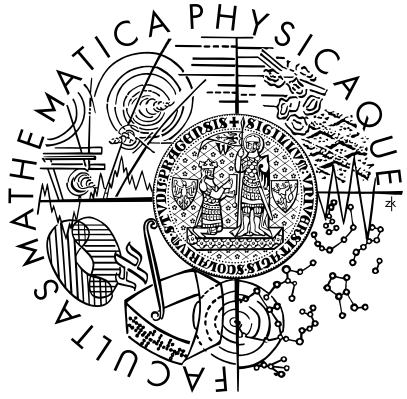
Faculty of Mathematics and Physics
Charles University, Prague

Master Thesis



Vojtěch Toman

Compression of XML Data

Department of Software Engineering
Supervisor: Prof. RNDr. Jaroslav Pokorný, CSc.
Computer Science

I would like to thank Prof. RNDr. Jaroslav Pokorný, CSc. for his helpfulness and patience while supervising this work.

I declare that I have worked out this thesis on my own, using only the resources stated. I agree that the thesis may be publicly available.

Prague, March 20, 2003 Vojtěch Toman

Contents

Abstract

Title: Compression of XML Data
Author: Vojtěch Toman
Department: Department of Software Engineering
Supervisor: Prof. RNDr. Jaroslav Pokorný, CSc.
Supervisor's e-mail address: pokorny@ksi.ms.mff.cuni.cz
Abstract: XML is becoming a standard format for electronic data storage and exchange. However, due to the verbosity of XML, its space requirements are often substantially larger in comparison to other equivalent data formats. The problem can be addressed if data compression is applied. In this thesis, we investigate the possibilities of syntactical compression of XML data based on probabilistic modeling of XML structure. The proposed compression scheme works sequentially, making on-line compression and decompression possible. During the decompression, the reconstructed data can be accessed via the SAX interface. We have implemented an experimental XML compression library (Exalt) and have extensively tested its performance on a wide variety of XML documents. The library si extensible, and inteded to serve as a platform for further research in the field of XML data compression.
Keywords: XML, data compression, syntactical compression, probabilistic modeling.

Název práce: Komprese XML dat
Autor: Vojtěch Toman
Katedra: Katedra softwarového inženýrství
Vedoucí práce: Prof. RNDr. Jaroslav Pokorný, CSc.
e-mail vedoucího: pokorny@ksi.ms.mff.cuni.cz
Abstrakt: Jazyk XML se rychle stává standardem pro reprezentaci a ukládání elektronických dat. Textová povaha XML s sebou bohužel přináší zvýšení prostorových nároků, často výrazné ve srovnání s ekvivalentními datovými formáty. Jednou z cest k řešení tohoto problému je použití metod komprese dat. V této diplomové práci zkoumáme možnosti syntaktické komprese XML dat využívající pravděpodobnostního modelování XML struktury. Výsledné kompresní schéma pracuje inkrementálně, díky čemuž může být jak komprese, tak i dekomprese prováděna sekvenčně. Během dekomprese je navíc možné přistupovat k rekonstruovaným datům pomocí rozhraní SAX. Implementovali jsme experimentální knihovnu pro kompresi XML dat (Exalt) a provedli řadu testů na širokém spektru XML dokumentů. Vzhledem k tomu, že knihovna je snadno rozšiřitelná, může být použita jako platforma pro další výzkum na poli komprese XML dat.
Klíčová slova: XML, komprese dat, syntaktická komprese, pravděpodobnostní modelování.

Chapter 1

Introduction

The Extensible Markup Language (XML) [30] is rapidly becoming a *de facto* standard format for electronic data structuring, storage and exchange. It has been enthusiastically adopted in many areas of computer industry; for example, it is being increasingly used in a variety of database and eBusiness applications. Many expectations are surrounding XML, and it is therefore likely that the amount of data available in XML will grow substantially in the near future.

The problem with XML is that it is text-based, and verbose by its design (the XML standard explicitly states that "terseness in XML markup is of minimal importance"). As a result, the amount of information that has to be transmitted, processed and stored is often substantially larger in comparison to other data formats. This can be a serious problem in many occasions, since the data has to be transmitted quickly and stored compactly. Large XML documents not only consume transmission time, but also consume large amounts of storage space.

The problem can be addressed if data compression is used to reduce the space requirements of XML. Because XML is text-based, the simplest and most common approach is to use the existing text compressors and to compress XML documents as ordinary text files. However, although it is possible to reduce the amount of data significantly in this way, the compressed XML documents often remain larger than equivalent text or binary formats [20]. It is obvious that this solution is only suboptimal, since two documents carrying the same message should have the same entropy—and therefore it should be possible to compress them to about the same size. The main reason is that general-purpose compressors often fail to discover and utilize the redundancy contained in the structure of XML. Another problem with these compressors is that they introduce another pass into XML processing, since decompression is necessary before the data can be processed.

Recently, a number of XML-conscious compressors have emerged that improve on the traditional text compressors. Because they are designed to take advantage of the structure of XML during the compression, they often achieve considerably better results. Furthermore, some of them also make the XML processing APIs[1] accessible, allowing to process

[1] Application Programming Interface

the data right during the decompression.

Very often, these tools rely on the functionality of the existing text compressors, and only adapt them to XML. In this thesis, we decided to go in a slightly different direction. The fact that XML has a context-free nature motivated us to take a more syntactical oriented approach to XML compression. Among the available syntactical compression techniques, so called grammar-based codes, introduced recently in [16], attracted our attention. The grammar-based coding represents a rather novel topic in lossless data compression, and is still a subject of intense research. Since we were not aware of any previous implementation of this promising technique, to employ it in our compression scheme was a challenging task. Moreover, it was also an interesting test on how well it performed in practice.

In XML, the data is organized into a hierarchical and self-describing structure of elements. We observed that this structure is fairly regular and repetitive in many occasions. If we were able to model it during the compression, it would be possible to predict it—and, as a result, to improve the overall compression effectiveness of our compression scheme.

We have implemented two modeling strategies in our compression scheme, which we call the simple modeling and the adaptive modeling, respectively. While the first strategy acts rather as a foundation for further enhancements, the latter improves on it in many ways. In the adaptive modeling, the compressor learns the structure of the input data, and utilizes this knowledge in the process of compression. It follows from our experimental results that by using the adaptive modeling, the amount of data that has to be compressed can be substantially reduced. Another benefit of the adaptive modeling is that it gives us resources that allow us to measure the structural complexity of XML documents.

The implementation of both the compressor and the decompressor is available in a form of a C++ library called Exalt (*An Experimental XML Archiving Library/Toolkit*). The library was designed to be component-based, and easy to use and extend. Thanks to that, it represents a suitable framework for future research in the field of syntactical XML data compression.

The following text is organized as follows. In Chapter 2, we present a brief overview of XML and related technologies. Chapter 3 provides some necessary background in data compression. Besides the traditional compression techniques, the grammar based codes are discussed in more detail. In Chapter 4, the existing XML-conscious compressors are overviewed. In Chapter 5, we discuss the main principles and features of our compression scheme. Since we make use of probabilistic modeling of XML structure during the compression, Chapter 6 describes the two possible modeling strategies that we have implemented. In Chapter 7, the architecture of the prototype implementation is sketched. Some details on the implementation are then discussed in Chapter 8. We have extensively tested the performance of our compressor, and the results are summarized in Chapter 9. Finally, we draw some conclusions in Chapter 10. Appendix A contains the user documentation to our compressor, while the developer documentation can be found in Appendix B.

