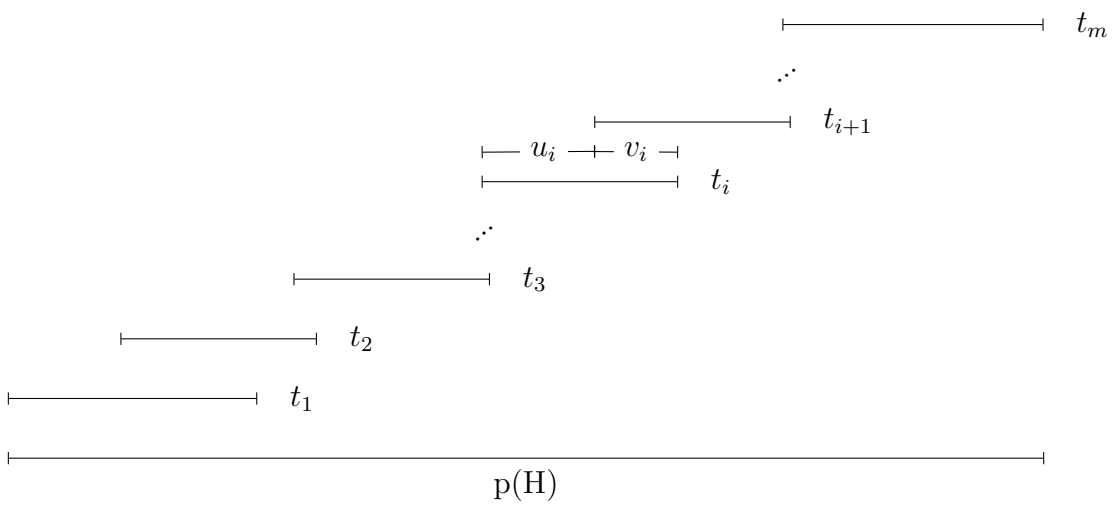

Figure 3.1: The projection $p(H)$ of the set of strings $\{s_1, ..., s_m\}$

Let us take the strings in the same order as their corresponding vertices appear in the path H. Put the strings above each other such that they are maximally overlapped. The projection $p(H)$ which describes the strings that are maximally overlapped (i.e., the overlaps are not duplicated) clearly results in a common superstring of R. Depending on the Hamiltonian path H the overlaps in the projection $p(H)$ have different lengths, hence the projection changes depending on the path H.

Figure 3.1 shows an example of the projection constructed from a Hamiltonian path H in an overlap graph. The corresponding set of strings consists of s_1 to s_m. The strings t_1 to t_m are the same strings in the order they appear on H. We can represent the string $t_i, i = 1, ..., m-1$ as a prefix-suffix pair (u_i, v_i) where v_i is the suffix of t_i that is also a prefix of t_{i+1} such that the overlap between t_i and t_{i+1} is maximal. The prefix of t_i whose length is $|t_i| - |v_i|$ is u_i, i.e., u_i is the part of t_i that can not be overlapped with t_{i+1}. Using this partitioning we can write the projection $p(H)$ as $u_1 u_2 \cdots u_{m-1} t_m$.

Since the maximal overlaps $v_i, i = 1, ..., m-1$ appear in the projection only once, the length of the projection is $(|s_1|+...+|s_m|)-(|v_1|+...+|v_{m-1}|)$. The length of the maximal overlaps are equivalent to the weights of the overlap graph G_R, and so $v_1 + \cdots + v_{m-1} = |H|$. As the size of $|H|$ grows, the size of the projection gets shorter. With this analogy, the size of H is the compression of the common superstring, and the length of $p(H)$ is the length of the common superstring. We now know that the Hamiltonian path in the overlap graph can be used to form a common superstring of the set of strings. Next, we show that also a shortest common superstring of R has a corresponding Hamiltonian path. To prove this, we introduce the following lemma.

Lemma 3.1. The compositions of $p(H)$ and a shortest common superstring s_{scs} for a reduced set of strings are equivalent.

Proof (by construction). Let there be a reduced set of strings $R = \{s_1, s_2, ..., s_m\}$ and a shortest common superstring s_{scs} for R. The indices of the strings s_1 to s_m are in the same order they appear in s_{scs}. Let v_i be the longest suffix of s_i such that it is also a prefix of s_{i+1}, i.e., there is an overlap v_i between s_i and s_{i+1}. Additionally, let us define u_i to be the prefix of s_i that does not overlap with v_i. By this construction we see that the composition of any SCS for any reduced set of strings is equivalent with the composition of the superstring acquired via the projection from the Hamiltonian path.

Hence, there exists a Hamiltonian path and the corresponding projection for any SCS.

□

Theorem 3.2. *The projection $p(H)$ for the longest Hamiltonian path in the overlap graph is equivalent to a shortest common superstring for a reduced set of strings.*

Proof sketch. By Lemma 3.1 there exists a decomposition of the SCS that corresponds to a Hamiltonian path in an overlap graph of a reduced set of strings. □

The following defines a conceptually equivalent greedy heuristic for the SCS problem in the terms of a Hamiltonian path in the overlap graph.

To construct an approximate longest Hamiltonian path, select an edge e from the set E_R such that

1. The edge e has the largest weight. If there are multiple such edges, the decision can be made arbitrarily.
2. The edge e has not been selected before.
3. It is possible to construct a Hamiltonian path with e and the already selected edges.

The third constraint above indicates the following:

i The selection does not form a cycle.

ii The earlier selected edges can not have the same start vertex as e.

iii The earlier selected edges can not have the same end vertex as e.

Theorem 3.3. *Let H' be the approximate longest Hamiltonian path in the overlap graph created using the greedy heuristic and let H be the actual longest Hamiltonian path. Then,* $|H'| \geq \frac{1}{2}|H|$.

In other words, the compression achieved by the greedy heuristic is at least half of the optimal compression. For the proof we refer the reader to (Tarhio and Ukkonen, 1988).

3.2 Aho-Corasick Machine

The Aho-Corasick (AC for short) machine is a finite state automaton (FSA) like data structure that was developed to be used for pattern matching to find occurrences of the given keywords (i.e., strings) in a larger text string (Aho and Corasick, 1975). The input is a finite set of strings called keywords and an arbitrary string called a text string. The output of the program is a list of locations of every occurrence of every keyword in the text. The AC-machine data structure and the associated pattern matching algorithm were first described in (Aho and Corasick, 1975).

As the string searching part of (Aho and Corasick, 1975) is not used in (Ukkonen, 1990) and is not otherwise relevant in this thesis, we only discuss the AC-machine as a data structure and omit the string searching functionality of the original description. Formally, the AC-machine is defined* as follows.

Definition 3.4. Aho-Corasick machine is a tuple (Q, Σ, g, f), where:

1. Q is a set of states, containing at least the start state;
2. Σ is the alphabet;
3. $g : Q \times \Sigma \rightarrow Q, g(q_s, \sigma) = q_d$ is a function that defines a state transition from a state q_s to a state q_d for a symbol σ;
4. $f : Q \rightarrow Q, f(q_s) = q_d$, is a function that defines a failure transition from state q_s to state q_d.

If there is a goto transition from q_s to q_d in an AC-machine, we say that the state q_s is the parent of the state q_d and that the state q_d is a child of the state q_s. A depth of a state q

*(Aho and Corasick, 1975) also defined an output function. This is needed only when the AC-machine is used for searching strings.

is the minimum number of state transitions that is needed to get q from the start state. We use a term failure path for a continuous sequence of failure transitions with decreasing depth.

When the AC-machine is used in a text search program, the goto function defines the state transition from start state q_s to the destination state q_d with the current input symbol $\sigma \in \Sigma$ of the state automaton. If no such goto transition exists, the failure function f is queried and the failure transition from the start state q_s to the destination state q_d occurs. The alphabet Σ contains every symbol in the set of keywords (and in case of a string search program, also every string in the processed text string) encoded by the AC-machine.

To construct the AC-machine for the set of keywords, the FSA like goto function is first created. The goto function is then used to construct the failure function, which completes the construction of the AC-machine. The goto function is similar to the child function of a trie, i.e., it describes a rooted tree that stores a set of keywords such that edges are labeled with the symbols of the alphabet and the edge label is different between each sibling node. We say that a node spells out a string that is formed by concatenating the edge labels from root to the node. Since there is an unambiguous correspondence between a state and a string, those words are occasionally used interchangeably. The failure function describes a chain of state transitions in the case when the goto function is not defined for the current input symbol. The failure function is queried as long as the goto function is not defined. In the AC-machine the goto function is defined for every symbol in the alphabet in the start state. Therefore, the failure function is not defined for the start state. Other states, whether the goto function is fully defined or not, always have a defined failure function value. Algorithms 1 and 2 are used as complete definitions of the goto and failure functions.

The AC-machine can be illustrated as a graph where each state is represented as a node and each goto transition is represented as a labeled edge. Failure transitions can also be drawn. We call such an illustration a goto graph of an AC-machine. In a goto graph, the start state is represented by a root node.

Figure 3.7 illustrates the AC-machine for the set of keywords $R = \{$“baa”, “baba”, “abab”, “aab”$\}$. The solid lines describe the goto function and the dashed lines describe the failure function, for example, $g(8, a) = 9$ and $f(4) = 11$.

Explanation of Algorithm 1. Algorithm 1 creates a goto function for a set of strings. It begins by setting the start state on line 2. The algorithm continues by looping over

Algorithm 1 Aho-Corasick machine: Construction of the goto function

Input: Set of keywords $R = \{s_1, s_2, ..., s_m\}$

Output: Goto function g that is initially undefined for all states

Preconditions: Initially $g(q, \sigma)$ is undefined for each state and symbol

```
1:  function CALCULATEGOTOFUNCTION(R)
2:      q ← 1
3:      for i ← 1 until m do
4:          insert(s_i)
5:      for σ s.t. g(1, σ) is not defined do
6:          g(1, σ) ← 1
7:  function INSERT(s = σ_0 σ_1 ··· σ_{k−1})
8:      p ← 1
9:      j ← 0
10:     while g(p, σ_j) is defined do
11:         p ← g(p, σ_j)
12:         j ← j + 1
13:     for l ← j until k − 1 do
14:         q ← q + 1
15:         g(p, σ_l) ← q
16:         p ← q
```

the set of keywords and inserting them to the goto function on lines 3 to 4. At the end of the main function, on lines 5 to 6, the goto function is defined for all $g(1, \sigma)$ that were undefined, creating a loop within the start state. This ensures that the failure function of the start state is never used when the AC-machine is used for searching strings.

The procedure *insert* inserts a single keyword to the goto function. The idea in the lines 10 to 12 is to skip the prefix of s that already exists in the trie. The variable j initialized in line 9 keeps track of this prefix. After the first symbol not already represented in the trie is found, the new branch is created and the remaining suffix of s is inserted symbol by symbol at lines 13-16.

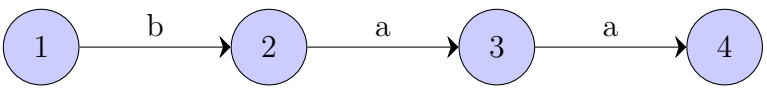

Figure 3.2: Illustration of the AC-machine after the first key is added in Example 3.5.

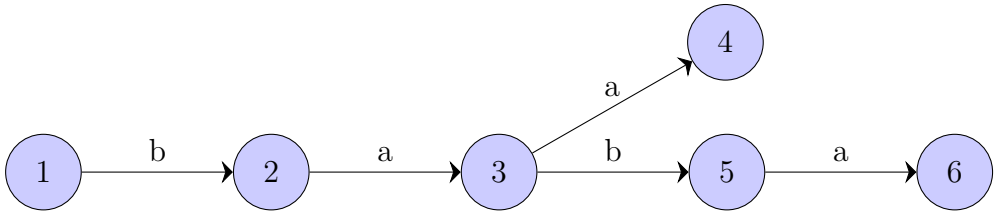

Figure 3.3: Illustration of the AC-machine after the second key is added in Example 3.5.

Example 3.5:

Let us simulate this algorithm with a set of keywords $R = \{$"baa", "baba", "abab", "aab"$\}$. The algorithm starts from line 2 where the start state (root node in the figures that illustrate the AC-machine) is created. Since there are four keywords, the first for loop is executed four times. In the first round, the *insert* procedure is called for the first keyword "baa". In the *insert* procedure, the first while loop is not executed since there are no outgoing edges from the root node yet. The for loop is executed for every symbol and the three nodes (2,3,4) are added to the goto graph with edge labels 'b', 'a', and 'a'. Figure 3.2 presents this state of the construction.

With the second round of the for loop, we add the keyword "baba". In this case, the *insert* procedure executes the while loop twice, since the common prefix of "baa" and "baba" has length 2. At this point, a branch is added to the goto graph and the remaining symbols 'b' and 'a' are inserted. Figure 3.3 corresponds with this state.

In the same way, the rest of the keywords are added. For keyword "baba" the insertion happens in the same way as with the first keyword, that is, the while loop in the *insert* procedure is not executed since there is no goto transition from the root with a symbol 'a'. For the last keyword, the branching of the goto graph happens again in a similar way to the second keyword. At this point, the goto function is a trie of the input keywords. After the first for loop in the main function is executed, the second loop on lines 5 to 6 creates an edge from root to root for every symbol in the alphabet such that $g(1, \sigma)$ is not defined. With this example, we already have an edge from the root for both of the symbols in the binary alphabet, hence no edges are created in this step. However, the corresponding edge is drawn in the image for clarity. This completes the execution of Algorithm 1. The resulting goto graph is described in Figure 3.4

Let us define a depth function for the AC-machine. At this point, depth is merely a concept of the machine, but later on it becomes a part of the data structure in Algorithm 3. In an AC-machine the depth of a state q, denoted by $d(q)$, is the depth of the node

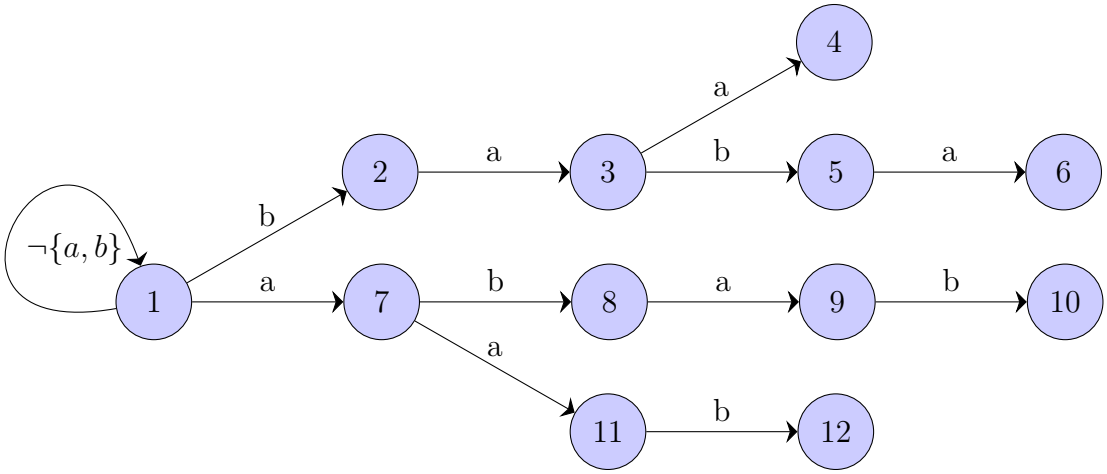

Figure 3.4: The complete goto graph for the set of keywords $R = \{$"baa", "baba", "abab", "aab"$\}$.