Chapter 2

XML basics

The *Extensible Markup Language* or XML [30] is a markup language that describes electronic data in the form accessible both to humans and computer programs. Its roots can be seen in another markup language called SGML (Standard Generalized Markup Language). The main domain of SGML lies in structuring textual documents for automated processing and publishing. In SGML, the data is represented as a mixture of text and markup. The markup is represented by textual tags, and organizes and identifies the individual components of the document.

The SGML standard soon turned out to be unnecessarily complex and very difficult to implement. The implementations of SGML were often incomplete or unstable, and, at the same time, more expensive than the proprietary solutions. Paradoxically, the power of SGML represented its biggest disadvantage. Therefore, a considerable effort has been made to define simpler and more usable subsets of SGML.

Besides XML, probably the most successful application of SGML is Hypertext Markup Language (HTML), a language for publishing hypertext documents on the Web. Although there are obvious similarities between HTML and XML, XML is much more general. HTML is nothing but a fixed set of tags with predefined meaning, while in XML, we are allowed to define any tags we want; there is no predefined set to choose from. Therefore, we often speak of XML as a *metalanguage*, because it makes it possible to define other languages. From this point of view, HTML can be seen as an application of XML.

Each XML document may optionally contain the description of its grammar. This grammar is described in the *document type definition* (DTD) and may be used for automated testing of the structural validity of the document. The DTD defines the *logical structure* of the document.

XML documents may be split into independent components to make their reuse possible. It is also possible to combine XML with data in different formats. In this case, we speak of *physical structure* of XML documents.

2.1 XML documents

As in SGML, textual tags are used to markup the components of the document (see Figure 2.1). The building block of each XML document is an *element*. An element is defined by its *start tag* and *end tag*, and by the enclosed data (called the *content* of the element). To distinguish the tags from the data, characters `<` and `>` are used to enclose the start tags, and characters `</` and `>` to enclose the end tags. The elements may be empty, i.e. with no content. In such a case, they are represented by the start tag immediately followed by the end tag, or by so called *empty-element tag*. Each element has a type, defined by its name. XML is case-sensitive, therefore `name`, `Name`, and `NAME` are different names of elements.

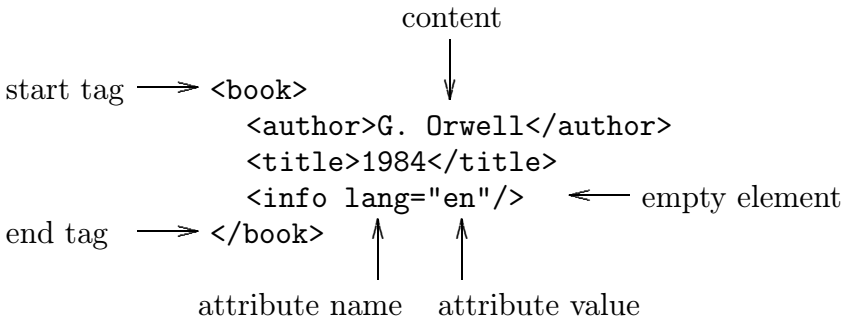


Figure 2.1: Sample XML document

Each element may contain a sequence of *subelements*. The elements are properly nested within each other. In other words, if the start tag of some element is in the content of another element, the end tag is in the content of the same element. The hierarchy of the nested elements forms a tree: there is exactly one *root* element, which is not contained in any other element.

If the element contains only nested elements and no character data, we speak of *element content*. The character data and the nested elements may be mixed within the element—in that case, we speak of *mixed content*.

The elements may contain a set *attributes*. An attribute is defined by its name and its value. If the element contains attributes, the list of them is part of the start tag (or empty-element tag). Because the attributes provide additional information on the elements, they are often referred to as metadata.

We say that an XML document is *well-formed*, if it contains one root element, its elements are properly nested, and they do not contain any two attributes with the same name.

XML documents may—and should—begin with *XML declaration*. In this declaration, the version of XML being used is specified. Optionally, it is also possible to specify the character encoding, and the presence of external DTD:

```
<?xml version="1.0" encoding="ISO-8895-2" standalone="yes"?>
```

The XML declaration is a special case of so called *processing instruction*. Processing instructions may be used within the documents to pass specific information to the application that processes the document. Each processing instruction is defined by its *target*, optionally followed by the *value*:

```
<?target value?>
```

In some occasions, it is necessary to insert a sequence of characters that might be mistaken for the XML markup (characters `<` or `>`, for instance). For this purpose, we may use so called *character data section* to identify the segment of such character data:

```
<![CDATA[Press <<<Enter>>>.]]>
```

We are allowed to use comments within XML documents. A sequence of characters is considered to be a comment, if it begins with the string `<!--` and ends with the string `-->`:

```
<!-- This is a comment -->
```

If the document conforms to some DTD (see Section 2.2), we may specify it in the *document type declaration*. The document type declaration can point to an *external subset* (that is, to the DTD that is stored externally), or can contain the DTD directly in an *internal subset*, or can do both. A document type declaration starts with the keyword `DOCTYPE` and looks as follows:

```
<!DOCTYPE book SYSTEM "book.dtd">
```

The example above says that the root element of the document is `book`, and that the DTD is stored in the file `book.dtd`. The name of the file, following the `SYSTEM` keyword, represents so called *system identifier*.[1] Optionally, a *public identifier* may be specified. The public identifier is preceded by the `PUBLIC` keyword. It does not specify the location of the DTD directly; it is up to the application to resolve it based upon the information contained in the identifier. If the application is not able to resolve the location of the DTD from the public identifier, the system identifier is used.

If the document has an internal subset, the declarations are enclosed with the characters `[` and `]`:

```
<!DOCTYPE book [
...
]>
```

[1] In system identifiers, URIs (Uniform Resource Identifiers) are used to specify the location of the resources.

2.2 Document Type Definition

The *document type definition* (DTD) defines the logical structure of XML documents. In the DTD, we specify the elements, attributes, and entities that can be used in the document, and the context in which they can be used. If a well-formed XML document has an associated DTD and if it complies with the constraints expressed in the DTD, we say that it is *valid*.

A DTD contains a list of declarations enclosed with the characters `<!` and `>`. There are four types of the declarations, which are distinguished by the following keywords: `ELEMENT` (element type declaration), `ATTLIST` (attribute-list declaration), `ENTITY` (entity declaration), `NOTATION` (notation declaration).

In Figure 2.2, a simple DTD that may be used to describe the structure of the document in Figure 2.1 is shown.

```
<!ELEMENT book (author+, title, info?)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT info EMPTY>
<!ATTLIST info lang CDATA #IMPLIED
               genre (prose | poetry) "prose">
```

Figure 2.2: Sample DTD

In the *element type declarations*, the structure of the individual elements in the document is described. Each element type declaration looks as follows:

```
<!ELEMENT element_name (content_model)>
```

The *content model* specifies the type of the content for the element. If the element contains only character data, it should be declared as `PCDATA`.[2] The keyword `EMPTY` may be used for declaring the elements that are empty. If the element contains nested elements, regular expressions are used to specify their type, order and number of occurrences. Besides the names of the nested elements, following operators may be used in the regular expressions: "`*`" (0 or more), "`+`" (1 or more), "`?`" (0 or 1), "`,`" (sequence), and "`|`" (alternative). It is also possible to use parentheses within the regular expressions to define subexpressions. If we do not know the structure of the element, we may use the keyword `ANY` to declare its content, which means that the element may contain any elements that are declared in the DTD, as well as character data.

If the element has mixed content (i.e. it contains character data optionally interspersed with the nested elements $\mathtt{e}_1$ to $\mathtt{e}_n$), following regular expression has to be used in the element type declaration:

[2] Parsed character data

```
<!ELEMENT element_name (#PCDATA | e1 | ...  | en)*>
```

In this declaration, the `PCDATA` keyword must be at the first position in the expression. The types of the nested elements may be constrained, but not their order or their number of occurrences.

As an illustration, consider the declaration of element `book` in Figure 2.2. According to the DTD, it may contain one or more `author` subelements, followed by one `title` subelement. The last and optional subelement is `info`.

Attributes are declared separately from the elements, in so called *attribute-list declarations*. The attributes of one element are usually declared in one attribute-list declaration, but it is possible to declare each of them in a separate declaration. However, the same attributes belonging to different elements must be declared separately for each element. The attribute-list declaration contains the name of the element, and one or more *attribute definitions* specifying the name of the attribute, its type, and, in some occasions, its default value:

```
<!ATTLIST element_name attribute_name type ...>
```

The type of the attribute may be one of the following: `CDATA` (character data), `NMTOKEN` (name token), `NMTOKENS` (set of name tokens), `ENTITY` (entity reference), `ENTITIES` (set of entity references), `ID` (value that is unique within the document), `IDREF` (value of type `ID`), `IDREFS` (set of values of type `ID`), `NOTATION` (the name of the notation), an enumeration of possible values.

In the attribute declaration, the keyword `REQUIRED` means that the attribute must always be provided, `IMPLIED` that it is optional. If the declaration is neither `REQUIRED` nor `IMPLIED`, then the a *default value* can be specified. If the default value is preceded by the `FIXED` keyword, the attribute must always have the default value.

In Figure 2.2, two attributes of element `info` are declared. The attribute `lang` is optional and its values are character data strings. The second attribute, `genre`, may have two possible values (`prose` and `poetry`). If it is not specified, the default value (`prose`) is used.

Besides the elements and attributes, also the entities and notations may be declared in the DTD. Entities represent the units of the physical structure of XML documents. There are two basic types of entities: *general entities* and *parameter entities*. The general entities can be referenced within the entire document, while the parameter entities can be referenced only within the document type definition.

General entities may be contain text or binary data. The text entities may be stored either in the document (in that case we speak of *internal text entities*), or in a separate file (*external text entities*). The *binary entities* must always be stored externally.

The *entity declarations* contain the `ENTITY` keyword. In the following example, declarations of various types of entities are demonstrated.

```
<!ENTITY xml "Extensible Markup Language">
```

```
<!ENTITY footer SYSTEM "footer.xml">
<!ENTITY employees PUBLIC "-//MyCorp//TEXT Employee List //EN"
                          "employees.xml">
<!NOTATION jpeg SYSTEM "image/jpeg">
<!ENTITY car SYSTEM "car.jpg" NDATA jpeg>
```

In the first row, internal entity `xml` is declared. In the second and third rows, two external text entities (`footer` and `employees`) are declared. The location of the external entity must be specified by a system identifier, or optionally by a public identifier.

The binary entity `car` represents an image in the JPEG format. The `NDATA` section specifies the type of the notation. Based upon this type, the application should be able to identify the format of referenced binary data. The *notation declarations* contain the keyword `NOTATION` and assign system or public identifiers to the notations.

The parameter entities are declared similarly to the internal text entities. In the parameter entity declaration, the character `%` precedes the name of the entity:

```
<!ENTITY % common "(para | img)">
```

As stated above, parameter entities can be used only within the DTD, whereas general entities can be used anywhere within the document. To reference the parameter entity, characters `%` and `;` surrounding the entity name are used. In the case of general text entities, the character `&` has to be used instead of `%`. Therefore, to reference the `common` parameter entity, one should write `%common;`, while the `footer` text entity is referenced with `&footer;`. The only way how to reference the data of binary entities is by means of the attributes of type `NOTATION` (for example, `<img src="car">`).

Although the mechanism of DTD is powerful, there are several areas where it may not be sufficient. For example, it is not possible to state how many times a certain element can occur within the document. There is also no way how to specify the format or the type of character data. In the DTD, the character data is always considered to be of type PCDATA, no matter if it represents strings, numbers, or dates. As a solution to this, a new language for describing the content and the structure of XML documents has been proposed. This language—called XML Schema—is XML-based, and improves on the functionality of DTD in many ways [33].

2.3 Processing of XML documents

While it is fairly easy to generate XML documents, it is much more difficult task to process them. There are many problems arising from the parsing of XML data—as a simple example, the white space characters, used for the formatting of the document, may cause misinterpretations if handled imprecisely. Moreover, the processing of XML documents involves other tasks to be performed, such as entity resolving, DTD processing,

validation, error reporting, etc. Therefore, developers of XML-based applications often use standardized APIs for accessing the XML content.

Among these APIs, the most popular are *Simple API for SAX* (SAX) [26] and *Document Object Model* (DOM) [29].

```
startDocument()
startElement("book")
startElement("author")
characters("G. Orwell")
endElement("author")
startElement("title")
characters("1984")
endElement("title")
startElement("info", attributes(["lang", "en"]))
endElement("info")
endElement("book")
endDocument()
```

Figure 2.3: Example of SAX event stream

SAX is based on an event-driven approach: each time the XML parser encounters any relevant information in the document (for example, element start tags, character data, comments, etc.), it emits an "event" to the client application. It is up to the application how it deals with the supplied data. Because the document is processed incrementally, it is not necessary to store it in the memory, which can be crucial for large documents. In Figure 2.3, the SAX events that are emitted by the XML parser during processing the document in Figure 2.1 (page 4) are listed.

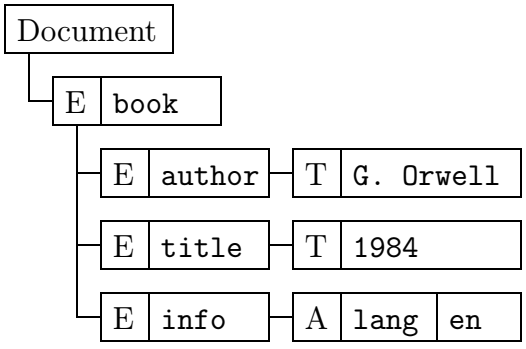


Figure 2.4: Example of a DOM tree

In the DOM model, the document is processed first, and then stored in a tree object structure. The client application can access this structure, and modify its content. The

DOM interface makes it possible to check if the document is well-formed (or valid with respect to its DTD) before the client application starts to process it. In Figure 2.4, the DOM tree representing the document in Figure 2.1 is displayed.