(number of edges from root to that node) that corresponds to the state in the goto graph. This is equivalent to the length of the string that represents the state, i.e., if a node q in the goto graph is represented by the string s, then $d(q) = |s|$.

Description of Algorithm 2. Next, we describe the construction of the failure function in more detail. The failure function for a state is calculated using the failure function of the parent of that state. Thus, the computation starts from the start state, and continues to the states of depth 1, the states of depth 2, and so on.

Since the failure function is not defined for the start state (depth 0), the calculation starts from depth 1. We define $f(q) = 1$ for all states q with depth $= 1$. Assuming the failure function is calculated for all states of depth $d - 1$, then the failure function of a state q is defined with the instructions in Figure 3.5.

1. Let p be the parent of the state q such that $g(p, \sigma) = q$.
2. Set $r = f(p)$.
3. Set $r = f(r)$ until $g(r, \sigma)$ is defined. Since $g(start, \sigma)$ is defined for all symbols, at some point, the condition is met.
4. Set $f(q) = r$.

Figure 3.5: Listing of the failure calculation for state s.

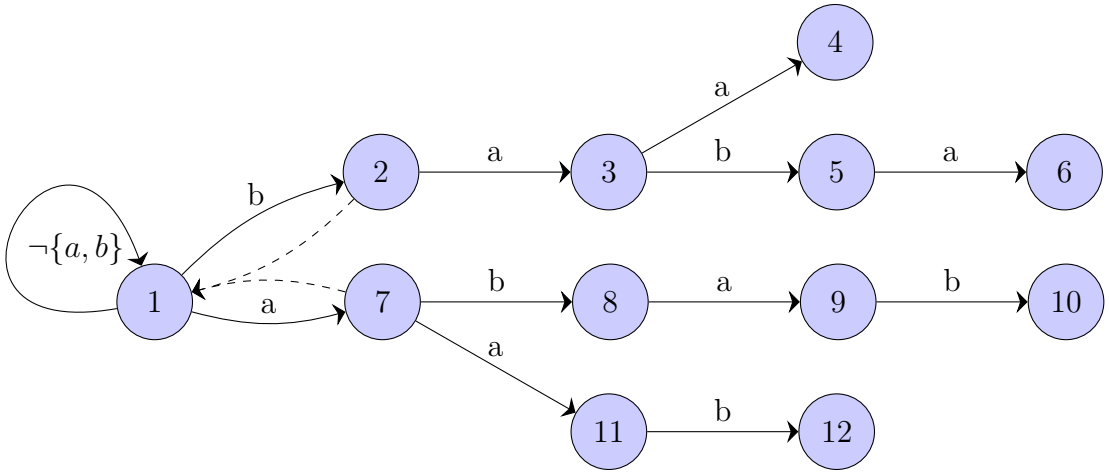

Figure 3.6: The point of Example 3.6 when $f(q)$ is defined for all states q with depth ≤ 1.

Example 3.6:

Let us continue with the previous example from the goto function construction. By definiton, $f(1)$ is not defined. For the states of depth 1 ($q \in \{2, 7\}$) the failure function is defined as $f(q) = 1$. Figure 3.6 presents this state.

Next we calculate the failure function for the states 3, 8 and 11, that is, the states with depth 2. For state 3, we take the parent state 2 and follow the failure path to the start state. Since $g(1,a) = 7$ we define $f(3) = 7$. For the states 8 and 11 the procedure is similar. We set $f(8) = 2$ and $f(11) = 7$. The procedure is iterated until the failure function is defined for every state. Figure 3.7 describes the final result. Because the AC-machine only contains these two functions, Figure 3.7 also presents the fully constructed AC-machine.

In Example 3.6 the value for the failure function was always found in a single iteration of the step 3 of the above definition (Figure 3.5). Next we go through the pseudocode for the failure function calculation in Algorithm 2.

Explanation of Algorithm 2. Since the failure values must be calculated from the lowest depth states to the highest depth states, we go through the goto transitions using a breadth-first search (BFS). This is provided by a queue that is appended in every non-leaf state, i.e., the children of the currently processed state are put into the queue and the next state to be processed is popped. The queue is initialized to be empty in the first line of the algorithm. The for loop on lines 3 to 5 adds all the children of the start state to the queue. Because the start state is special as it contains goto transitions to itself,

Algorithm 2 Aho-Corasick machine: Construction of the failure function

Input: Goto function g from algorithm 1

Output: Failure function f

```
1:  function CALCULATEFAILUREFUNCTION(g)
2:      queue ← empty
3:      for each σ s.t. g(1, σ) = q ≠ 1 do
4:          queue ← queue ∪ {q}
5:          f(q) ← 1
6:      while queue ≠ empty do
7:          let r be the next state in queue
8:          queue ← queue\{r}
9:          for each σ s.t. g(r, σ) = q is defined do
10:             queue ← queue ∪ {q}
11:             p ← f(r)
12:             while g(p, σ) is undefined do
13:                 p ← f(p)
14:             f(q) ← g(p, σ)
```

the additional constraint on line 3 is added to prevent an infinite loop. In the same for loop, we define $f(s) = 1$ for all states with depth 1. The while loop from line 6 to 14 implements the instructions in Figure 3.5. Since the AC-machine does not save parent information, this is done slightly differently, although the result is exactly the same as Figure 3.5 describes. On lines 7 and 8 we pop a state from the queue. At this point, the failure function for that state is already defined. Then we process each child of the state separately on lines 9 to 14 by 1) adding the child to the queue on line 10; 2) travelling the failure function of the parent as long as we get a state that have goto transition with the same symbol that appears between the parent and the currently processed child on lines 12 to 13; and 3) setting the failure function for the child on line 14. The execution of the function halts when the BFS of the AC-machine is ready and the queue is finally empty.

For now, as the AC-machine is fully defined we focus on its usage for finding the maximal pairwise overlaps of every input keyword, as Lemma 3.7 suggests.

Lemma 3.7. Let there be a state q and state $o, q \neq o$ in an AC-machine that are represented by strings s and t, respectively. The failure value $f(q) = o$ if and only if t is the longest proper suffix of s that is also a prefix of some keyword.

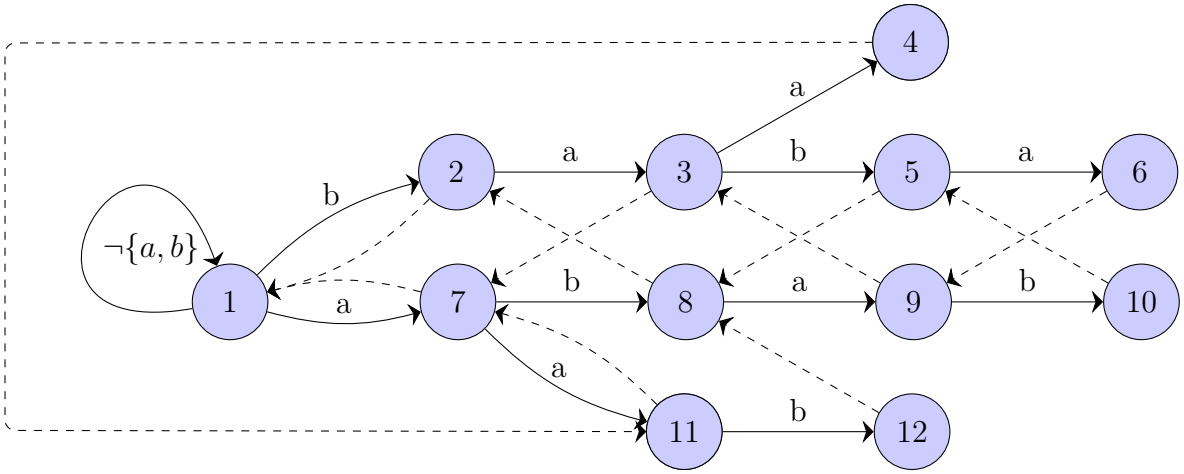

Figure 3.7: Fully constructed AC-machine for the keywords $R = \{$"baa", "baba", "abab", "aab"$\}$.

Proof (by induction). Let us have an induction hypothesis that the lemma is true for all states whose depth is smaller than some depth $d > 1$. The hypothesis trivially holds for the base case when the depth of the state is 1. That is, when $f(q) = o$ is the start state, represented by the empty string and the string represented by the state q has a length 1.

Let us assume there is a state q of depth d that is represented by a string $s = \sigma_1\sigma_2 \cdots \sigma_d$. Let r be the parent state of q. Therefore r is represented by a string $\sigma_1\sigma_2 \cdots \sigma_{d-1}$. Let $r_1, \cdots, r_n$ be a sequence of states such that:

1. $r_1 = f(r)$,
2. $r_i = f(r_{i-1})$ for all $1 \leq i < n$,
3. $g(r_i, \sigma_d)$ is undefined for all $i < n$,
4. $g(r_n, \sigma_d) = o$.

Since Algorithm 2 encodes the above rules on lines 12 to 14 the state o is defined for $f(q)$. Let the strings $u_1, ..., u_n$ represent the states $r_1, ..., r_n$, respectively. According to the induction hypothesis, u_1 is the longest (or equal longest) proper suffix of $\sigma_1\sigma_2 \cdots \sigma_{d-1}$ that is also a prefix of some keyword. Also, u_2 is a longest proper suffix of u_1 that is also a prefix of some keyword and so on. Therefore u_n is a longest proper suffix of s such that $u_n\sigma_d$ is also a prefix of some keyword. Since Algorithm 2 sets $f(q) = o$, the lemma is proven.

□

Data structures for the goto and failure functions. Lemma 3.7 is used later on in the description of Ukkonen's algorithm. Next, we look over the time complexities of Algorithms 1 and 2. First, we will investigate the time complexity of the insert and read operations of the functions g and f. The failure function maps each state (except the start state) to some other state. It is effective and practical to implement this as an array which stores the value $f(q) = r$ such that the array index q has a value r. Therefore, both inserting to and reading from the failure function is performed in constant time. The goto function maps a pair of values to a single value. In the same way, if we use a two-dimensional array which stores a $|\Sigma|$ sized row for each state in the AC-machine, the read and write operations of g would execute in constant time. This of course results in very high memory usage since the table would be sparsely used, especially with a large alphabet and long keywords. The goto function can be implemented in many other ways. One solution, which improves the memory efficiency, is to save a balanced binary tree for each state in the AC-machine. With balanced binary trees, the insert and read operations are performed in logarithmic time. Since each such binary tree contains at most $|\Sigma|$ elements, the time complexity of this approach is $O(\log(|\Sigma|)$.

Theorem 3.8. *The goto function for the AC-machine can be constructed in time $O(n \cdot t(g))$, where $t(g)$ is the time complexity of a single read/write query of the goto function.*

Proof sketch. The main function of Algorithm 1 consists of two for loops. The first loop calls a procedure once for each keyword of the input. The function calls are executed in a constant time, so the time complexity of the first for loop is linear with respect to the number of input keywords. The second for loop starting on line 5 is used to search all children of the start state. Since the start state has only $|\Sigma|$ goto transitions, the loop is executed at most $|\Sigma|$ times. In the worst-case, every symbol in every input keyword is different. In that case $|\Sigma| = n$, where n is the sum of the input symbols. Therefore, the second for loop is executed at most n times, resulting in the time complexity of the main function being linear with respect to the size of the input.

Let us next examine the *insert* procedure. The first two lines 8 and 9 can clearly be executed in constant time. The first while loop is used to find the symbol of the string that is not yet defined in the goto function. Let $s = uv$ be a string, such that u is the part of s that is already represented in the goto function. The index variable j is used so it contains the value of the first symbol of v after the while loop is executed. The for loop continues from j so the lines 13 to 16 are executed only for v. In the other words the *insert* procedure behaves such that every symbol of the input keyword is processed either in one

of the two loops but not both. Since the time complexities of the loop bodies depend on the implementation of g, the total time complexity of the function is $O(n) \cdot O(1) = O(n)$ or $O(n) \cdot O(\log(|\Sigma|)) = O(n \cdot \log(|\Sigma|))$ when the goto function can be used in constant or logarithmic time, respectively.

□

Theorem 3.9. *The time complexity of Algorithm 2 is the same as the time complexity of Algorithm 1.*

Proof sketch. In the discussion of Theorem 3.8 we reasoned that finding all children of the state can be done in linear time. In the failure calculation, we have the same kind of instruction in the for loop on line 9. Since this loop is executed for every state of the machine, this kind of approach leads us to $O(n|\Sigma|)$ time complexity for finding every child of every state. Instead of scanning the goto function for each symbol in the alphabet, we can use an auxiliary data structure to explicitly save the children of each state. One possibility is to use a linked list ($O(1)$ time insertions) and save the information of the children while creating the goto function. This adds $O(1)$ instructions to the for loop in the *insert* procedure. The body of this for loop is executed $O(n)$ times so the total time complexity is not changed. This allows us to get the children of a state in linear time with respect to the number of the children (and not the number of symbols in the alphabet). Since every state has at most one parent, the body of the for loop is executed $O(n)$ times in total even though it is nested with another while loop (line 6). Thus the while loop on line 6 as well as the for loop in line 9 are executed once for every state, that is $O(n)$ times. The lines 10 and 11 clearly take constant time. The time complexity of line 14 depends on the goto function implementation and can also be a constant time operation. Lastly, let us examine the inner while loop on line 12. The idea is to travel the failure path until a state is found such that the goto function for that state is defined with the same symbol as the goto transition is defined for state we are defining the failure function for. According to the definition of the failure function, the state $r = f(q)$ has depth at most one less than the depth of the state q. This means that the while loop is always executed at most d times where d is the depth of the state *state*. (Aho and Corasick, 1975) showed that this while loop is executed $O(n)$ times in total. Since all instructions in Algorithm 2 are executed at most $O(n)$ times, the proof is complete.

□

3.3 The Approximation Algorithm

In this section, we discuss the algorithms that use the AC-machine to find the pairwise overlaps of a set of input keywords to implement the previously defined greedy heuristic. Conceptually, we create an overlap graph of the keywords and select the edges according to the rules of Figure 3.5. In practice, the computation works in two steps. First we preprocess the AC-machine and create the needed auxiliary data structures in Algorithm 3 followed by Algorithm 4, which implements the greedy selection of the edges in the overlap graph.

By Lemma 3.7, we can find the longest proper suffix of any state that is also a prefix of some keyword by following the failure path of the state. In particular, let the starting state p itself be represented by some keyword (i.e., p is a leaf). Now the failure value $f(p)$ defines the state q that is represented by the longest proper suffix of p that is also a prefix of every keyword s_i such that q is an ancestor of s_i. Let us denote this kind of descendant relationship with a function L and call it a set of supporters.

Definition 3.10. $L(q)$ is a set of indices of the keyword set R such that states represented by $s_i \in R, i \in L(q)$ are descendants of the state q.

In other words, if a keyword index j is in $L(p)$, then the state represented by s_j is a descendant of the state p. For example, the set $L(3) = \{1, 2\}$ for the example AC-machine in Figure 3.7. Since the prefix relation is transitive, we can iterate over the failure function to find every pairwise overlap between p and the keywords. Algorithm 4 does exactly this. Lemma 3.11 formalizes this property.

Lemma 3.11. Let there be an AC-machine for a keyword set $R = \{t_1, ..., t_m\}$ and a state p represented by string s. There is an overlap of length $d(q)$ between s and a keyword $t_i \in R$ if and only if for some $k \geq 0$ state $q = f^k(p)$ is such that $i \in L(q)$.

Proof (by induction). Let us have an induction hypothesis that the lemma is true for all states whose depth is smaller than some depth $d' \geq 1$. For the base case of the induction, we use the start state with depth 0. The hypothesis holds with $k = 0$ since $L(start)$ contains every keyword index from 1 to m and there also exists an empty overlap (length $d(start) = 0$) between the empty string that the start state is represented by and all the keywords in R.

In the induction step, we assume there is a state $p, d(p) = d'$ represented by string s. Assume also that there is an overlap u of length $d(q)$ between s and some keyword t_i. The overlap u represents a state q. There are three possibilities:

1. The overlap u is the string s itself. In this case clearly $i \in L(p)$. Since $p = f^0(p)$, that is $p = q$ and $d(q) = d(p) = |s|$, the lemma is true.

2. The overlap u is the longest proper suffix of s that is also a prefix of t_i. By Lemma 3.7 this is true if and only if $f(p) = q$. In this case $i \in L(q)$ and the lemma is true with $k = 1$.

3. In this case, the overlap between s and t_i is shorter than the longest proper suffix of s that is a prefix of some keyword $t_j, i \neq j$. Let u' be the maximal overlap between s and t_j. That is, u' is the longest proper suffix of s that is also a prefix of some keyword. Let also q' be represented by the string u'. By Lemma 3.7 $f(p) = q'$. Because u and u' are both suffixes of s and $|u'| < |u|$ we know that there is also an overlap u between u' and t_i. Since $d(q') < d(p) = d'$, we can apply the induction hypothesis to the state q'. Therefore, there is an overlap of length $d(q)$ between q' and a keyword t_i (which we know is true) if and only if for some $k \geq 0$ state $q = f^k(q')$ is such that $i \in L(q)$. Since we know by assumption $q' = f(p)$ and by induction $q = f^k(q')$ it follows that $q = f^{k+1}(p)$. We also know that $i \in L(q)$ and $|u| = d(q)$. This concludes the proof.

□

Theorem 3.12. *Let p be a state represented by string s_i and let $s_i, s_j \in R$. The maximum overlap between s_i and s_j is represented by the state q such that q is the first state in the failure path of p $f^k(p)$ and $j \in L(q)$. The length of this maximum overlap is $d(q)$.*

Proof sketch. The proof directly follows from Lemmas 3.7 and 3.11. □

Implementation of the heuristic using the AC-machine. By Theorem 3.12 the AC-machine implicitly encodes the information of the longest pairwise overlaps between any two keywords. By Lemma 3.1 the lengths of the maximal overlaps encoded in the AC-machine are equivalent to the weights of the corresponding edges in the overlap graph. We use the property of Theorem 3.12 to implement the greedy heuristic for the Longest Hamiltonian Path problem in the overlap graph. The overlaps encoded by the failure

function have length equivalent to the depth of the state that have an incoming failure path. Hence, the longest possible overlap can be found by searching the state that has some incoming failure path or paths such that the depth of that state is maximized. This suggests the following implementation.

We process every state by following the failure path and saving which state the failure transition is coming from. The states with greater depth are processed first. This can be done with a reversed breadth-first search ordering. Since $d(p) > d(f(p))$ for every p, the reversed BFS ordering of the states ensures that when the state q is being processed, all the states p with $q = f^k(p)$ are already processed and we have the information of each such state. Since we have this information for every such state, we also know all the overlaps between $L(q)$ and each state q, s.t. q is in failure path from p. We can now select a corresponding edge in the overlap graph as long as it does not violate the constraints of the heuristic, that is, considering earlier selections the corresponding edge does not form a cycle, nor does its start vertex already have an outcoming edge, nor does its end vertex already have an incoming edge. By processing all the states in this fashion, we end up with the set of edges that forms a Hamiltonian path in the overlap graph.

Example 3.13:

Let us apply this method to the AC-machine described in Figure 3.7. The processing starts with depth 4 from state 6 or state 10. While processing those states, we save the information of the incoming failure path for state 9 and 5, respectively. Depending on the order in which we process the states with depth 3, we end up selecting one of the two possible overlaps. Either state 5 is processed first and the overlap between s_3 and s_2 is selected, or state 9 is processed first and the overlap between s_2 and s_3 is selected. The information of the incoming failure paths for states 3, 8 and 11 is also saved while processing the states of depth 3. Let us assume that state 9 was processed first and the second of the presented overlaps were selected. While processing state 3 we know the incoming failure path from state 6 (via state 9). This indicates that there exists an overlap between state 6 and every state in $L(3) = \{1, 2\}$ The self-overlap between s_2 and s_2 is forbidden since it would create a cycle. However, the other overlap between s_2 and s_1 cannot be selected either because the edge from s_2 is selected earlier. State 8 also has information of two incoming failure paths. Now there already exists an edge with s_3 as an end vertex. Thus, no edges are selected. When processing state 11 the only possible overlap is compatible with the rules and the edge from s_1 to s_4 is selected. For the states of depth 1 it does not matter which one is processed first. In any case, while

processing state 7 we find only one failure path coming from a leaf state corresponding to keyword s_2. Since s_2 already has an outgoing edge, the process continues without any selection. While processing state 2 we find two possible selections. We can select either (s_3, s_1) or (s_4, s_2). Depending on this last selection, the common superstring for the projection of the selected Hamiltonian path is either "baababab" or "bababaab", which both have the same length 8 and the same compression $14 - 8 = 6$.

When finding two equally long maximal overlaps, the heuristic allows arbitrary selections. Because of this, the outcome of the algorithm may differ between implementations. For example, the first time we made a selection between two allowed edges, if we selected differently the heuristics would have produced either "aababaa", "baababa" or "abababaab". These all are the actual shortest common superstrings with length and compression 7.

Description of Algorithm 3. The actual algorithm is implemented in two parts. Algorithm 3 defines the first part, which is used to calculate a set of auxiliary data structures that are needed in order to implement the actual heuristic. The algorithm takes as an input the AC-machine constructed with Algorithms 1 and 2 as well as the set of keywords R, which is also the same set that the goto and failure functions were constructed for. The listing in Figure 3.8 consists of the data structures that are calculated in algorithm 3. Most of them are used in algorithm 4. The listing uses I as a set of keyword indices in R.

1. $F : I \to Q, F(i) = q$ Maps the index of the keyword to the corresponding state. I.e., a state $q = F(i)$ is represented by the keyword s_i.

2. $E : Q \to I, E(q) = i$ Inverse function of F. If the state q is a leaf represented by s_i then $E(q) = i$. For every non-leaf states q' we define $E(q') = 0$.

3. $d : Q \to Q$ Depth function for each state as defined before.

4. $L : Q \to 2^I$ Set of supporters as defined before.

5. $b : Q \to Q, b(q) = r$ Reverse breadth first search ordering of the states.

6. B The last state in the BFS ordering of the machine. I.e., $b(B)$ is the second last state in BFS ordering, $b(b(B))$ is the third last and so on.

Figure 3.8: Static data structures created in Algorithm 3

Since by Lemma 2.11 a shortest common superstring for a reduced set of keywords is the same as the set where it was reduced from, Algorithm 3 also performs the elimination of those keywords that are substrings of other keywords in the set. There are two situations when the structure of the AC-machine indicates the occurrence of such a keyword. First, if there is a goto transition from a state that is represented by a keyword, that keyword must be a substring of some other keyword. That is, if $g(F(i), \sigma)$ is defined for any pair (i, σ) then the keyword s_i must be a prefix (i.e., substring) of some other keyword and can be removed. The second indication of such a keyword is when a leaf state has an incoming failure transition from some other state. By Lemma 3.7, if $F(i) = f(q)$ then the keyword s_i must be a proper suffix of some other state. This means that s_i is either a suffix (if the state q is itself a leaf) or a substring that is not a prefix nor suffix of some other keyword. The removal of a substring is done using the function F. If some keyword s_i is a substring of another keyword, we set $F(i) = 1$ to indicate that the state represented by s_i is not processed as if it was a keyword.

Explanation of Algorithm 3. Algorithm 3 proceeds in two independent steps that could be processed in either order. In the first part, we loop through every keyword (line 2) and every symbol in the keywords (line 5). Before we enter the for loop on line 5 we set the state q to the start state. In the loop we traverse the AC-machine from the start state to the state represented by the current keyword s_i. On the way, we add the index i to every state we pass through (line 7). Processing every keyword in this way constructs the function L as defined. After processing each keyword (if line 8 evaluates to *true*) we set the values for functions F and E as defined (lines 9 and 10). Further, we set $F(i) = 1$ for those keyword indices that correspond to a keyword that is a prefix of some other keyword (lines 11, 12).

The second part of this algorithm calculates the reversed breadth-first ordering b for the AC-machine and set the depths for every state. In addition, substring keywords other than prefixes are removed. This part is basically a BFS traversal through the machine. We create a queue and initialize it with the start state 1 on line 13 and set the depth of the start state to zero (line 14). The variable B, initialized on line 15, contains the value of the latest state that is already processed, thus at the end of the algorithm the variable B contains the last state in the BFS ordering (which is the first state in the reversed BFS ordering). The actual BFS starts on lines 16 to 18 when we start to process the states in the queue. When all states are processed, the while loop ends and the function terminates. In the for loop starting at line 19 we add the children of the current state to the queue as

Algorithm 3 Ukkonen's algorithm: Preprocessing

Input: Outputs of Algorithms 1 and 2, the associated set of keywords $R = \{s_1, ..., s_m\}$

Output: F, E, d, L, b and B as defined earlier

Preconditions: Initially $E(q) = 0$ for every state q

```
1:  function CALCULATEAUXILIARYFUNCTIONS(g, f, R)
2:      for i = 1, ..., m do
3:          let s_i = σ_0, ..., σ_{k-1}
4:          q ← 1
5:          for j = 0, ..., k − 1 do
6:              q ← g(q, σ_j)
7:              L(q) ← L(q) · {i}  *
8:              if j = k − 1 then
9:                  F(i) ← q
10:                 E(q) ← i
11:                 if q is not a leaf of the AC machine then
12:                     F(i) ← 1
13:     queue ← 1
14:     d(1) ← 0
15:     B ← 1
16:     while queue ≠ empty do
17:         let r be the next state in queue
18:         queue ← queue\{r}
19:         for each q that is a child of r and q ≠ r do  †
20:             queue ← queue · q
21:             d(q) ← d(r) + 1
22:             b(q) ← B
23:             F(E(f(q))) ← 1
```

the BFS requires. On line 21 we set the depth of the current state to be one greater than its parent state. We also update the B variable as discussed before. Finally, line 23 sets $F(i) = 1$ for each keyword index i such that the state represented by s_i has an incoming failure transition from any other state.

*In the original algorithm this line was $L(q) \leftarrow L(q) \cdot \{j\}$.

†To prevent infinite recursion the inequality $q \neq r$ was added.

Line 12 together with line 23 ensure that for all keywords i such that s_i is a substring of some other keyword, the value of $F(i) = 1$, i.e., the set of keywords $\{s_j : F(j) \neq 1\}$ is reduced.

Theorem 3.14. *The time complexity of Algorithm 3 is the same as time the complexity of Algorithm 1.*

Proof sketch. As with the construction of the AC-machine, the time complexity of querying the goto function depends on its underlying implementation. At first, we ignore the queries for the goto function and focus on the rest. In the first part of the algorithm, we iterate over all keywords at the outer for loop. In the inner for loop we iterate over each symbol in the keyword so the total number of iterations in the inner loop equals the total number of symbols in all the keywords combined. On line 7 we add an element to a set of items $L(q)$. Since this set does not need to be ordered, it can be implemented for example as a linked list so the time complexity for adding an item is $O(1)$. Assuming, each state has a flag indicating whether it is a leaf, also the statements on lines 8 to 12 can be performed in constant time. This kind of flag can be easily added in Algorithm 1. This means that the time complexity of the outer for loop is $O(n \cdot t(x))$ where n is the total number of symbols in the input and $t(x)$ is the time complexity of querying the goto function.

The lines 13 to 15 can clearly be executed in constant time. The while loop starting at line 16 does the breadth first search for the AC-machine. Since the number of states in the machine is $O(n)$, the execution time for the BFS is also $O(n)$. Here we assume that the queue is also implemented using a structure that enables insertions and deletions in a constant time (e.g., a linked list). Lines 21 to 23, neither of which are related to the BFS search itself, can also clearly be executed at constant time. This results in the total time complexity of Algorithm 3 being $O(n \cdot t(x))$. For example, if the goto function is implemented using direct indexing, the reading and writing of g is performed in constant time, thus the time complexity is $O(n)$. On the other hand, if g is implemented using for example balanced binary trees, then the access time of g is $O(\log(|\Sigma|))$ and the total time complexity of the algorithm is $O(n \cdot \log(|\Sigma|))$.

□

Description of Algorithm 4. Let us now discuss the algorithm that implements the greedy selection of the edges in the overlap graph to form an approximate longest Hamiltonian path in more detail. Earlier, we simulated the greedy heuristic without defining the

appropriate data structures. Now, with Algorithm 3 we have augmented the AC-machine with a set of auxiliary functions and one variable, listed in Figure 3.8. This auxiliary data is static in the following Algorithm 4, i.e., the data is only read and not written at any point. In addition, we need some dynamic data to store the information related to the selection of the edges in the overlap graph. As we discussed in the simulation of the heuristic, when the execution enters a state q, all other states with depth $> d(q)$ are already processed and there is information about every $F(i)$ that is a starting point of any failure path to q, such that there is no already selected edge (s_i, s') for any keyword s'. Let us denote this information with p.

Definition 3.15. At the point of the execution of Algorithm 4 when a state q is being processed, $p(q)$ is a set of keyword indices i such that each state $F(i), i \in p(q)$ is a starting point of a failure path to q, i.e., $q = f^k(F(p(q))) : k > 0$ and there is no edge (s_i, s') already selected for any $s' \in R$.

When we encounter a state q with $i \in P(q)$ we know that there exists an overlap of length $d(q)$ between s_i and every keyword whose corresponding state is accessible in the AC-machine through the state q, i.e., that state is descendant of q. We have already defined a data structure L for this information, hence there is a maximal overlap of length d between every s_i and every s_j, where $i \in P(q)$ and $j \in L(q)$. Note that although the dynamic data structure P ensures that every keyword $s_i, i \in P(q)$ can be selected as a start vertex of the new edge, the $L(q)$ returns static data. If there are already selected edges $(s', s_j), j \in L(q)$ for each j and for any s', no more edges can be selected while processing q. A constraint, that is to prevent a vertex having multiple ingoing edges, is encoded in the function $forbidden(i)$ where i is an index of a keyword.

Definition 3.16. At any point of execution of Algorithm 4

$$forbidden(i) = \begin{cases} true, & \text{if } (s', s_j) \in H \text{ for any } s' \in R; \\ true, & \text{if } s_j \text{ is a substring of any other string in } R; \\ false, & \text{otherwise.} \end{cases}$$

We encode the constraints that no vertex can have multiple inbound or outbound edges with the functions L, P and $forbidden$. The last constraint, which prevents the edge selections that form a cycle, is guarded with functions $first$ and $last$. Both of these functions are defined for all keyword indices. At the beginning of Algorithm 4, $first(i) = last(i) = i$ for every keyword index i. The value of these functions is updated such that at

any point of execution if we have selected a set of edges that forms a path starting from s_i and ending to s_j, then $first(j) = i$ and $last(i) = j$. Note that the values for the vertices in the middle of such paths are not updated nor queried. Now, before the new edge (s_i, s_j) is selected, we check that the selection does form a cycle by checking that $first(i) \neq j$. Note that this is the same thing as checking that $last(j) \neq i$. Only one of those checks is necessary. If the check fails, the edge in question must be discarded.