Whether to use SAX or DOM depends on the character of the client application. If we use XML for on-line data exchange, the event-based approach is probably better suited. On the other hand, if we use XML documents as simple databases, for instance, we will probably choose the DOM model. In Table 2.5, the main differences between the SAX and DOM interfaces are summarized.

SAX	DOM
The data of the entire document is not stored in the memory.	The entire document is stored as a tree in the memory.
It is not possible to modify the document.	It is possible to modify the document tree.
The data of the document is immediately accessible to the application.	The document has to be processed first before it is accessible to the application.

Table 2.5: Differences between SAX and DOM

Chapter 3

Data compression

This chapter provides necessary background in information theory and lossless data compression, focusing on techniques that are used or referenced in the thesis later on. Many of the techniques are outlined only roughly, because the exact knowledge of their workings is not necessary for our purposes. More detailed information can be found in [21] or [18], for instance.

Adequate space is devoted to so called grammar-based codes, a syntactical compression technique that we have employed in our XML compression scheme. Since grammar-based codes represent a rather new topic in the field of data compression, and are still a subject of intense research, we describe the principles of this compression scheme, as well as the main results achieved so far, in detail.

3.1 Fundamental concepts

3.1.1 Coding and decoding

Let A be a finite nonempty set of symbols. Let A^* be the set of all finite sequences from A (including the empty sequence λ), and A^+ the set of all nonempty sequences from A. The cardinality of A is denoted as $|A|$, and for any $x \in A^*$, $|x|$ represents the length of x.

The *code* K is a triple $K = (S, C, f)$, where S is a finite set of *source units*, C is a finite set of *code units*, and f is a mapping from S to C^+. The mapping f assigns to every source unit from S just one *codeword* from C^+. The codeword consists of a sequence of code units. Two distinct source units should be never assigned the same codewords, therefore f has to be an injective mapping.

We say that a string $x \in C^+$ is *uniquely decodable* with respect to f, if there is at most one sequence $y \in S^+$ such that $f(y) = x$. Similarly, the code $K = (S, C, f)$ is uniquely decodable, if all strings in C^+ are uniquely decodable with respect to f.

The code K is said to be a *prefix code*, if it has a prefix property, which requires that no codeword is a proper prefix of any other codeword. Prefix codes represent an important and frequently used class of codes, since they are uniquely decodable while reading the

encoded message from left to right.

The conversion of the original data to the compressed (encoded) data is referred to as *coding* or *encoding*. In the reverse process, *decoding*, the compressed data are decompressed (decoded) to reproduce the original data. The corresponding algorithms are called *encoder* and *decoder*, respectively.

There are various ways how to measure the degree of data reduction obtained as the result of the encoding. The most common measures are the *compression ratio* and the *compression rate*.

Compression ratio is a relative term which compares the length of the compressed data to the length of the original data:

$$Compression\ ratio = \frac{Length\ of\ original\ data}{Length\ of\ compressed\ data}$$

Compression rate characterizes the rate of the compressed data. Typically, it is in units of bits per character (bpc) and indicates the average number of bits that are necessary to represent one source character. Compression rate is an absolute term.

Example 3.1. The compression ratio of a compression technique that results in one character of compressed data for every two characters of the original data is 2:1 (sometimes, it is said that the compression ratio is 50%). If the original character is represented by eight bits (i.e. one byte), the compression rate is 4 bits per character. □

3.1.2 Entropy and redundancy

When data is compressed, the goal is to reduce redundancy, leaving only the informational content. To measure the quantity of information within the data, Shannon [27] defined a concept of entropy.

Let $S = \{x_1, x_2, \ldots, x_n\}$ be the set of source units. Let p_i be the probability of occurrence of source unit x_i, $1 \leq i \leq n$. The *entropy* of unit x_i is equal to

$$H_i = -log_2 p_i \text{ bits.}$$

This definition has an intuitive interpretation: If $p_i = 1$, it is clear that x_i is not at all informative since it had to occur. Similarly, the smaller value of p_i, the more unlikely x_i is to appear, and therefore its informational content is larger.

The *average entropy of a source unit* from S is defined as follows:

$$AH = \sum_{i=1}^{n} p_i H_i = -\sum_{i=1}^{n} p_i log_2 p_i \text{ bits.}$$

The *entropy of a source message* $X = x_{i_1} x_{i_2} \ldots x_{i_k}$ from S^+ is then

$$H(X) = -\sum_{j=1}^{k} p_{i_j} log_2 p_{i_j} \text{ bits.}$$

Since the length of a codeword for the source unit x_i must be sufficient to carry the information content of x_i, entropy represents a lower bound on the number of bits that are required for the coded message. Therefore, the total number of bits must be at least as large as the product of the entropies of the units in the source message. And since the value of entropy is generally not an integer, variable length codewords must be used if th elower bound has to be achieved.

If we use d_i bits for encoding the source unit x_i, $1 \leq i \leq n$, then the *length of the encoded message* X is equal to

$$L(X) = \sum_{j=1}^{k} d_{i_j} \text{ bits.}$$

From the above mentioned reasons, it follows that $L(X) \geq H(X)$.[1] The *redundancy of the code* K *for the message* X is defined as follows:

$$R(X) = L(X) - H(X) = \sum_{j=1}^{k} (d_{i_j} + p_{i_j} log_2 p_{i_j}) \text{ bits.}$$

Redundancy can be interpreted as a measure of the difference between the codeword length and the information content.

The *average length of a codeword* of code K is equal to the weighted sum:

$$AL(K) = \sum_{i=1}^{n} d_i p_i \text{ bits.}$$

The *average redundancy of code* K is equal to

$$AR(K) = AL(K) - AH(S) = \sum_{i=1}^{n} p_i((d_i + log_2 p_i) \text{ bits.}$$

If a code K has minimum average codeword length for given probability distribution, it is said to be a *minimum redundancy code*. We say that a code K is *optimal*, if it has minimum redundancy. A code K is *asymptotically optimal* if it has the property that for a given probability distribution, the ration of $AL(K)/AH$ approaches to 1 as entropy tends to infinity. That is, asymptotic optimality guarantees that average codeword length approaches the theoretical minimum.

A code K is *universal* if it maps source units to codewords so that the resulting average codeword length is bounded by $c_1 AH + c_2$, where c_1 and c_2 are constants. Given an arbitrary source with nonzero entropy, a universal code achieves average codeword length that is at most a constant times the optimal possible for the source. A universal code is asymptotically optimal iff $c_1 = 1$.

[1] For more details, see for example [27].

3.2 Representation of integers

3.2.1 Fibonacci codes

In this section, we describe a universal coding scheme based on Fibonacci numbers. While the Fibonacci codes are not asymptomatically optimal, they compare well to other codes (for example the Elias codes [21]) in a very large initial range.