Explanation of Algorithm 4. Algorithm 4 starts by setting the initial values for the dynamic functions $P, first, last$ and $forbidden$ in the for loop starting at line 2. On lines 4 to 6 the algorithm initializes the data that is related to the used keywords, that is the keywords that form a reduced set. Line 4 sets the P function for each state that has a direct failure link from some of those keywords. For those same indices, the functions $first(i)$ and $last(i)$ are initialized to i, indicating there are no edges yet selected. The else clause on lines 7 to 8 is to ensure that there will be no incoming edges to the keywords that are substrings of other keywords. (Note: setting $F(j) = 1$ for the same keywords in algorithm 3 ensures the prevention of outcoming edges from those strings). After the initialization is done, the first state to be processed is set on line 9. Note that setting $q = B$ would cause one more unnecessary iteration of the following while loop since the last state in the BFS ordering can not have incoming failure transitions, hence the first state is set to the second last state in that ordering.

The while loop starting at line 10 implements the actual selection of the edges. Since deepest states are processed first, and nonemptiness of $P(q)$ and $L(q)$ indicates an existing overlap of depth $d(q)$, the maximal overlaps are selected first, if allowed by the constraints. On line 11 we ensure that $P(q)$ is not empty, since otherwise there are no allowed overlaps. After that, each keyword index in $L(q)$ that is not forbidden is processed (line 12). Each iteration (except the last one if no allowed edge is found) of that for loop decreases the size of $P(q)$ and $\{i \in L(q) : forbidden(i) = false\}$. On lines 13 to 14 we check the set $P(q)$ to ensure there is at least one possible start point to the new edge. After line 15 there is a potential new edge that needs to be verified against the cyclicality constraint (on line 16). If the edge (s_i, s_j) would create a cycle, we check on line 17 if there exists another index in $P(q)$ and, if so, the edge is created. Otherwise the execution jumps to the next iteration. Note that if the edge (s_i, s_j) would create a cycle, then no other edge (s', s_j) will for any s'. Therefore it $P(q)$ only needs to be queried at most twice with each iteration.

Algorithm 4 Ukkonen's algorithm: Selection of the edges

Input: Outputs of previous Algorithms, the set of keywords $R = \{s_1, ..., s_m\}$

Output: The approximate longest Hamiltonian path H in the overlap graph

```
1: function createPath(f, F, L, b, B, R)
2:     for j = 1, ..., m do
3:         if F(j) ≠ 1 then
4:             P(f(F(j))) ← P(f(F(j))) · j
5:             first(j) ← j
6:             last(j) ← j
7:         else
8:             forbidden(j) ← true
9:     q ← b(B)
10:    while q ≠ 1 do
11:        if P(q) is not empty then
12:            for each j in L(q) s.t. forbidden(j) = false do
13:                if P(q) is empty then *
14:                    break
15:                i ← the first element of P(q)
16:                if first(i) = j) then
17:                    if P(q) has only one element then
18:                        continue
19:                    else
20:                        i ← the second element of P(q)
21:                H ← H · {(s_i, s_j)}
22:                forbidden(j) ← true
23:                P(q) ← P(q)\{i}
24:                first(last(j)) ← first(i)
25:                last(first(i)) ← last(j)
26:            P(f(q)) ← P(f(q)) · P(q)
27:        q ← b(q)
```

On line 21 the edge is selected. Line 22 ensures the definition for the $forbidden$ function remains valid and lines 24 and 25 update the $first$ and $last$ functions accordingly. On line

*This line is not present in the original algorithm.

23 the selected keyword index is removed from $P(q)$. After all possible edges starting from $s_i, i \in P(q)$ are examined, the process can move on to the next state in the b link chain (line 27). Before that, on line 26, the $P(q)$ is concatenated to $P(f(q))$ so the possibly remaining unused keyword indices are passed to the shallower states.

Before we go into the time complexity of Algorithm 4, let us first discuss the total length of supporters sets $|L(q)|$ over all states q. Although $L(q)$ may contain (partly or completely) the same indices as $L(r), q \neq r$, the sum $\sum_q |L(q)|$ is $O(n)$. Specifically, $\sum_q |L(q)| = n$ for a reduced set of input keywords and $\sum_q |L(q)| < n$ otherwise, when the start state is excluded.

By iterating through all supporters of all states, we effectively iterate through all symbols in the input. This is because, for each state q, each supporter i in $L(q)$ corresponds to a symbol in the ith input string, and every symbol corresponds to exactly one state and supporter.

Lemma 3.17. For a reduced set of keywords, the sum $\sum_q |L(q)|$ over each state q, except the start state, is n, where n is the total number of symbols in the input keywords.

Proof (by induction). Let us have an induction hypothesis that the lemma is true for all AC-machines constructed from at most M keywords. The hypothesis trivially holds for AC-machine with a single keyword. That is, there are $|s|+1$ states, where s is the keyword, and $|L(q)| = 1$ for each $|s|$ states, producing $\sum_q |L(q)| = |s| = n$.

Let there be an AC-machine AC constructed from a reduced set of keywords of size M and a corresponding supporters set function L. Let t be a keyword not presented in AC. When t is added to AC, a path from start state to a leaf is branched at depth d such that $t[1...d]$ is the longest prefix of t that has a corresponding state in AC. The $L(r)$ for each new state r consists of only the index of the keyword t. The sets of supporters $L(q)$, for each existing state q are appended with the index of the keyword t.

In other words, the sets of supporters are increased by one for each $|t|$ states between the start state and the newly created leaf. According to the induction hypothesis $\sum_q |L(q)| = |s| = n$.

□

Theorem 3.18. *The time complexity of Algorithm 4 is $O(n)$.*

Proof sketch. As with L in Algorithm 3 we assume that the set returned by function P is implemented using a linked list, allowing constant time insertions, deletions, and

concatenations. The first for loop starting at line 2 can clearly be executed in linear time since all the statements take constant time and $m \leq n$. The while loop starting at line 10 iterates once for each state that is $O(n)$ times. The for loop starting at line 12 dominates the time complexity of Algorithm 4. Since the execution of one iteration of the for loop is a constant time operation and by Lemma 3.17 $\sum_q |L(q)| = O(n)$, which is the total number the for loop is iterated, the total time complexity of Algorithm 4 is $O(n)$.

□

Theorems 3.8, 3.9, 3.14 show that Algorithms 1, 2 and 3 have the same time complexity depending on the implementation of the goto function for the AC-machine. Moreover, Algorithm 4 runs lin linear time. As discussed earlier, the goto function can be implemented to allow read and write queries in $O(1)$ using direct indexing over a two-dimensional array. Another feasible possibility, especially for the larger alphabets, is to use balanced binary trees with $O(\log(|\Sigma|))$ time complexity. This conclusion leads us to the following theorem.

Theorem 3.19. *With Algorithms 1, 2, 3 and 4 the greedy heuristic for the approximate longest Hamiltonian path in the overlap graph can be implemented in time* $O(n \cdot t(x))$ *where* $t(x)$ *is the time complexity of a single read/write query of the goto function.*

4 Dictionary Compression: The LZ Family

Dictionary compression algorithms (Storer and Szymanski, 1982) process the input as a sequence of symbols. Using the structure and repetitiveness of the input, longer input sequences are encoded as shorter tokens. This reduces the space needed to store the original input data. In this approach, a so-called dictionary is maintained. The dictionary may be static or it can be updated during the (de)compression process. The tokens (also called phrases) contain pointers to the dictionary such that each token unambigiously corresponds to a sequence of symbols (Pu, 2005).

In this chapter we discuss the Lempel-Ziv family (Ziv and Lempel, 1977), (Ziv and Lempel, 1978), (Welch, 1984) of dictionary compression methods. First we discuss the Lempel-Ziv77 (LZ77 for short) algorithm (Ziv and Lempel, 1977) as an introduction to the relatively new algorithm called Relative Lempel-Ziv (Kuruppu et al., 2010) (RLZ for short). We show a few examples of these algorithms and discuss the possible improvement for RLZ dictionary construction with use of a shortest common superstring approximation algorithm.

4.1 Lempel-Ziv77

Lempel-Ziv77 is an adaptive dictionary compression method. By adaptive, we mean that during the compression the dictionary is updated depending on the input. At first, the dictionary is empty. The original LZ77 algorithm uses a part of the input, called a sliding window, as a dictionary. However, we introduce a version of LZ77 that has an unbounded window size, allowing the phrases to point to any position of the input that is already processed, i.e., the dictionary consists of the already processed part of the input.

Each phrase in LZ77 is either

1. a literal phrase, or
2. a repeat phrase.

Literal phrases are used when the current symbol in the input file has not been seen before in the file. Phrases consist of a pointer and a length, where the pointer defines the point in the already processed text where the repeating sequence of symbols starts. To use a repeat phrase, the text up to the pointer must already be tokenized. The length encodes the size of the original text corresponding to the repeat phrase. The more repetitive the input text is, the bigger the length values in repeat phrases are. This means a single phrase encodes a longer portion of the original text, resulting in better compression. On the other hand, if the length value is small enough, a single repeat phrase may need more space than the original text as is. For example, if the repeat phrase is encoded using two 4-byte integers and the input symbols are encoded in a single byte, the repeat phrase with a length of less than 8 would actually consume more space than the original text.

Example 4.1:

Let us simulate the LZ77 algorithm for a text string $T =$ "abbabaabbaba". We denote literal phrases with $[\sigma]$, where σ is the symbol of the phrase. Repeat phrases are denoted with (position, length). The processing starts at the first symbol of T. Since the symbol "a" is encountered for the first time, we create a literal phrase ["a"]. The same thing happens with the next symbol "b". The third symbol "b" is encoded with a repeat phrase $(2, 1)$ which means that the phrase encodes a string of length 1 starting at position 2. The next repeat phase $(1, 2)$ encodes the string "ab". At this point, we have encoded the string "abbab". The next phrase encodes the following "a" with a phrase $(1, 1)$. Since the text string encoded to this point contains a string "abbab" the last phrase is a pointer to the start of this string with a length of 6. The final LZ77 factorization is therefore ["a"]["b"]$(2, 1)(1, 2)(1, 1)(1, 6)$.

To decompress the LZ77 parse, we process the phrases from the first phrase to the last phrase. For each phrase, using the already decompressed data, we augment the dictionary with the text corresponding to the phrase. After the decompression, the generated dictionary is the same as the dictionary used with the compression. In other words, the resulting decompression dictionary is the same sequence of symbols as the original uncompressed data.

4.2 Relative Lempel-Ziv

Relative Lempel-Ziv is a simple and efficient (Deorowicz and Grabowski, 2011) general-purpose (Hoobin et al., 2011) compression method. It combines a static dictionary with an LZ77 parsing, providing semi random access to the compressed data. It was developed to compress a set of genomes, however, further research has shown that RLZ scales well with any repetitive data, providing one of the best (de)compression time and compression ratios (Gagie et al., 2016).

To compress a collection of genomes of the same species, RLZ works as follows. One genome is selected as a base (or reference) sequence. This genome acts as a static dictionary. Further, an LZ77-like parse of every other sequence is generated relative to the base. The LZ77 parses of the other sequences are restricted to have references only to the base sequence. As opposed to the adaptive dictionary of the LZ77 algorithm, where the dictionary is constructed during the (de)compression, RLZ uses a static dictionary, providing random access for any phrase whether or not the decompression of the previous part of the data is already performed. Definition 4.2 formalizes the Relative Lempel-Ziv parsing (or factorization).

Definition 4.2. Let T and R be strings. Relative Lempel-Ziv factorization of T relative to R, denoted by RLZ$(T|R)$ is a factoriszation $T = w_1 w_2 \cdots w_n$, where for each $1 \leq i \leq n$, w_i is either

1. a symbol σ that does not occur in R, or
2. the longest prefix of $T[j...|T|]$, where $j = |w_1 \cdots w_{i-1}|$ that is a substring of R.

Each factor w_i is encoded as a pair (p_i, l_i) where p_i is the index of R where w_i starts from and l_i is the length of w_i.

Example 4.3:

	1	2	3	4	5	6	7
$R =$	a	b	b	a	b	a	a

Let $T =$ “abbababaaaaabbabaa” and $R =$“abbabaa” be strings. The RLZ factorization of T relative to R is RLZ$(T|R) = (1, 6)(5, 3)(6, 2)(1, 7)$.

As discussed above, RLZ can be used as a general-purpose compressor. With a collection of genomes of the same species, one genome is a natural choice for a dictionary that other

genomes are factorized against. Choosing the dictionary for arbitrary data (not necessarily a collection of the same kind) is however more difficult since there is no natural reference. As the number of factors highly depends on the contents of the dictionary, this decision is however essential. Fortunately, it has been shown that even with a lack of a natural reference, an effective artificial one can be easily constructed. (Hoobin et al., 2011) has shown that by taking random samples of the data and concatenating them together, an efficient artificial reference can be constructed. The achieved compression ratios are surprisingly good with dictionaries as small as 0.1% of the original input size. Further (Gagie et al., 2016) have given the theoretical analysis of why this method works so well.

Our hypothesis is that RLZ compression with general data can be further improved using the sampling method described in (Hoobin et al., 2011). The idea is to eliminate redundancy in the artificially constructed dictionary. One simple way of achieving this is to use a shortest common superstring approximation to compress the dictionary. In principle, depending on the repetitiveness of the data, this method should decrease the dictionary size while keeping the number of factors roughly the same, thus improving the compression. Another possibility is to use a compressed dictionary that is the same size as the conventionally constructed dictionary. Theoretically, the RLZ factorization against this kind of dictionary should result in a decreased number of factors, hence improving compression.

This hypothesis has been tested and verified in this thesis. The results presented in Section 5.4 show that the number of factors is drastically decreased when the dictionary for several highly repetitive corpuses are constructed using the common superstring approach. Also, runtimes for dictionary construction remain low when a linear time approximation algorithm for the SCS problem is used.

5 Empirical Evaluation

In this chapter, we introduce and describe an open-source implementation of Ukkonen's shortest common superstring approximation algorithm, discussed in Chapter 3. We go through the most significant aspects of the implementation and usage of the code. We also describe the data that was used to benchmark the efficiency of the implementation as well as the experiments related to relative Lempel-Ziv compression using the SCS of the sampled original data as the reference string.

Finally, we present experimental results that compare the implementation with another implementation of the same heuristic. We also show how the number of factors in the RLZ compression is reduced when the SCS approach is used.

5.1 Implementation

Overview: The implementation is written in the C programming language without any external libraries. Only the C standard library and POSIX libraries were used. Although the algorithms presented in Chapter 3 are relatively short, the actual implementation also contains other code that binds Algorithms 1, 2, 3 and 4 together. Various data structures were also implemented for the algorithm. In total, the implementation consists of 2400 lines of code, including the comments and empty lines. The implementation also includes some test scripts for building and testing the code.

As discussed earlier, the goto function of the AC-machine can be implemented in many different ways. In our implementation, the underlying data structure for the goto function is defined at compile time to be either a two-dimensional array or a red-black tree (Cormen, 2009). Hence, by Theorem 3.19 the time complexity of our implementation of Ukkonen's algorithm is either $O(n)$ or $O(n \cdot \log(|\Sigma|))$, respectively.

Since the definition of the input of the algorithm is a set of keywords, we have to guard against that constraint too. In practice, that means we need to examine the input keywords and remove any possible duplicate values. The duplicate removal is implemented by sorting

the input keywords and removing any consecutive duplicate strings. The keywords are sorted using string quicksort (Bentley and Sedgewick, 1997). Therefore the overall time complexity of our software is $O(n \cdot \log(n))$. At the time of writing, the duplicates are forced to be searched. There is no command line parameter that would omit the $O(n \cdot \log(n))$ time string sorting if it were already known that no duplicates were present. However, as we see later even with the $O(n)$ implementation of Ukkonen's algorithm the $O(n \cdot \log(n))$ time preprocessing is insignificant even with relatively large inputs.

Our implementation uses the `char` datatype of the C programming language as the type of the element in an integer alphabet that is used in the algorithm. The `char` datatype is an obvious choice, since the C standard defines many string-processing functions for `char` arrays. However, since `char` is only 8 bits long, this choice limits our implementation for alphabets Σ with $|\Sigma| \leq 256$. The code is however designed so that the alphabet data type is bound to a macro. Thus, the change in the data type would be as easy as possible if the software is further developed to work with bigger alphabets.

The git repository of our implementation can be found from GitHub:
https://github.com/apason/ukkonen-90

Compile-time parameters: To optimize the memory consumption and the execution time depending on the input data, various alternative approaches were implemented. The purpose was to include various compile or runtime flags that would change the behavior of the application. All of those modifications except one were discarded. The current version includes a compile-time macro `OPTIMIZE_LINKS` that changes the data structure that is used to the children of each state. Later on, we refer to this parameter as the *links optimization*. The optimization works as follows. Each state must be able to refer to all of its children in time linear with respect to the number of children. Each child of a state is saved in a list. When links optimization is turned on, this list is implemented as an array and allocated to have as many elements as there are symbols in the alphabet. Since the list has a fixed size, there will be only one memory allocation for each state. The drawback of this feature is the increased memory consumption with large alphabets. When links optimization is turned off, the list containing the children of a state is implemented using a linked list. A new node to the list is allocated for each child, thus the overhead from the memory management is relatively larger. In practice, the links optimization is viable only with small alphabets and can decrease the execution time and the memory consumption with larger alphabets. As we will see later, in general, the best performance for big alpha-

bets is achieved with the combination of red-black trees and disabled links optimization. On the contrary, when the alphabet is small enough, direct indexing over two-dimensional arrays combined with enabled links optimization results in the best execution time.

The input keywords are given to the program as an input file that contains one keyword per line. The line separator is an ASCII line feed character (code 10). This adds a limitation to the alphabet, since in practice the decimal value 10 cannot be a part of any given keyword. In Section 5.1 we introduce another input system that allows any character code (except 0) to be used. In that input method the keywords are merely sampled from the input file and the individual keywords are not defined. When the input keywords are read, there is a symbol buffer where the individual lines are first read to. No input keyword can be longer than this buffer. The size of the buffer defaults to 2048 symbols but can be modified in compile time by setting the desired value for the macro `MAX_LINE`. Note that the `MAX_LINE` macro defines the size of the buffer, not the size of a maximum keyword, and that the keywords are read to the buffer with the newline character (that is considered not part of the keyword) and that the char arrays must be terminated with the `NULL` character, the actual maximum keyword size is `MAX_LINE` -2.

Since the value of the depth function d can not exceed the length of the longest keyword(s) that the AC-machine encodes, we can define the datatype for the codomain of d depending on the maximum keyword length. The default data type is an 8-bit positive integer, hence by default, up to 255-length keywords are supported. When a keyword longer than 255 is used, the data type for the codomain of the depth function must be larger. By defining the compile-time macro `LONG_KEYS` the depth function uses a 16-bit datatype instead. This allows the maximum keyword length to be $2^{16} - 1$ which should be enough for the RLZ dictionary construction.

The underlying data type that encodes the numbers of states defaults to a 4-byte unsigned integer. That means the maximum number of states that the AC-machine can encode is 2^{32} which is approximately 4 billion. Since the number of states cannot be known beforehand, the fact that the number of states is at most the number of symbols in the input is useful. As we use a one-byte alphabet, the size of the input file should not exceed 4GiB when using the default configuration. The data type for the state numbers can be changed as follows. There is a macro `MAX_STATE` that automatically defines the data type to be as small as possible. Possible data types are 8, 16, 32 and 64-bit unsigned integers. Note that it is not necessary to set the `MAX_STATES` macro to a smaller value, even if the input size were less than 2^{16} bytes. This only affects the memory consumption of the program,

which is anyway quite small with input sizes less than 16KiB.

There are also two compile-time options that affect the execution. If the macro `SCS` is not defined, the program only executes Algorithms 1 and 2 creating a plain AC-machine for the input keywords. This option was added in order to compare the construction of the AC-machine to another similar implementation (Salmela et al., 2006). When using the program for approximate shortest common superstring calculation, the `SCS` macro must always be defined. The last compile-time option is the `INFO` macro. When `INFO` is defined, additional information of the calculation is printed along with the approximate shortest common superstring. The additional information contains some used data types, the number of removed duplicates in the input keywords, the number of different symbols in the input (real alphabet size), the number of states and a table that lists the most important phases of the execution and their time usages. Without the `INFO` macro the program outputs only the approximate SCS for the input.

The following table summarizes the available compile-time options.

Macro name	**Possible values**	**Default**	**Notes**
ARRAY_GOTO	defined / undefined	N/A	Either one but not both of this macro
RB_TREE_GOTO	defined / undefined	N/A	or this macro must be defined.
OPTIMIZE_LINKS	defined / undefined	undefined	Enables the links optimization.
MAX_LINE	Any positive integer	2048	The size of the keyword buffer.
LONG_KEYS	defined / undefined	undefined	Sets max depth to 65536.
MAX_STATES	Any positive integer	2^{32}	Should be as small as possible.
SCS	defined / undefined	undefined	Enables an approx. SCS calculation.
INFO	defined / undefined	undefined	Prints additional information if defined.

Compilation: The compilation and the usage of our implementation requires a C compiler and the GNU Make software. Since the program uses `<sys/resource.h>` and `<sys/time.h>` POSIX header files, the program can only be used in a POSIX compliant (e.g., Linux, BSD, macOS) operating system providing these libraries. There are no other requirements for building and running the binary. The default C compiler is GCC and it is defined in the file `common.mk`.

There is a top-level Makefile in the root of the repository and other Makefiles in the `tests/` and `src/` folders. Every action can be executed in the repository root, since the top-level Makefile further calls the Makefiles in these subfolders. The default action triggers the

project build. When the build is done, the `target/scs` binary is created and ready to be executed. To build the binary, either the `ARRAY_GOTO` or `RB_TREE_GOTO` macro must be defined. There is no default value for the goto function's data structure. Other macros can be defined if needed. The Makefile in the `src/` folder, that builds the binary, is configured to append every compilation command with an environment variable named `DEFS`. This is a convenient way of passing macro definitions to the compiler. For example, to compile our implementation to calculate the shortest common superstring approximation that uses the red-black trees, one may issue the following command.

```
user@host:.../ukkonen-90/$ make DEFS="-DRB_TREE_GOTO -DSCS"
```

To compile the binary with support of keywords of length 1000 and direct indexing with links optimization enabled and additional information printed, one may issue the following command.

```
user@host:.../ukkonen-90/$ make DEFS="-DARRAY_GOTO -DOPTIMIZE_LINKS -DSCS \
                           -DLONG_KEYS -DMAX_LINE=1002 -DINFO"
```

The Makefile also contains jobs called `clean`, `clobber` and `delete_tests`. The `clean` job removes all intermediate object files. The `clobber` does the same and also removes the executable in the `target/` folder. The `delete_tests` target removes the test instances in the `tests/` folder. After that, the next time the tests are executed the new random test instances are also generated.

Usage: When the binary is ready to use, there are two methods for executing it. The simplest method is to execute the binary with a single input filename. The file should contain a single byte encoded text with line feed (ASCII 10) delimited keywords. The file should also end with a line feed. Note that the input can be any single byte encoded text with the following constraints:

1. The input keywords must not contain a byte of value 10, since it is reserved for delimiting the keywords. Any keyword that contains line feed values are treated as multiple keywords.

2. The input file must not contain zero values.

It does not matter whether the text is ASCII, extended ASCII, some single byte ISO-encoding or any custom-made single byte encoding. The following example demonstrates the usage with single input file as a parameter. In the example, the binary is compiled with `SCS` macro enabled and `INFO` macro disabled.

```
user@host:.../ukkonen-90/$ cat input
baa
baba
abab
aab
user@host:.../ukkonen-90/$ target/scs input
aababaa
```

The other way of using our implementation is with command line options that are associated with flags. The flag `-f` defines the input file. The flag `-a` and `-b` defines the input types. Only one of the input types can be specified at a time. The `-a` flag defines the same kind of input method we discussed earlier, thus the following command is equivalent to the earlier example:

```
user@host:.../ukkonen-90/$ target/scs -a -f input
```

The input type specified with the `-b` flag requires two more options and the behavior of the program is different. In this case the input file is considered as a source file where the keywords are randomly sampled as defined by `-l` (line length) and `-c` (cut) parameters. The sampling of the keywords is performed as follows. The `-l` parameter defines the length of a single keyword (all keywords have the same length). The `-c` parameter defines how many lines are sampled. The cut parameter takes a floating point value between 0 and 1. The total number of sampled input keywords multiplied by the length of a single keyword must equal (the value can be rounded) the original input file size multiplied by the cut parameter. For example, if the input file is 10KB and the `-l` parameter has a value of 100 and the cut parameter has a value of 0.50, then the keywords are sampled such that there are 50 keywords of length 100 (that is $10\text{KB} \cdot 0.50 = 5\text{KB}$). Note that when the keywords are sampled from rather than delimited in the input file, they can also contain the line feed characters. After the keywords are sampled, it is possible (especially with highly repetitive input files and big cuts) that there are multiple identical keywords. Before the SCS algorithm is executed, the duplicates are first removed. The following example

demonstrates the usage with an input file that is sampled to get a set of keywords of lengths 128 such that the size of the sampled keywords equals 10% of the size of the given input file.

```
user@host:.../ukkonen-90/$ target/scs -b -f input -l 128 -c 0.1
```

Tests: Our implementation has been tested to be correct with three different sets of tests. The tests are located in the same Git repository as the actual implementation. There is also a helper script for generating random test instances and scripts that run the actual tests. All test-related files are in the `tests/` folder of the root of the git repository. To run the tests just issue

```
user@host:.../ukkonen-90/$ make tests
```

in the root folder (of the repository). This invokes the Makefile in the `tests/` folder. If the test instances have not yet been created, the script `create_instance.sh` is executed and two test folders are generated. Note that depending on your hardware, the generation of the test instances can be quite slow (up to 30 min). The first test folder contains random instances for alphanumeric, binary, DNA and hexadecimal alphabets, with different number and length of keywords. This test set is used to test that the generated common superstring actually contains every keyword in the test file. This test is executed with the `test1.sh` shell script file. The `grep` utility is used to ensure that each keyword is present in the output of our implementation.

The same set of tests is also used with the `test3.sh` file. In this test we compile two versions of our implementation, one which uses the direct indexing and another that uses the red-black tree implementation. The tests instances are then run with each version of the implementation and for each test, the test is passed if the generated superstrings are equal. In other words, the idea of this test is to ensure that the data structure that is used with the goto function does not affect the outcome.

The second set of test instances is rather small. There are only few very small instances with binary alphabet. The test script `test2.sh` runs these tests as follows. First, each test instance is executed with a python script `scs.py` that uses the `pysat` python library to calculate the length of a longest common superstring of the instance. The same instance is then run with our implementation and the compression values for both are calculated.

Finally, the compression is compared and the test is considered to be passed if the compression acquired with our implementation is at least half of the optimal compression. The idea of this test set is to make sure that the proven quality of the compression holds.

Note that in order to run all the tests, the python environment with the pysat library must be configured. Note also that the make format uses GNU specific features that require GNU Make or GNU Make compliant makefile software.

5.2 Benchmark Data

Here we describe the two datasets that are use in Section 5.4. We discuss the origin of both of those datasets and inspect their basic properties. Since the data is slightly manipulated to fit our experiments, we also describe this preprocessing.

PizzaChili datasets: PizzaChili (Ferragina and Navarro, 2005) is a deprecated project of University of Pisa and University of Chile, whose purpose is to share publicly available corpuses and related full-text indices for others to experiment with. The dataset defined here consists of some of the repetitive text collections from the PizzaChili project. We refer to this dataset as the PizzaChili dataset. There are seven files of repetitive text from real (as opposed to artificial) origin in the PizzaChili dataset. The following listing gives a short description of each of those files. Table 5.1 includes some basic information of the files. The inverse match probability is defined as the inverse of the probability that two randomly chosen characters from the file are the same. This is also referred to the effective alphabet size. With uniformly distributed data, the effective and actual alphabet sizes are exactly the same.

1. The **cere** file consists of 17 concatenated sequences of Saccharomyces Cerevisiae. The origin of this file is the Saccharomyces Genome Resequencing Project.

2. The **coreutils** file consists of 9 versions of the coreutils 5.x source code files.

3. The **einstein.de.txt** file consists of all German wikipedia versions of Albert Einstein up to 2010-01-12.

4. The **einstein.en.txt** file consists of all English wikipedia versions of Albert Einstein up to 2006-10-10.

File	Size(MiB)	Alphabet size	Inverse match probability
cere	440	5	4.301
coreutils	196	236	19.553
einstein.de.txt	89	117	19.264
einstein.en.txt	446	139	19.501
influenza	148	15	3.845
kernel	247	160	23.078
para	410	5	4.096

Figure 5.1: Basic information of the PizzaChili files. (Ferragina and Navarro, 2005)

5. The **influenza** file consists of 78,041 sequences of Haemophilus Influenzae. The origin of this file is the National Center for Biotechnology Information.

6. The **kernel** file consists of 36 versions of the Linux kernel source files of versions 1.0.x and 1.1.x.

7. The **para** file consists of 36 concatenated sequences of DNA of Saccharyomyces Paradoxus. The origin of this file is the Saccharomyces Genome Resequencing Project.

Preprocessing of PizzaChili: Each file in the dataset is preprocessed as follows. First, the newline characters are substituted with tab characters (ASCII key code 9). After that, a number of new files are sampled from the files such that the new file consists of multiple lines delimited by a newline character. Each line has a length l and the total (rounded) number of characters, excluding the newlines, are $0.01c \cdot s$, where s is the size of the original file. In other words, the c parameter describes the percentage of the size of the original file. The parameters c and l have the following value combinations: (15, 128), (15, 254), (25, 254), (50, 254), (50, 512), (75, 512), thus at total six sampled files are generated. The sampling is performed by selecting a random position between the beginning of the file and the end of the file subtracted by l and copying a string of length l from that position to the sample file. This process is then repeated until the wanted file size is achieved. Any duplicate lines are removed and the process is repeated until the wanted number of lines is reached. At the end of this process we have a sample file that contains unique keywords of length l such that the size of all keywords combined is $c\%$ of the original size. The files are named `s_c_l`. For example, the file `cere_50_128` has size 220MB (plus the number of newlines) and contains unique keywords of length 128.