The Fibonacci codes are based on Fibonacci numbers of the order $m \geq 2$. Fibonacci numbers of the order m are defined recursively:

$$\begin{aligned} F_0 &= F_{-1} = \ldots\ F_{-m+1} = 1 \\ F_n &= F_{n-m} + F_{n-m+1} + \ldots + F_{n-1}, n \geq 1 \end{aligned}$$

Suppose that $m = 2$. In that case, every integer number N greater than 0 has precisely one binary representation of the form $R(N) = \sum_{i=0}^{k} d_i F_i$ where $d_i \in \{0, 1\}, k \leq N$, and F_i are the order-2 Fibonacci numbers. There are no adjacent ones in this representation.

N	$R(N)$	$F(N)$
1	1	1 1
2	1 0	0 1 1
3	1 0 0	0 0 1 1
4	1 0 1	1 0 1 1
5	1 0 0 0	0 0 0 1 1
6	1 0 0 1	1 0 0 1 1
7	1 0 1 0	0 1 0 1 1
8	1 0 0 0 0	0 0 0 0 1 1
16	1 0 0 1 0 0	0 0 1 0 0 1 1
32	1 0 1 0 1 0 0	0 0 1 0 1 0 1 1

Table 3.1: Example of order-2 Fibonacci codes

The order-2 Fibonacci code for N is defined to be $F(N) = d_0 d_1 d_2 \ldots d_k 1$. In other words, the Fibonacci representation $R(N)$ is reversed, and 1 is appended. The resulting binary codewords form a prefix code, since every codeword terminates in two consecutive 1's which cannot appear anywhere else in a codeword. Table 3.1 shows Fibonacci representations for some integers.

It is proven that the Fibonacci code of order 2 is universal with $c_1 = 2$ and $c_2 = 3$. Because $c_1 \neq 1$, it is not asymptotically optimal. Fibonacci codes of higher orders compress better, but no Fibonacci code is asymptotically optimal.

3.3 Statistical methods

3.3.1 Huffman coding

Huffman codes approximate the optimum coding by variable-length prefix codes. The original Huffman's algorithm is static and takes an ordered list of probabilities $p_1, p_2, \ldots, p_n$ associated with the source units as input, and constructs a code tree called the *Huffman tree*. Two passes over the data are required.

In the adaptive variant of the algorithm, the probabilities change while passing through the message, thus changes of the initial tree have to be performed. The message can be encoded in one pass. Furthermore, we do not need to store the mapping for the Huffman code, which was required by the static variant.

In the following text, we describe the principles of static Huffman coding, and then we discuss the adaptive versions.

3.3.1.1 Static Huffman coding

The algorithm of static Huffman coding takes a list of nonnegative weights as input and constructs a full binary tree[2] whose leaves are labeled with the weights. The weights represent the probabilities of the associated source units. In Algorithm 3.1, the original Huffman's algorithm for construction of the Huffman tree is presented.

Input: n source units and an ordered list of probabilities (frequencies) $p[i]$, $1 \leq i \leq n$, associated with the source units.

Output: Ordered list of n binary codewords.

begin /* construction of the Huffman tree */
  create a leaf $o(p[i])$ of a binary tree for each source unit i;
  /* node o of the unit i is labeled by the probability $p[i]$ */
  $k := n$;
  while $k > 2$ **do begin**
    choose two smallest nonzero probabilities $p[r]$ and $p[s]$, where $r \neq s$;
    $q := p[r] + p[s]$;
    create a node o labeled by q and edges $[o(q), o(p[r])]$ and
    $[o(q), o(p[s])]$ labeled by 0 and 1, respectively;
    $p[r] := q$; $p[s] := 0$; $k := k - 1$;
  end /* construction of codewords */
  concatenate labels 0 and 1 of edges on the path from the root
  to the leaf and assign it to the source unit i, $1 \leq i \leq n$;
end

Algorithm 3.1: Static Huffman coding

[2] A binary tree is full if every node has either zero or two children.

The Algorithm 3.1 yields a minimal prefix code. Furthermore, it is proven to produce a mininimum redundancy code. The upper bound on the redundancy of a Huffman code is $p_n + 0.086$, where p_n is the probability of the least likely source unit. The average length AL of codewords obtained by Huffman's algorithm is the same as the weighted path-length $AEPL = \sum_{i=1}^{n} d[i] * p[i]$ where $d[i]$ is the length of the path from the root to the leaf i.

It is not necessary to normalize the frequencies as probabilities, i.e. as numbers from the interval $[0, 1]$. The Huffman's algorithm yields the same results with frequencies of source units in a message. Also, the labeling of edges by 0 and 1 in Algorithm 3.1 is only one of the possible conventions. The reverse order also leads to the optimal code. Only the lengths of codewords are essential.

If the list of weigths is presorted, the Huffman mapping can be generated in $O(n)$. The number of nodes of the Huffman tree is $2n - 1$.

Example 3.2. Let $(1, 1, 1, 1, 3, 3, 3, 3, 7)$ be an ordered list that expresses frequencies of source units occurring in a message over an alphabet of size $n = 9$. The application of Algorithm 3.1 provides a Huffman tree and the associated Huffman code as illustrated in Figure 3.1. While the average entropy of the source is $2,869$ bit, the average length of a codeword is $2,840$ bit. □

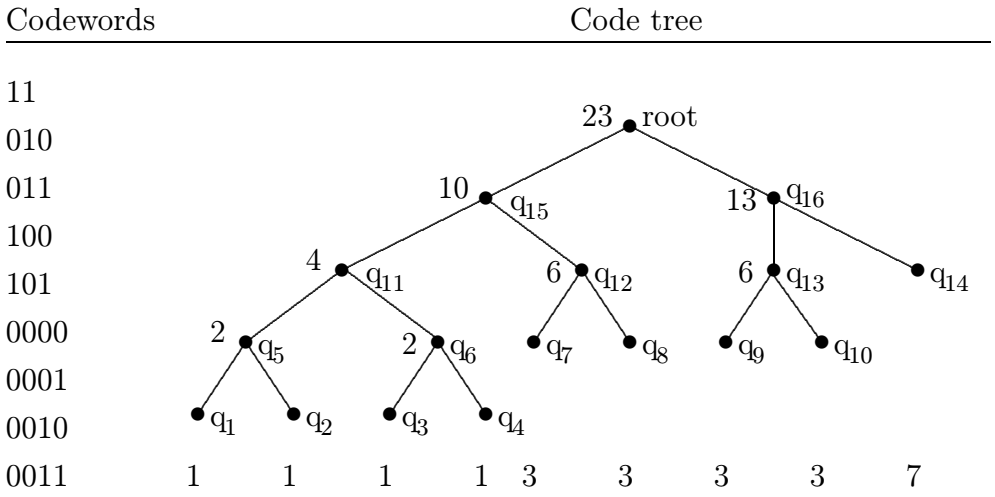


Figure 3.1: Huffman code and code tree

3.3.1.2 Adaptive Huffman coding

One disadvantage of the static Huffman codes is that the code tree has to be included in the encoded message. Therefore, there were various attempts to make the original algorithm adaptive. The two most significant adaptive variants of the static Huffman's algorithm are called FGK and V algorithm, respectively. In these algorithms, the processing time required

to encode and decode a source unit is proportional to the length of the codeword. Thanks to that, encoding and decoding can be performed in the real time. Another advantage of the adaptive algorithms is that they require only one pass over the data since the encoder is "learning" the characteristics of the data on the fly. To make the decoding possible, the decoder must learn along with the encoder by continually updating the Huffman tree so as to stay in synchronization with the encoder.