The DNA dataset: The second dataset is called the DNA dataset. The origin of this data is the Sequence Read Archive of National Library of Medicine. The data consists of 3.8 million reads of escherichia coli, 100 symbols of each. The alphabet $\Sigma = \{A, C, G, T, N\}$ size of the original file is 5. The 'N' characters were relatively rare in this file (3506 out of 3849544 lines contain at least one 'N'). Since the PizzaChili dataset does not have a file with alphabet size less than 5, the DNA dataset was modified by removing all lines containing the 'N' character, resulting in a file with alphabet of size 4. Further, the duplicates were removed from the data and the data was randomly sampled to 11 different files as follows. The first file was from the main file such that the number of lines was 3125, thus the first file has a size of $3125 + 3125 \cdot 100 = 315625$ bytes. The first file was named `1_dna_3125.acgt`. The process was repeated for the last ten files such that the number of lines for each file was two times the number of lines in the previous file. The same naming convention was used with the exception that each consecutive set of three zeros was substituted with the letter 'k'. The first character in the file name is a hexadecimal digit of the number of the file. For example, the third file was named to `3_dna_12500.acgt`, the fourth file was named to `4_dna_25k.acgt` and the last (11th) file was named to `B_dna_3200k.acgt`

5.3 Benchmark Setups

The experiments were done in two different environments. In this section, we describe the relevant aspects of the software and hardware for both of them. We also name the environments so they can be referred to in Section 5.4. The first environment is called Haapa. It is the more powerful of the environments, providing, due to its larger amount of memory, a possibility to run much bigger problem instances. The second environment is referred to as Env2. For running our experiments, Haapa is however the slower of the two environments. Although Haapa has many times more CPUs, the single CPU is significantly slower. Since the code used in the experiments are all single-threaded, the performance of a single core is all that matters. The memory clock speed in Haapa is also configured to be significantly slower, thus limiting the performance of our implementation with huge memory requirements. Fewer experiments are run on Env2 since the memory of the machine provided a limitation to the size of the problem instances. Table 5.1 describes the main features of both environments.

	Haapa	**Env2**
Operating system	Ubuntu Linux 20.04 LTS	Gentoo Linux *
Linux kernel version	5.11.0	5.4.80
GCC version	9.4.0	10.2.0
Glibc version	2.31	2.32
GNU Time version	1.7	1.9
Processor model	Intel Xeon E7-4830 v3	Intel Core i7-6700K
Processor clock speed	2.1 GHz	4.0 GHz
Processor cache size	30 MiB	8 MiB
Memory density	1.5 TiB	32 GiB
Configured memory clock speed	1333 MHz	2133 MHz

Table 5.1: Software and hardware specifications of the environments Haapa and Env2.

5.4 Results

In this section, we examine the performance of our implementation. As a reference, we use another implementation of the same greedy heuristic for the approximate shortest common superstring problem. The *reference implementation* (Alanko and Norri, 2017) performs the computation in a space efficient way using only $O(n \cdot \log(|\Sigma|))$ bytes of memory. The time complexity of the reference implementation is also $O(n \cdot \log(|\Sigma|))$. In contrast, our implementation of Ukkonen's algorithm works in linear time when the code is compiled to use direct indexing for the operations with the goto function. Our implementation is also able to use red-black trees for the goto operations. Both of these approaches were tested. In addition, as discussed in Section 5.1, our implementation has an additional compile-time parameter that defines the data structure used for the children list of the states. As it is, there are four different versions of our implementation, at total:

1. Direct indexing ($O(n)$) with links optimization enabled.
2. Dicect indexing ($O(n)$) with links optimization disabled.
3. Red-black tree ($O(n \cdot \log(|\Sigma|))$) with links optimization enabled.
4. Red-black tree ($O(n \cdot \log(|\Sigma|))$) with links optimization disabled.

*Gentoo Linux is a so-called rolling release distribution. That means it does not have any version numbers.

	cere	**einstein.en.txt**
75_512	0.36%	3.54%
50_512	0.33%	3.06%
50_254	0.71%	4.87%
25_254	0.59%	4.04%
15_254	0.54%	3.25%
15_128	1.06%	4.77%

Table 5.2: Worst-case time consumptions in percentage for the input preprocessing.

For convenience, we refer to these versions as DIE, DID, RBE and RBD, respectively. All of the results, from our implementations as well as the reference implementation, presented in this section are from binaries that have been compiled with the same C compiler (gcc) with the same optimization settings. The predefined optimization set `-O3` have been used as well as the linking time optimization `-flto`. The following process was executed with all presented time benchmarks. Each experiment was repeated five times. For the five execution times we picked the median value. After that, if there were any values with at least 5 percent difference, those values were removed and the new median value was picked. If there was an even number of values left, the bigger one of the two middle values were chosen. The idea behind this procedure was to eliminate the possible outliers from the results. Most often there were no outliers and the presented value was acquired immediately. For the PizzaChili files with keyword lengths ≤ 254 there were 31/700 discarded values when time usages were measured in Haapa.

Preprocessing time: As discussed in Section 5.1, our program uses the $O(n \cdot \log(n))$ string quicksort algorithm to remove the possible duplicate input keywords. By compiling our program with the `INFO` macro defined, we get, among other things, an analysis of the time spent in different phases of the program. Referring to Tables 5.3, 5.4, 5.5 and 5.6, the RBD implementation was the fastest one to process any sampled `einstein.en.txt` file. Correspondingly, the DIE version was the fastest one for any `cere` file. In Table 5.2 we show the time portion of our implementations (DIE for `cere` and RBD for `einstein.en.txt`) that were used for the duplicate removal as a percentage of the total runtime. Since the used implementation versions are the fastest ones for the particular instances, the time portions for the duplicate removals are maximal. Table 5.2 shows the worst-case time percentages for the preprocessing phase.

Investigating Table 5.2, a few observations can be made.

1. With the same line length, increasing the input size increases the time portion required to preprocess the input. This is expected since the preprocessing part of the program works in $O(n \cdot \log(n))$ time, which is asymptotically more than the implementation of the greedy heuristic.

2. With the same input size, an increase in the keyword length causes the relative preprocessing time to decrease.

As we see, the maximum relative preprocessing time for the cere files is 1.06% of the total runtime. The results were similar for all instances having small (< 12) alphabets. Based on this data, it is reasonable to say that the preprocessing time of the program is insignificant for instances with a small alphabet, while the instance sizes are in the ranges of our experiment. For bigger alphabets ($|\Sigma| > 12$) the relative preprocessing time is greater. However, in our experiments, the time required for the preprocessing with large alphabets was at most around 5 percent. One may interpret this value to be significant, however while comparing our implementation with large inputs to the reference implementation, the differences with the relative time usages are usually significantly more than 5%.

Performance comparisons: The performance of our implementation and the reference implementation were compared. The acquired time usages were measured using the GNU time software. For the PizzaChili dataset, all files with line lengths of 128, 254 and 512 were investigated. This experiment was run in Haapa. For the DNA dataset, the comparisons were run for each file. This experiment was also run in Env2 for all but the biggest problem instance, since the memory of Env2 was not sufficient for that.

The runtimes for the PizzaChili dataset files with line length ≤ 254 are specified in Tables 5.3, 5.4, 5.5 and 5.6. Each table shows the runtimes for the set of each 7 files sampled with the same parameters. The File column is the prefix of the used filename. For example, the files in Table 5.3 are `cere_15_128`, `coreutils_15_128` and so on. The column labeled Reference shows the runtimes with the reference implementation. Columns labeled with DIE, DID, RBE and RBD show the runtimes of our implementation versions. The best runtimes for each input file are bolded. The last two columns are discussed later.

File	Reference	DIE	DID	RBE	RBD	Comp.	Rsize
cere	146.00	**91.65**	100.44	115.76	120.89	0.319	0.0478
coreutils	73.23	105.04	74.24	104.80	**51.27**	0.463	0.0694
einstein.de.txt	37.76	29.11	23.95	29.79	**18.13**	0.0463	0.00694
einstein.en.txt	246.17	134.09	110.58	135.21	**85.12**	0.0269	0.00403
influenza	40.62	29.97	**29.79**	34.35	32.92	0.338	0.0507
kernel	114.56	109.74	85.58	111.53	**65.13**	0.244	0.0366
para	122.66	**86.50**	94.77	110.11	116.21	0.387	0.0580

Table 5.3: Runtimes in seconds for PizzaChili dataset with cut 0.15 and keyword length 128, compression, relative size of the dictionary. The fastest runtimes are bolded.

File	Reference	DIE	DID	RBE	RBD	Comp.	Rsize
cere	144.06	**102.97**	113.97	129.63	136.11	0.409	0.0614
coreutils	69.29	112.36	78.83	107.45	**55.21**	0.527	0.0791
einstein.de.txt	39.44	30.27	24.44	30.05	**18.54**	0.0652	0.0098
einstein.en.txt	278.75	154.52	129.48	154.96	**102.88**	0.0268	0.00402
influenza	38.75	33.03	**32.76**	38.23	37.28	0.463	0.0695
kernel	113.62	117.58	91.70	117.96	**71.77**	0.269	0.0403
para	119.23	**97.25**	107.02	123.52	129.05	0.445	0.0668

Table 5.4: Runtimes in seconds for PizzaChili dataset with cut 0.15 and keyword length 254, compression, relative size of the dictionary. The fastest runtimes are bolded.

File	Reference	DIE	DID	RBE	RBD	Comp.	Rsize
cere	294.24	**175.33**	191.33	216.54	221.55	0.316	0.0790
coreutils	145.94	187.77	131.09	180.38	**95.82**	0.426	0.106
einstein.de.txt	73.96	52.18	42.56	52.80	**33.40**	0.0442	0.0110
einstein.en.txt	483.52	244.33	202.77	246.41	**159.36**	0.0210	0.00524
influenza	80.12	**57.89**	59.75	65.41	63.83	0.410	0.103
kernel	235.27	194.60	154.91	198.07	**118.92**	0.182	0.0455
para	253.59	**163.89**	179.08	208.02	214.48	0.348	0.0870

Table 5.5: Runtimes in seconds for PizzaChili dataset with cut 0.25 and keyword length 254, compression, relative size of the dictionary. The fastest runtimes are bolded.

File	Reference	DIE	DID	RBE	RBD	Comp.	Rsize
cere	715.62	**350.59**	384.95	431.81	439.19	0.213	0.106
coreutils	377.27	370.95	261.72	358.27	**188.88**	0.289	0.145
einstein.de.txt	171.43	99.20	83.85	101.56	**67.74**	0.0255	0.0127
einstein.en.txt	993.62	463.47	371.48	456.26	**294.33**	0.0157	0.00786
influenza	197.47	**118.44**	120.73	129.16	128.45	0.335	0.167
kernel	569.85	392.67	306.08	392.72	**241.98**	0.104	0.0519
para	629.26	**336.03**	361.32	403.99	424.42	0.237	0.119

Table 5.6: Runtimes in seconds for PizzaChili dataset with cut 0.50 and keyword length 254, compression, relative size of the dictionary. The fastest runtimes are bolded.

As discussed in Section 5.1, the links optimization was designed to improve the performance of our implementation with small alphabets. In general, the approach of direct indexing over a two-dimensional array is only reasonable with small alphabets, since it significantly increases the memory consumption of the program. Also, in practice, our implementations with red-black trees were faster with big alphabets. Investigating the runtimes from Tables 5.3, 5.4, 5.5 and 5.6 we make the following observation.

1. The DIE version was the fastest for all instances with small (< 12) alphabets.
2. The RBD version was the fastest for all instances with big (> 12) alphabets.
3. The DID version was slightly ($< 1\%$) faster than the DIE version for the files `influensa_128_15` and `influensa_128_25` ($|\Sigma| = 12$).
4. The RBE version was the fastest for none of the instances.

As expected, the DID and RBE versions are generally not competitive with the DIE and RBD versions. Later on, we only present the results for either DIE (when alphabet size ≤ 12) or RBD (when alphabet size > 12) versions of our implementation.

Let us now examine the runtimes more carefully for the instances with keyword lengths 254. Table 5.9 illustrates the runtimes for each instance in PizzaChili dataset with keyword length 254. Thus, each plot uses the data from Tables 5.4, 5.5 and 5.6. Note that since the data points are not uniformly distributed in the x-axis, these graphs do not illustrate the time complexities of the programs. The x marked blue lines and circle marked black lines show the time usages of the reference implementation and our implementation, respectively. Runtimes in seconds are presented in the left vertical axis. As we see,

File	Reference	Our impl.	Compression	Rsize
cere	707.99	**436.36**	0.299	0.149
coreutils	362.85	**227.42**	0.332	0.166
einstein.de.txt	184.57	**80.48**	0.0385	0.0192
einstein.en.txt	1147.92	**392.66**	0.0181	0.00903
influenza	185.32	**147.45**	0.457	0.228
kernel	567.09	**281.81**	0.123	0.0613
para	616.93	**417.35**	0.296	0.148

Table 5.7: Runtimes in seconds for PizzaChili dataset with cut 0.50 and keyword length 512, compression, relative size of the dictionary. The fastest runtimes are bolded.

the time usage of the reference implementation grows more rapidly in every case. This relative change in runtimes is illustrated with the orange triangle marked lines. The values are presented as a percentage of the time usage of our implementation compared to the time usages of the reference implementation. The percentage values are shown on the right vertical axis. This data suggests that our implementation outperforms the reference implementation with big enough inputs. However, investigating Tables 5.3 and 5.4 we also see that increasing the line length while keeping the instance size constant affects the runtimes contrarily. For every file, the time usage of our implementation increases when the line length is increased from 128 to 254 while keeping the cut in 15 percent. With the reference implementation, the time usage grows only with the einstein files. Moreover, the growth is relatively smaller than the growth with our implementation. By this data, we assume that there are probably instances similarly sampled from PizzaChili data for which the reference implementation is faster.

Tables 5.7 and 5.8 show similar data for the PizzaChili dataset with line lengths 512. These results are consistent with the runtimes of the smaller instances. That is, our implementation compared to the reference implementation gets faster when the size of the instances grows while the keyword length remains constant. Also, the relative speed of our implementation versus the reference implementation decreases while the input size remains constant and the line length increases.

Let us now examine the runtime benchmarks for the DNA dataset. This experiment was run in both environments. The results are shown in Table 5.10. The first column of the table, labeled with # defines the prefix of the filename. For example, the row starting with

File	Reference	Our impl.	Compression	Rsize
cere	1162.88	**665.73**	0.237	0.178
coreutils	637.61	**335.40**	0.256	0.192
einstein.de.txt	303.83	**126.08**	0.0277	0.021
einstein.en.txt	1744.57	**560.52**	0.0146	0.011
influenza	324.12	**221.69**	0.399	0.299
kernel	946.69	**432.82**	0.0881	0.066
para	1033.05	**634.22**	0.237	0.178

Table 5.8: Runtimes in seconds for PizzaChili dataset with cut 0.75 and keyword length 512, compression, relative size of the dictionary. The fastest runtimes are bolded.

5 corresponds to the file `5_dna_5k.acgt`, which consists of 5000 lines of DNA sequences, 100 bases in each line.

The columns labeled Reference and DIE show the runtimes in seconds. Due to the high memory usage of our implementation, the last file could not be run in Env2. For both environments, there is also a Percentage column, which shows the runtimes of our implementation in percentages compared to the runtimes of the reference implementation. The last column contains the compression ratio of the generated superstring, that is, the size of the superstring divided by the size of the input instance. Note that the achieved compression is a result of the greedy heuristic, which is used by both of the implementations. Hence, the compression is the same to the number of significant digits represented in the table.

From the data, we can observe that the runtimes are significantly higher in Haapa. This result is expected, since Haapa has an older CPU with lower clock speed. The memory speed in Haapa is also lower. With the smaller instances, the reference implementation has lower runtimes. In Haapa, the reference implementation is faster with instance sizes $\leq$ 200,000 lines. In Env2 this threshold is 100,000 lines. The corresponding instance sizes in mebibytes are 20 and 10, respectively. After that, our implementation dominates.

To further illustrate runtimes, we also sampled a set of instances with linearly increasing size from 100,000 lines to 2,000,000 lines in 100,000-line steps. This set was then run in Env2. Every instance was run only once and no precise values are shown. The results are shown in Figure 5.2 (the image description is same with the images in Table 5.9). Since this image has a linear scale, the time complexities of the implementations can be seen.

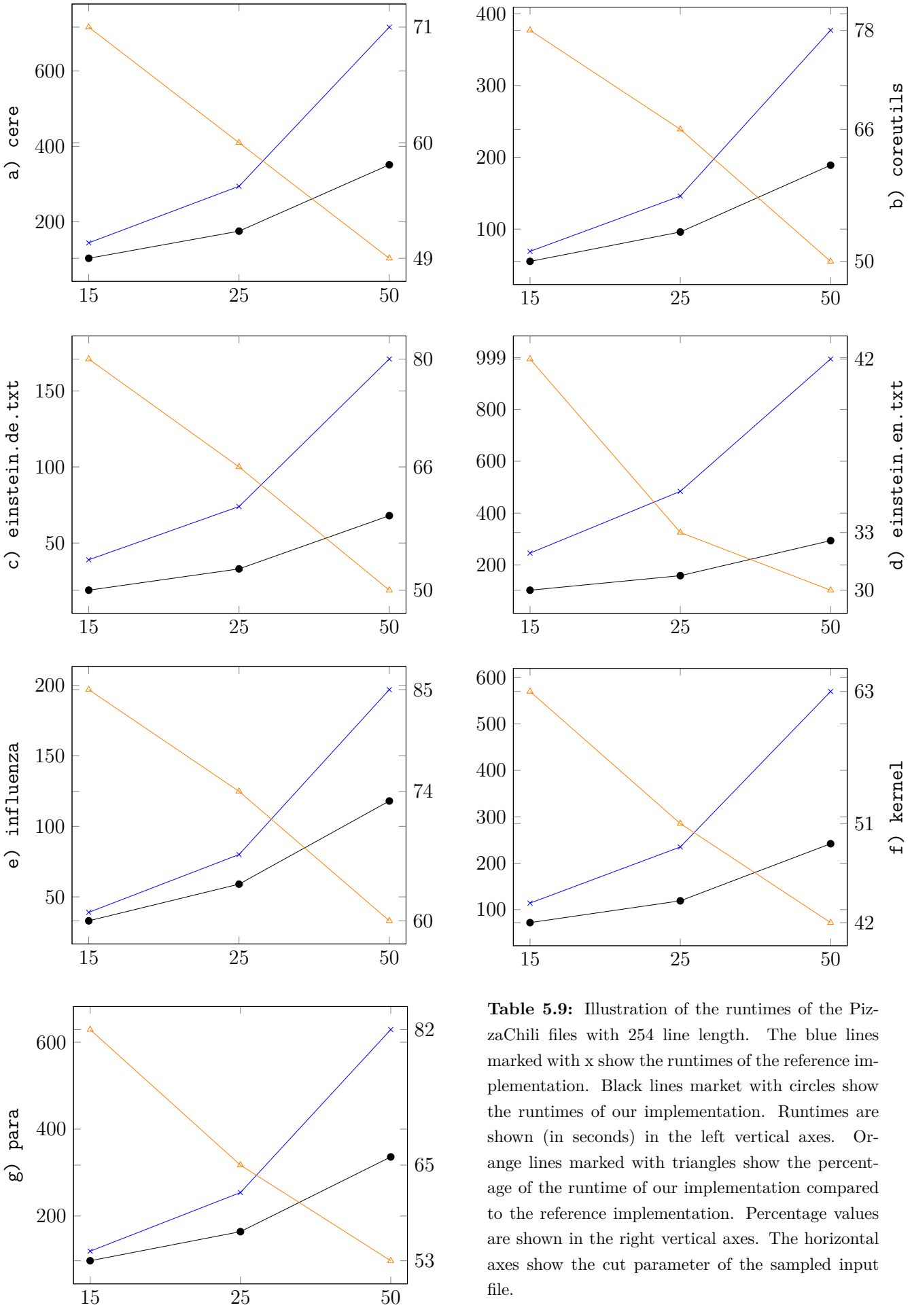

Table 5.9: Illustration of the runtimes of the PizzaChili files with 254 line length. The blue lines marked with x show the runtimes of the reference implementation. Black lines market with circles show the runtimes of our implementation. Runtimes are shown (in seconds) in the left vertical axes. Orange lines marked with triangles show the percentage of the runtime of our implementation compared to the reference implementation. Percentage values are shown in the right vertical axes. The horizontal axes show the cut parameter of the sampled input file.

	Haapa			Env2			
#	**Reference**	**DIE**	**Percentage**	**Reference**	**DIE**	**Percentage**	**Comp.**
1	0.24	**0.21**	87.5	**0.09**	**0.09**	100.0	0.927
2	**0.45**	**0.45**	100.0	**0.19**	0.21	110.5	0.909
3	**0.88**	0.97	119.2	**0.39**	0.5	128.2	0.887
4	**1.83**	1.94	106.0	**0.88**	1.22	138.6	0.840
5	**4.06**	4.65	114.5	**2.03**	2.85	140.4	0.769
6	**9.56**	10.98	114.9	**5.46**	6.25	114.5	0.660
7	**23.45**	24.53	104.6	16.12	**13.14**	81.51	0.512
8	62.02	**51.07**	82.3	44.63	**26.47**	59.4	0.362
9	167.34	**102.38**	61.6	107.72	**51.92**	48.2	0.254
A	412.7	**202.41**	49.0	247.37	**104.78**	42.4	0.191
B	953.86	**406.2**	42.6				0.155

Table 5.10: Runtimes in seconds for the DNA dataset, the runtime of our implementation compared to the reference implementation (in percentages) in both environments, compression (the size of the common superstring divided to the size of the input). The fastest runtimes for each environment are bolded.

Comparing the runtimes on the PizzaChili and the DNA datasets we can conclude that our solution implements the greedy heuristic faster:

1. When the instance size is big enough,
2. Especially with shorter input keywords and
3. With smaller alphabet sizes.

RLZ factorization. As discussed in Chapter 4 the size of the RLZ compressed file is the size of the reference string used as a dictionary plus the size of the factors referencing to this dictionary. Let us now discuss the compression obtained by the RLZ algorithm with two different methods for generating the dictionary. The first method that we use for the dictionary construction simply samples substrings of fixed length from random positions of the file that is to be compressed. The samples are then concatenated and the factorization is run against that dictionary. The second approach has not been studied before. The idea is to randomly sample a number of substrings, as before but instead of concatenation, the SCS approximation algorithm is run and the result is used as the reference dictionary.

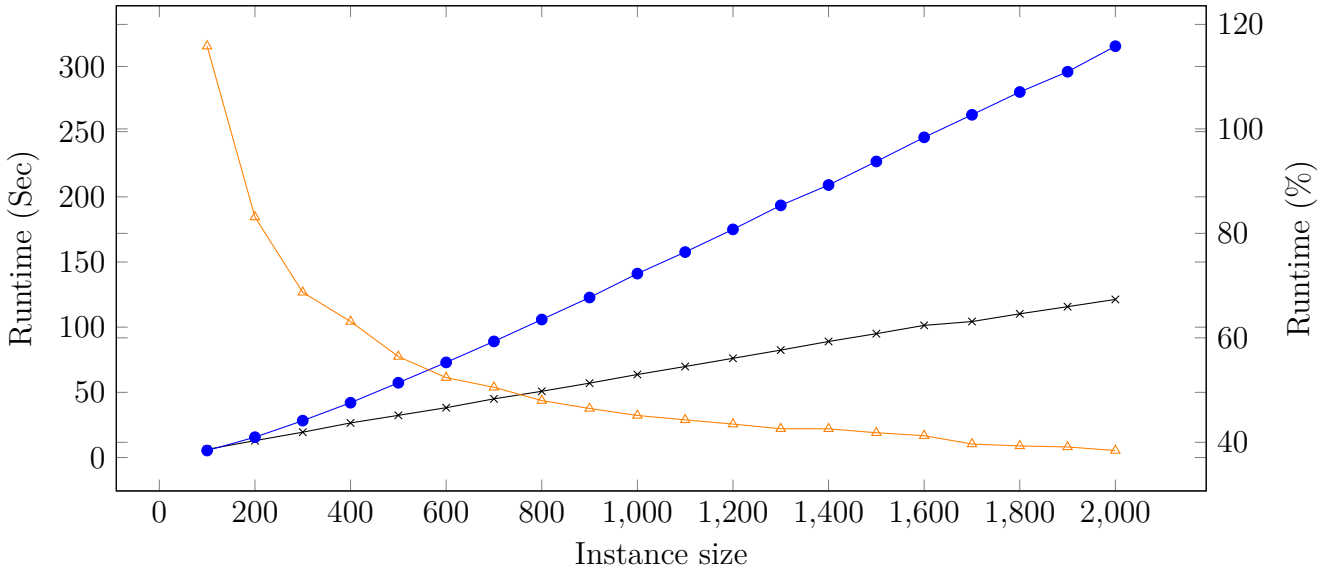

Figure 5.2: Comparison of the runtimes between our and reference implementation with DNA data consisting of keywords of length 100. The blue line marked with x shows the runtimes of the reference implementation. The black line marked with circles shows the runtimes of our implementation. Runtimes are shown (in seconds) in the left vertical axis. The orange line marked with triangles shows the percentage of the runtime of our implementation compared to the reference implementation. Percentage values are shown in the right vertical axis. The x-axis specifies the size of the instance in thousands of lines.

The PizzaChili dataset sampled with line lengths from 128 to 512 and cut parameters from 15% to 75% were used. The Compression column in Tables 5.3, 5.4, 5.5, 5.6, 5.7 and 5.8 show the size of the generated superstring relative to the instance size. This compression value depends on the size of the instance (the cut parameter). The Rsize value is calculated from the superstring compression value to get the size of the superstring relative to the original file. In other words, when the common superstring is used as an RLZ dictionary the Rsize value multiplied with the size of the original file to be compressed, is the size of the dictionary. As we see, the compressions are better with bigger instances. However, increasing the keyword length while keeping the instance size constant, the compression ratio gets worse.

We ran the RLZ factorization for every PizzaChili file using the common superstrings from the instances sampled with various parameters as a dictionary. The upper part of Table 5.11 shows the number of RLZ factors generated using the approximate SCS as a dictionary. For reference, the lower part of the table shows the number of factors when the dictionary is generated by simply concatenating the samples so that the lengths of the dictionaries are the same. We see that the SCS method for the dictionary construction results in a significantly smaller number of factors. The number of factors with the SCS

File	15_128	15_254	25_254	50_254	50_512	75_512
cere	2,730,732	2,135,129	1,574,261	1,233,553	816,328	686,684
coreutils	3,348,186	3,052,924	1,677,551	641,487	511,471	289,703
einstein.de.txt	300,857	168,361	163,409	154,105	83,080	85,206
einstein.en.txt	1,546,254	780,202	779,287	844,213	418,429	403,249
influenza	1,443,648	1,169,337	964,633	740,696	580,713	467,416
kernel	948,673	736,718	391,851	235,094	168,852	131,755
para	3,829,215	3,205,234	2,080,952	1,220,678	961,559	700,598
cere	10,986,309	6,900,933	5,264,601	3,930,879	2,148,565	1,904,918
coreutils	6,591,442	5,580,636	4,525,247	3,548,432	2,812,876	2,387,512
einstein.de.txt	1,456,740	800,626	737,333	683,388	379,491	361,252
einstein.en.txt	5,191,299	3,443,373	2,949,429	2,592,922	1,557,116	1,445,208
influenza	2,349,774	1,701,867	1,519,536	1,309,105	942,168	844,213
kernel	8,001,635	6,478,714	5,708,406	4,938,226	3,530,819	3,217,852
para	11,264,145	8,445,457	6,628,010	5,086,768	3,403,923	2,940,037

Table 5.11: The number of factors in relative Lempel-Ziv compression. The upper numbers result when the RLZ dictionary is a common superstring of the corresponding file. The common superstring is generated from the files randomly sampled with parameters specified in the column title. The lower numbers result when the dictionary is a string obtained by concatenating random samples together such that both dictionaries have the same length.

method ranges from 63.5% (`influenza_25_254`)to 4.1% (`kernel_75_512`) of the number of factors generated when using a concatenated dictionary.

This suggests that the acquired RLZ compression is significantly better when the SCS approach is used. However, the number of factors is only a part of the compression. The final compression ratio also depends on the size of the dictionaries, which are the same in both cases. Let us assume that each factor in the compressed file consists of two 4-byte integers. Note that in practice, the factors could be stored with less space. For example, in many cases the generated superstring is shorter than 24Mi symbols, making 3 bytes sufficient for factor length and position index. For simplicity, we now use 4 byte integers.

Table 5.12 shows the final sizes of the RLZ compression for these two methods. The value in each cell is a sum of the corresponding dictionary and the number of factors presented in Table 5.11 multiplied with 8 bytes. The dictionary size can be calculated by multiplying the corresponding Rsize value with the size of the original file.

For example, let us consider the RLZ compression of the file `para` with dictionary generated with sample parameters cut = 0.50 and line length = 254. By Table 5.6 the compression ratio of the file `para_50_254` is 0.237. That means the size of the superstring compared to the original file is $50\% \cdot 0.237 = 0.119$, which is the corresponding Rsize value. Let us also assume that the dictionary is encoded using a naive approach, where each symbol is encoded in one byte. The absolute size of the dictionary is therefore 410Mib $\cdot 0.119 =$ 48.79Mib. The absolute size of the factors is $1,220,678 \cdot 8$ bytes = 9.31 MiB. The total size of the compressed `para` file is therefore $48.79 + 9.31 = 58.10$ mebibytes.

With conventionally generated dictionary of the same size, the number of factors was 5,086,768, with the size of $5,086,768 \cdot 8$ bytes = 38.8 mebibytes. The total size of the compressed files is therefore $37.80 + 48.79 = 87.59$ MiB.

Table 5.12 shows the total compressed sizes for each PizzaChili file with different cut and line length parameters. Note that the values presented in Tables 5.3, 5.4, 5.5, 5.6, 5.7 and 5.8 are rounded. The values presented in Table 5.12 are calculated with more precision, causing a small difference compared to values calculated from the earlier presented data. For the same reason, the 75/512 version of the `einstein.de.tx` in the lower section of the table is bolded. The compression with more precise values is better in this case. Note that in real applications, the dictionaries could be encoded using a $\log_2(|\Sigma|)$-bit encoding. Also, the factors could be encoded in less space. The values presented in table 5.12 are examples with sizes of 8 byte factors and a single byte encoding with the dictionary.

Comparing the values in Tables 5.11 and 5.12 we see the significance of the dictionary size with the compression. The bigger the dictionary is, the less significant the number of RLZ factors becomes. We also see that the number of factors tends to decrease faster with the conventional dictionary construction when the dictionary size or the line length parameter is increased. This may be explained with the fact that the probability to find a long substring from the dictionary is lower than to find a short substring. If the sampled substrings are large enough, there is no significant benefit from using the SCS approach.

Regardless of the decrease in the number of factors, the resulting compression is not necessarily better with lesser number of factors. This is because the size of the dictionary tends to grow more rapidly than the number of factors decreases. This is especially true with the SCS dictionary construction, when no compression with the parameters 75%/512, resulted the best compression. Contrarily, with these parameters the conventional dictionary construction resulted the best performance with three files.