The FGK algorithm is based on so called *sibling property* which is defined as follows. A binary tree has a sibling property if each node (except the root) has a sibling, and if the nodes can be listed in order of nonincreasing weight with each node adjacent to its sibling. It can be proven that a binary prefix code is a Huffman code if and only if the code tree has the sibling property.

In the FGK algorithm, we maintain a counter for source units and increment them by the value $INCR$ (usually $INCR = 1$). The increments cause possible reorganizations of the associated Huffman tree. We explain the propagation of increasing a leaf weight in two phases (they can be coalesced in a single traversal from the leaf to the root).

1. Let a Huffman tree T1 be the current code tree. Exchange certain subtrees of T1 in such a way that each node o whose weight will be increased in the phase 2 obtains in the new induced ordering the highest number among nodes with the same weights. (So, if these nodes are $o_k, o_{k+1}, \ldots, o_{k+m}$, then $o = o_{k+m}$.) Denote the resulting tree by T2.

2. Propagate $INCR$ in T2 from the leaf to the root.

The sibling property is preserved at the end of phase 2. In Algorithm 3.2, the more complicated phase 1 is described.

Input: A current Huffman tree T1.

Output: A Huffman tree T2.

begin /* a recently processed node is in variable *current* */
  current := o;
  while *current* $\neq$ *root* **do begin**
    exchange the node in *current* (including a subtree that it defines)
    with the node o' which has the highest number among nodes with
    the same weight; /* *current* contains o' */
    current = *predecessor*(*current*);
  end
end

Algorithm 3.2: Adaptive Huffman coding (a core of FGK)

Example 3.3. Suppose the set of source units $\{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}, \mathbf{f}\}$. The Huffman tree T1 shown if Figure 3.2a) was formed in the FGK algorithm (numbers denote weights, bold numbers denote the associated ordering). The next unit to be encoded is **f**. The unit **f** will be encoded via T1 as 1011. After processing the phase 1, the resulting tree will be T2 (Figure 3.2b)). The phase 2 will cause increasing of weights as illustrated in Figure 3.2c). The next occurrence of the unit **f** will be encoded as 001. □

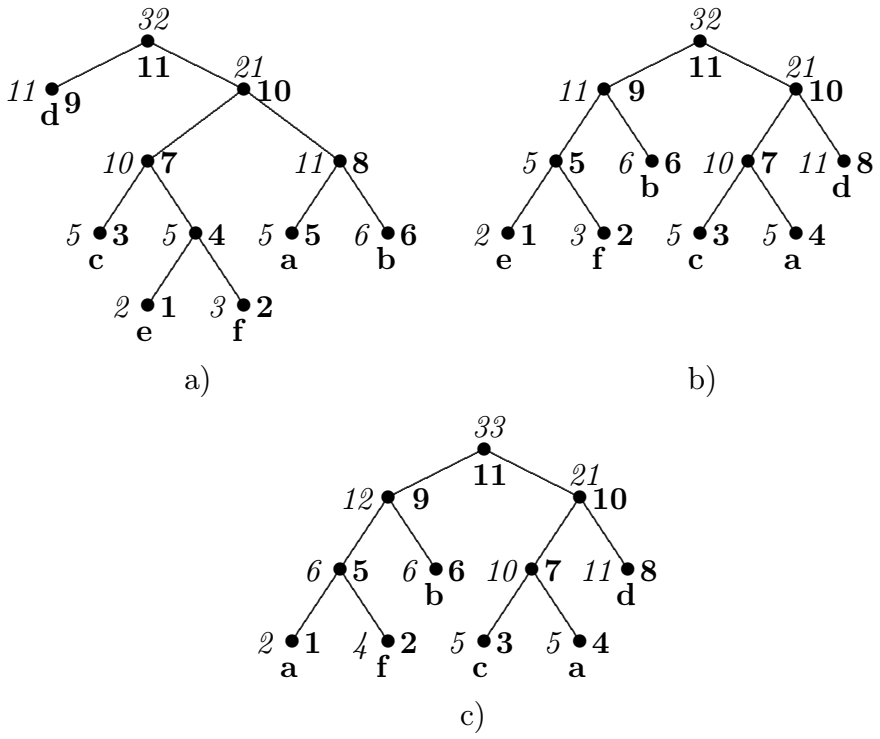


Figure 3.2: Adapting of a Huffman tree

It is known result that FGK algorithm is never much worse than twice optimal. If AL_{HS} and AL_{HD} are the average codewords lengths in static Huffman coding and dynamic Huffman coding, respectively, then $AL_{HD} \leq 2AL_{HS}$.

The time required for both the encoding and the decoding operations is $O(d)$, where d is the current length of the codeword.

Vitter proposed another adaptive Huffman algorithm which is known as the V algorithm. This algorithm incorporates two improvements over the FGK algorithm:

- The number of interchanges on the phase 1 is limited to 1.
- In minimizes not only $\sum_{i=1}^{n} d[i] * p[i]$, but also $\sum_{i=1}^{n} d[i]$ and $\max_i d[i]$.

The intuitive explanation of the advantage of the V algorithm over the FGK algorithm is as follows. As in the FGK algorithm, the code tree constructed by the V algorithm is

the Huffman code tree for the prefix of the source units seen so far. The adaptive methods do not assume that the relative frequencies of the prefix represent accurately the symbol probabilities over the entire message. Therefore, the fact that the V algorithm guarantees a tree of minimum height (height $= \max_i d[i]$) and minimum external path length ($\sum_{i=1}^{n} d[i]$) implies that it is better suited for coding the next source unit of the message.

The average codeword length in the V algorithm is upper bounded by $AL_{HS} + 1$, which is the optimum in the worst case among all one-path Huffman schemes.

3.3.2 Arithmetic coding

Suppose that we are given a message composed of source units over some finite alphabet. Suppose also that we know the probabilities of each of the source units, and want to represent the message using the smallest possible number of bits. It is a known result that the Huffman coding gives results equal to the entropy of the message only if the probabilities of the source units are negative powers of two. In other cases, the Huffman coding only approximates the optimum. The reason is that the source units are assigned discrete codewords, each integral number of bits long.

In arithmetic coding, the source units are not assigned a discrete codewords; instead, an overall code for the whole source message is calculated. Thanks to that, the arithmetic coding makes it possible to code the message arbitrarily close to its entropy.

Arithmetic coding is most useful for adaptive compression, especially with large alphabets. For static and semistatic coding, in which the probabilities used for encoding are fixed, Huffman coding is usually more suitable.

The basic idea of arithmetic coding is as follows. The source message is represented by an interval $[0, 1)$ on the real number line. Each unit of the message narrows the interval. As the interval becomes smaller, the number of bits needed to specify it grows. A high-probable unit narrows the interval less than a low-probable unit. In other words, high-probable units contribute fewer bits to the coded string. The method begins with a list of source units and their probabilities. The real number line is then partitioned into subintervals based on cumulative frequencies.