File	15_128	15_254	25_254	50_254	50_512	75_512
cere	**41.86**	43.31	46.78	56.26	72.01	83.45
coreutils	39.16	38.80	33.66	**33.22**	36.44	39.84
einstein.de.txt	2.91	**2.16**	2.23	2.31	2.35	2.50
einstein.en.txt	13.59	7.75	8.28	9.95	**7.23**	7.96
influenza	**18.51**	19.21	22.53	30.41	38.25	47.86
kernel	16.28	15.57	**14.23**	14.60	16.48	17.33
para	52.99	51.83	**51.54**	57.93	68.02	78.22
cere	104.84	79.67	**74.94**	76.84	82.17	92.74
coreutils	63.90	58.08	55.39	55.40	**54.00**	55.85
einstein.de.txt	11.73	6.98	6.61	6.35	4.61	**4.61**
einstein.en.txt	41.40	28.06	24.84	23.29	15.92	**15.91**
influenza	25.43	**23.27**	26.77	34.74	41.01	50.73
kernel	70.09	59.38	54.79	50.48	42.13	**40.87**
para	109.71	91.81	**86.23**	87.43	86.65	95.31

Table 5.12: RLZ Compression sizes in mebibytes of the original PizzaChili files with different dictionaries. The best compressions are bolded. The upper numbers result when the RLZ dictionary is a common superstring of the corresponding file. The common superstring is generated from the files randomly sampled with parameters specified in the column title. The lower numbers result when the dictionary is a string obtained by concatenating random samples together such that both dictionaries have the same length.

Investigating Table 5.12 even more, we can observe that for most of the lines in both sections of the table, the compressions behaves similarly. That is, the compression increases to some point which after it starts to decrease. In other words, it seems like there is (a local) minimum for the compression with different sampling parameters. It would be very significant if the optimal parameters could be predicted before the full compression.

6 Conclusions

In this thesis, we investigated the Shortest Common Superstring problem in detail. We focused on the greedy heuristic that provides a computationally easy-to-solve yet practically efficient approximate solution to the problem. The greedy heuristic and Ukkonen's algorithm, that solves this heuristic in linear time, were discussed in detail. The correctness and time complexity were also proved. We provided an implementation of Ukkonen's algorithm with performance comparisons to another algorithm for the same heuristic.

We briefly discussed the LZ family of dictionary compression algorithms. A more detailed introduction to the Relative Lempel-Ziv was given. We hypothesized that the RLZ compression method could be improved for general-purpose data by using an approximate shortest common superstring of the random samples of the input as a dictionary.

The aim of the comparison of the two implementations of the greedy heuristic was to examine the practical performance of our implementation and Ukkonen's algorithm in general. The results of these comparisons show that our implementation of Ukkonen's algorithm is a very competitive method to solve the greedy heuristic of the Shortest Common Superstring problem. Regardless of the alphabet size of the problem instance, and therefore the time complexity of our implementation (linear for small alphabets, $O(n \cdot \log(|\Sigma|)$ for alphabets larger than 12), our implementation was faster with sufficiently large inputs. This indicates that the approximate SCS problem can be quickly solved even for large inputs.

The hypothesis of the compression ratios of the RLZ was also answered. The results indicate that for highly repetitive corpuses, the number of RLZ factors in the compressed data is significantly reduced when using the SCS approach to construct the dictionaries.

The results of this thesis implicate that a speedy implementation of the greedy heuristic of the shortest common superstring can be crafted. However, the comparisons of the implementations lacked the measurement of the used memory. As the memory usage is also a significant aspect of the computation, it would be interesting to study this matter further. As future work, our implementation of Ukkonen's algorithm could be further developed. The version used in this thesis has given sufficient insight into the magnitudes

of time and memory usage. However, we assume that those values could be improved if our implementation were further developed.

The datasets used in this thesis were all highly repetitive corpuses of real origin. Although the number of RLZ factors was significantly decreased with the SCS approach, a further research with more broad datasets would be interesting. Also, the proportion of the time usage of Ukkonen's algorithm compared to the total time of the RLZ factorization was not measured. Performing this kind of study would be a natural continuation of this thesis.

Bibliography

Aho, A. V. and Corasick, M. J. (June 1975). "Efficient String Matching: An Aid to Bibliographic Search". In: *Commun. ACM* 18.6, pp. 333–340. ISSN: 0001-0782. DOI: 10.1145/360825.360855. URL: https://doi.org/10.1145/360825.360855.

Alanko, J. and Norri, T. (2017). "Greedy Shortest Common Superstring Approximation in Compact Space". In: *String Processing and Information Retrieval - 24th International Symposium, SPIRE 2017, Palermo, Italy, September 26-29, 2017, Proceedings*. Ed. by G. Fici, M. Sciortino, and R. Venturini. Vol. 10508. Lecture Notes in Computer Science. Springer, pp. 1–13. DOI: 10.1007/978-3-319-67428-5_1. URL: https://doi.org/10.1007/978-3-319-67428-5%5C_1.

Bentley, J. L. and Sedgewick, R. (1997). "Fast Algorithms for Sorting and Searching Strings". In: *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, 5-7 January 1997, New Orleans, Louisiana, USA*. Ed. by M. E. Saks. ACM/SIAM, pp. 360–369. URL: http://dl.acm.org/citation.cfm?id=314161.314321.

Blum, A., Jiang, T., Li, M., Tromp, J., and Yannakakis, M. (1994). "Linear Approximation of Shortest Superstrings". In: *J. ACM* 41.4, pp. 630–647. DOI: 10.1145/179812.179818. URL: https://doi.org/10.1145/179812.179818.

Bulteau, L., Hüffner, F., Komusiewicz, C., and Niedermeier, R. (2014). "Multivariate Algorithmics for NP-Hard String Problems". In: *Bull. EATCS* 114. URL: http://eatcs.org/beatcs/index.php/beatcs/article/view/310.

Commentz-Walter, B. (1979). "A String Matching Algorithm Fast on the Average". In: *Automata, Languages and Programming, 6th Colloquium, Graz, Austria, July 16-20, 1979, Proceedings*. Ed. by H. A. Maurer. Vol. 71. Lecture Notes in Computer Science. Springer, pp. 118–132. DOI: 10.1007/3-540-09510-1_10. URL: https://doi.org/10.1007/3-540-09510-1%5C_10.

Cormen, T. H. (2009). *Introduction to algorithms*. Cambridge, Masachusetts; London: The MIT Press. ISBN: 9780262033848 0262033844 9780262533058 0262533057. URL: http://www.amazon.de/Introduction-Algorithms-Thomas-H-Cormen/dp/0262033844.

Crochemore, M. and Perrin, D. (1991). "Two-Way String Matching". In: *J. ACM* 38.3, pp. 651–675. DOI: 10.1145/116825.116845. URL: https://doi.org/10.1145/116825.116845.

Crochemore, M. and Rytter, W. (2003). *Jewels of stringology*. New Jersey [u.a.]: World Scientific. ISBN: 9810248970.

Deorowicz, S. and Grabowski, S. (2011). "Robust relative compression of genomes with random access". In: *Bioinform.* 27.21, pp. 2979–2986. DOI: 10.1093/bioinformatics/btr505. URL: https://doi.org/10.1093/bioinformatics/btr505.

Ferragina, P. and Navarro, G. (2005). *The Pizza Chili corpus home page*. URL: http://pizzachili.dcc.uchile.cl/ (visited on 04/19/2022).

Gagie, T., Puglisi, S. J., and Valenzuela, D. (2016). "Analyzing Relative Lempel-Ziv Reference Construction". In: *String Processing and Information Retrieval - 23rd International Symposium, SPIRE 2016, Beppu, Japan, October 18-20, 2016, Proceedings*. Ed. by S. Inenaga, K. Sadakane, and T. Sakai. Vol. 9954. Lecture Notes in Computer Science, pp. 160–165. DOI: 10.1007/978-3-319-46049-9_16. URL: https://doi.org/10.1007/978-3-319-46049-9%5C_16.

Gallant, J. K. (1982). "String Compression Algorithms". AAI8221570. PhD thesis. USA.

Gallant, J., Maier, D., and Storer, J. A. (1980). "On Finding Minimal Length Superstrings". In: *J. Comput. Syst. Sci.* 20.1, pp. 50–58. DOI: 10.1016/0022-0000(80)90004-5. URL: https://doi.org/10.1016/0022-0000(80)90004-5.

Gog, S., Beller, T., Moffat, A., and Petri, M. (2014). "From Theory to Practice: Plug and Play with Succinct Data Structures". In: *Experimental Algorithms - 13th International Symposium, SEA 2014, Copenhagen, Denmark, June 29 - July 1, 2014. Proceedings*. Ed. by J. Gudmundsson and J. Katajainen. Vol. 8504. Lecture Notes in Computer Science. Springer, pp. 326–337. DOI: 10.1007/978-3-319-07959-2_28. URL: https://doi.org/10.1007/978-3-319-07959-2%5C_28.

Gusfield, D. (1997). *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press. ISBN: 9780511574931.

Gusfield, D., Landau, G. M., and Schieber, B. (1992). "An Efficient Algorithm for the All Pairs Suffix-Prefix Problem". In: *Inf. Process. Lett.* 41.4, pp. 181–185. DOI: 10.1016/0020-0190(92)90176-V. URL: https://doi.org/10.1016/0020-0190(92)90176-V.

Hoobin, C., Puglisi, S. J., and Zobel, J. (2011). "Relative Lempel-Ziv Factorization for Efficient Storage and Retrieval of Web Collections". In: *Proc. VLDB Endow.* 5.3, pp. 265–273. DOI: 10.14778/2078331.2078341. URL: http://www.vldb.org/pvldb/vol5/p265%5C_christopherhoobin%5C_vldb2012.pdf.

Horspool, R. N. (1980). "Practical Fast Searching in Strings". In: *Softw. Pract. Exp.* 10.6, pp. 501–506. DOI: 10.1002/spe.4380100608. URL: https://doi.org/10.1002/spe.4380100608.

Kaplan, H. and Shafrir, N. (2005). "The greedy algorithm for shortest superstrings". In: *Inf. Process. Lett.* 93.1, pp. 13–17. DOI: 10.1016/j.ipl.2004.09.012. URL: https://doi.org/10.1016/j.ipl.2004.09.012.

Karp, R. M. and Rabin, M. O. (1987). "Efficient Randomized Pattern-Matching Algorithms". In: *IBM J. Res. Dev.* 31.2, pp. 249–260. DOI: 10.1147/rd.312.0249. URL: https://doi.org/10.1147/rd.312.0249.

Knuth, D. E., Jr., J. H. M., and Pratt, V. R. (1977). "Fast Pattern Matching in Strings". In: *SIAM J. Comput.* 6.2, pp. 323–350. DOI: 10.1137/0206024. URL: https://doi.org/10.1137/0206024.

Kuruppu, S., Puglisi, S. J., and Zobel, J. (2010). "Relative Lempel-Ziv Compression of Genomes for Large-Scale Storage and Retrieval". In: *String Processing and Information Retrieval - 17th International Symposium, SPIRE 2010, Los Cabos, Mexico, October 11-13, 2010. Proceedings*. Ed. by E. Chávez and S. Lonardi. Vol. 6393. Lecture Notes in Computer Science. Springer, pp. 201–206. DOI: 10.1007/978-3-642-16321-0_20. URL: https://doi.org/10.1007/978-3-642-16321-0%5C_20.

Lim, J. and Park, K. (2017). "Algorithm Engineering for All-Pairs Suffix-Prefix Matching". In: *16th International Symposium on Experimental Algorithms, SEA 2017, June 21-23, 2017, London, UK*. Ed. by C. S. Iliopoulos, S. P. Pissis, S. J. Puglisi, and R. Raman. Vol. 75. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 14:1–14:12. DOI: 10.4230/LIPIcs.SEA.2017.14. URL: https://doi.org/10.4230/LIPIcs.SEA.2017.14.

Ma, B. (2008). "Why Greed Works for Shortest Common Superstring Problem". In: *Combinatorial Pattern Matching, 19th Annual Symposium, CPM 2008, Pisa, Italy, June 18-20, 2008, Proceedings*. Ed. by P. Ferragina and G. M. Landau. Vol. 5029. Lecture Notes in Computer Science. Springer, pp. 244–254. DOI: 10.1007/978-3-540-69068-9_23. URL: https://doi.org/10.1007/978-3-540-69068-9%5C_23.

Navarro, G. and Raffinot, M. (2000). "Fast and Flexible String Matching by Combining Bit-Parallelism and Suffix Automata". In: *ACM J. Exp. Algorithmics* 5, p. 4. DOI: 10.1145/351827.384246. URL: https://doi.org/10.1145/351827.384246.

Paluch, K. E. (2014). "Better Approximation Algorithms for Maximum Asymmetric Traveling Salesman and Shortest Superstring". In: *CoRR* abs/1401.3670. arXiv: 1401.3670. URL: http://arxiv.org/abs/1401.3670.

Peltola, H., Söderlund, H., Tarhio, J., and Ukkonen, E. (1983). "Algorithms for Some String Matching Problems Arising in Molecular Genetics". In: *Information Processing*

83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983. Ed. by R. E. A. Mason. North-Holland/IFIP, pp. 59–64.

Pu, I. M. (2005). *Fundamental Data Compression*. USA: Butterworth-Heinemann. ISBN: 0750663103.

Romero, H. J., Brizuela, C. A., and Tchernykh, A. (2004). "An Experimental Comparison of Approximation Algorithms for the Shortest Common Superstring Problem". In: *5th Mexican International Conference on Computer Science (ENC 2004), 20-24 September 2004, Colima, Mexico*. IEEE Computer Society, pp. 27–34. DOI: 10.1109/ENC.2004.1342585. URL: https://doi.org/10.1109/ENC.2004.1342585.

Salmela, Tarhio, J., and Kytöjoki, J. (2006). "Multipattern string matching with q-grams". In: *ACM J. Exp. Algorithmics* 11. DOI: 10.1145/1187436.1187438. URL: https://doi.org/10.1145/1187436.1187438.

Storer, J. A. and Szymanski, T. G. (1982). "Data compression via textual substitution". In: *J. ACM* 29.4, pp. 928–951. DOI: 10.1145/322344.322346. URL: https://doi.org/10.1145/322344.322346.

Tarhio, J. and Ukkonen, E. (1988). "A Greedy Approximation Algorithm for Constructing Shortest Common Superstrings". In: *Theor. Comput. Sci.* 57, pp. 131–145. DOI: 10.1016/0304-3975(88)90167-3. URL: https://doi.org/10.1016/0304-3975(88)90167-3.

Ukkonen, E. (1990). "A Linear-Time Algorithm for Finding Approximate Shortest Common Superstrings". In: *Algorithmica* 5.3, pp. 313–323. DOI: 10.1007/BF01840391. URL: https://doi.org/10.1007/BF01840391.

Welch, T. A. (1984). "A Technique for High-Performance Data Compression". In: *Computer* 17.6, pp. 8–19. DOI: 10.1109/MC.1984.1659158. URL: https://doi.org/10.1109/MC.1984.1659158.

Ziv, J. and Lempel, A. (1977). "A universal algorithm for sequential data compression". In: *IEEE Trans. Inf. Theory* 23.3, pp. 337–343. DOI: 10.1109/TIT.1977.1055714. URL: https://doi.org/10.1109/TIT.1977.1055714.

– (1978). "Compression of individual sequences via variable-rate coding". In: *IEEE Trans. Inf. Theory* 24.5, pp. 530–536. DOI: 10.1109/TIT.1978.1055934. URL: https://doi.org/10.1109/TIT.1978.1055934.