The *cumulative probability* is defined as follows. Let $a_1, a_2, \ldots, a_n$ be an ordered sequence of source units with probabilities $p_1, p_2, \ldots, p_n$. Then the cumulative probability of the source unit a_i is the sum $\sum_{j=1}^{i-1} p_j$.

Source unit	Probability p_i	Cumulative probability cp_i	Subinterval
d	0.001	0.000	$[0.000, 0.001)$
b	0.010	0.001	$[0.001, 0.011)$
a	0.100	0.011	$[0.011, 0.111)$
c	0.001	0.111	$[0.111, 1.000)$

Table 3.2: Probabilities of the source units

Example 3.4. In Table 3.2, probabilities of source units, their cumulative probabilities, and corresponding subintervals are given. In Figure 3.3, the narrowing of interval is demonstrated graphically for the message **aabc**. The message is coded as follows:

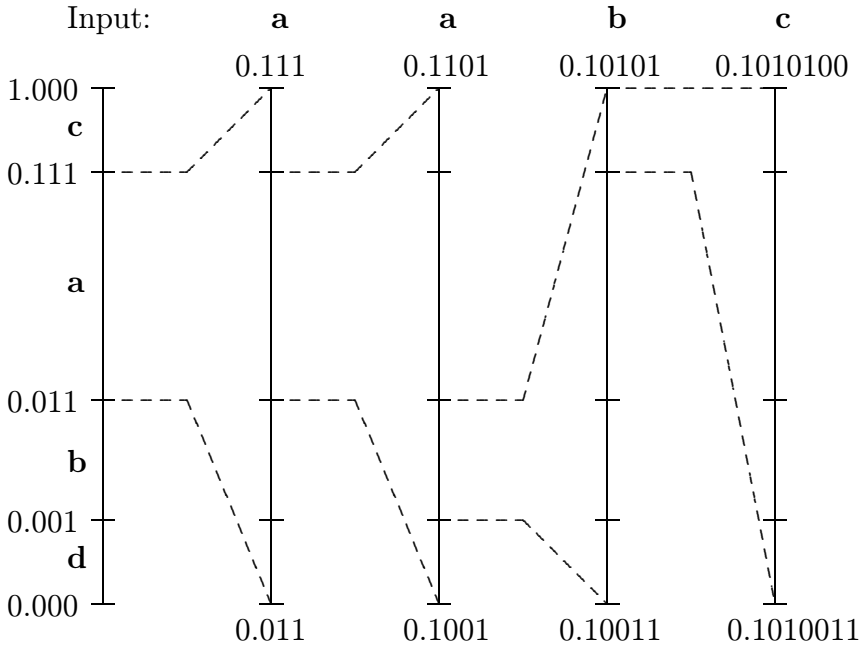


Figure 3.3: Narrowing of the intervals

The initial interval is $[0, 1)$. This interval is partitioned into subintervals using the cumulative probabilities of source units. For the first input character **a**, the subinterval $[0.011, 0.111)$ is chosen for the new interval. This interval is then divided up using the same proportion, and the subinterval $[0.1001, 0.1101)$ is chosen for the second character **a**. Continuing this way, subinterval $[0.10011, 0.10101)$ is chosen for the third character **b**, and subinterval $[0.1010011, 0.1010100)$ for the last character **c** of the input message. The message **aabc** is coded by this interval and any number in the interval can represented it. The number 0.1010011 is quite suitable since it is the shortest representation of the above mentioned interval. □

An interval can be represented by a pair (L, S), where L is the lower bound of the interval, and S is its length. Initially, $L = 0$ and $S = 1$. During coding one input character, new values of L and S are computed as follows:

$$\begin{aligned} L &= L + S * cp_i \\ S &= S * p_i \end{aligned}$$

where cp_i is the cumulative probability of the encoded unit (the i-th unit in the source message), p_i is the probability of the encoded unit.

One potential problem is that code might be too long to work with after a while. However, it is possible to output and discard parts as encoding proceeds. For example, after encoding the second symbol **a** from Example 3.4, the interval is $[0.1001, 0.1101)$. Regardless of further narrowing, the output must begin with 1, so this part can be output and forgotten.

During the decoding process, reverse operations are performed in comparison to encoding. Decoding of one symbols proceeds in three steps:

1. Check to which interval the number N representing the encoded string falls.
2. Subtract the lower bound of the selected interval from number N.
3. Divide the result of step 2 by the length of the selected interval.

Example 3.5. Consider a decoder decoding the code 1010011. Since this number falls into the interval $[0.011, 0.111)$, the first character must be **a**. Then subtract the lower bound of the interval:

$$0.1010011 - 0.011 = 0.0100011$$

and divide the result by the length of the interval for **a**:

$$0.0100011/0.1 = 0.100011$$

This number falls again into the interval $[0.011, 0.111)$. The second symbol is **a** again and the procedure proceeds:

$$0.100011 - 0.011 = 0.001011$$

$$0.001011/0.1 = 0.01011$$

This number falls into the interval $[0.001, 0.011)$, so the next symbol is **b**.

$$0.01011 - 0.001 = 0.00111$$

$$0.00111/0.01 = 0.111$$

This number falls into the interval $[0.111, 1.000)$, and therefore the next symbol is **c**.

$$0.111 - 0.111 = 0.000$$

$$0.000/0.001 = 0.000$$

At this point, the encoded message is decoded, but the decoder can proceed because 0 is the code for an arbitrary long sequence of **d**'s. Therefore, a special end-of input symbol has to be used to fix the end of the input message explicitly. □

This simple description of the idea of arithmetic coding has ignored a number of important problems for the implementation. Specifically, the process described above uses

many multiplicative operations, which are expensive, and requires extremely high precision arithmetic, since L and S must be maintained to many bits of precision. There is also a question how best to calculate the cumulative probability distribution, and how best to perform the arithmetic.

The implementation of arithmetic coding by Witten et al. from 1987 uses multiplicative operations for calculating the bounds of the intervals, and arrays to store the cumulative frequencies. The modeling and coding subsystems are separated, making the coder independent on any particular model. The model acts as the "intelligence" of the compression scheme, whereas the coder is the "engine room", which converts a probability distribution and a single symbol drawn from that distribution into a code. Improvements to the coder yield a reduction in time or space usage; improvements to the model yield a decrease in the size of the encoded data.

Moffat, Neal and Witten [22] revised the original algorithm, improving on it in several ways. Their implementation has the following features:

- A more modular division into modeling, estimation, and coding subsystems.
- Changes to the coding procedure that reduce the number of multiplications and divisions and which permit most of them to be done with low-precision arithmetic (shifts and adds).
- Support for larger alphabet sizes and for more accurate representations of probabilities.
- A reformulation of the decoding procedure that greatly simplifies the decoding loop and improves decoding speed.

To store the cumulative frequencies, Fenwick implicit tree data structure is used (for details, refer to [22]). In this representation, n words are required to represent an n-symbol alphabet, and the frequency counts are calculated and updated in $\Theta(\log n)$ time for each symbol in the alphabet.

3.4 Dictionary methods

The dictionary coding is based on the observation that, in a particular text message, some words are repeated. The dictionary coding replaces words (substring of a message) by pointers to some dictionary.

A *dictionary* is a pair $D = (M, C)$, where M is a finite set of phrases and C is a mapping that maps M on a set of codewords. Phrases in the particular dictionary will be the source units according to our terminology.

The selection of phrases for the dictionary may be performed by static, semiadaptive, or adaptive approaches. A static dictionary method uses some fixed dictionary that is prepared in advance. There are various ways how to build the dictionary. For example,

the dictionary can consits of digrams (pairs of consecutive characters). In the semiadaptive approach, the dictionary that is specific for the message is generated. A drawback is that this dictionary must be a part of the compressed message. Hence, the dictionary itself might be compressed too.

For later on, we will focus only on the adaptive methods. Almost all adaptive dictionary methods of the data compression are based on two basic principles described by Lempel and Ziv. These methods are labeled LZ77 and LZ78, respectively. LZ77 is referred to as the sliding window method, LZ78 uses the growing dictionary.

3.4.1 Sliding window methods

LZ77 method uses pointers to a fixed-size window that precedes the coding position. There are many variants of this method. For our purposes, we describe only the original version of LZ77.

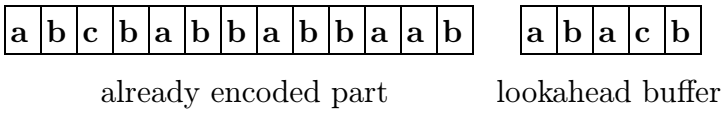


Figure 3.4: Sliding window

Figure 3.4 illustrates the sliding window which is divided into a part of the already encoded string and into a lookahead buffer. The typical size of the window is $N \leq 32768$ and the size B of the buffer ranges from 10 to 256. Initially, the first $N - B$ characters of the window are spaces and the first B characters of the text are in the buffer. One encoding step proceeds as follows:

The longest prefix p of the buffer is searched in the window. If such a substring s starting in the allready encoded part is found, then the prefix of the buffer is encoded by a triple $(i, j, \mathbf{a})$, where i is the distance of the first symbol of the substring s from the buffer, j is the length of the substring s, and $\mathbf{a}$ is the first character following the prefix p. The window is then shifted by $j + 1$ characters to the right. If no substring which matches some prefix in the buffer is found, then the triple $(0, 0, \mathbf{a})$ is produced, where $\mathbf{a}$ is the first character in the buffer. The substring s may overlap to the buffer.

In Figure 3.4, the substring found is **aba**. It starts two characters before the buffer. The longest prefix is coded by the triple $(2, 3, \mathbf{c})$. The window is then shifted by $3 + 1 = 4$ characters.

The decoding is similar to the encoding. The decoder has the same window as the encoder, but instead of searching for the longest prefix of the buffer, it copies the target of the triple into the lookahead buffer.

In each encoding step, $(N - B) * B$ character comparisons must be executed, unless a more intricate algorithm for string matching (such as the Knuth-Morris-Pratt's algorithm) is used.

The popular program Gzip uses LZ77 in conjunction with the Huffman coding for compression of the literals and lengths. In the PKZip compressor, the LZSS method is used. This method is a modification of LZ77 that includes the pointers into the output only if they point to a substring that is longer than te pointer itself; otherwise the characters themselves are included in the output. In order to make this possible, an extra bit must be added to each pointer or character to distinguish between them.

3.4.2 Growing dictionary methods

The growing dictionary methods are based on the LZ78 method, which creates the dictionary by inserting phrases from the message being compressed.

The LZ78 method divides the message into phrases. Each new phrase added to the dictionary is composed of the longest phrase which already exists in the dictionary with one additional symbol. Each such phrase is encoded by the index of its prefix appended by the symbol. This new phrase is inserted into the dictionary and then it may be referenced by the appropriate index. The encoding process in the LZ78 method is illustrated in the Table 3.3.

Input	**a**	**b**	**ab**	**c**	**ba**	**bab**	**aa**	**aaa**
Phrases	1	2	3	4	5	6	7	8
Output	$(0, \mathbf{a})$	$(0, \mathbf{b})$	$(1, \mathbf{b})$	$(0, \mathbf{c})$	$(2, \mathbf{a})$	$(5, \mathbf{b})$	$(1, \mathbf{a})$	$(7, \mathbf{a})$

Table 3.3: Example of the LZ78 method

Table 3.4 presents the dictionary created during the compression.

Phrase	Index	Coded phrase
a	1	**a**
b	2	**b**
ab	3	1**b**
c	4	**c**
ba	5	2**a**
bab	6	5**a**
aa	7	1**a**
aaa	8	7**a**

Table 3.4: Generated dictionary

The output is composed of pairs (phrase index, symbol). Phrases containing only one symbol are coded with the first component equal to zero.

The most timeconsuming part of the LZ78 method is searching out the dictionary. This searching can be implemented using a trie data structure. For each phrase, there is a node

in the trie containing its index. The phrase is on the path from the root to the node associated with it. The decoder maintains the same table as the encoder.

If the space in the memory for the dictionary is exhausted, the dictionary is cleaned out and the process of coding and decoding continues as if it started on a new input.

There are many variants of the LZ78 method. For example, the LZW method uses only indexes for the output. This possible due to following two ideas:

- The dictionary is initiated with entries for all input symbols.
- The last symbol of each phrase is taken as the first symbol of the next phrase.

Program Compress which is available on UNIX systems uses the LZC method based on the LZW method. In LZC, the pointers are encoded by codes of the increasing size with the number of phrases in the dictionary. Furthermore, LZC is monitoring the compression ratio and once it begins to decline, the dictionary is cleaned out and creat again from the initial setting.

3.5 Context modeling

The process of compression can be divided into two separate: *modeling* and *coding*. A model is a representation of the data being compressed. Modeling can be seen as the process of constructing this representation. During the coding, the information supplied by the model is converted into a sequence of codebits.

Arithmetic coding provides optimal compression with respect to the model used. In other words, given a model that provides information to the coder, arithmetic coding produces a compressed representation with minimum length.

Given an optimal coding method, modeling becomes a key to effective data compression. The context modeling techniques condition the probabilities of the source units on contexts consisting of one or more preceding source units. These models are called *finite-context models*. If just n preceding source units are used to determine the probability of the next source unit, we speak of *models of order-n*.

A context model may be *blended*, i.e. incorporating several context models of different orders. There are several methods how to effectively blend the models into a single one. In *weighted blending*, the probability of next source unit is computed as a weighted sum of the probabilities supplied by the individual submodels. In practice, this method turns to be too slow, therefore a simpler blending strategy is often used. Instead of weighting the probabilities of all the submodels, only one model is used. If it fails to predict the upcoming source unit, an *escape symbol* is generated by the encoder, and a different model (usually of lower order) is used. To make the escaping possible, the escape symbol should be assigned some probability in each model.

Context modeling may be *static* or *dynamic*. A model is static if the information that it carries remains unchanged during the compression. An adaptive (or dynamic) model modifies the representation of the input as the compression proceeds.

Finite-context models are a special case of the more sophisticated *finite-state models*, which correspond to finite-state automata.

3.5.1 Dynamic Markov modeling

In [7], Horspool and Cormack proposed an adaptive modeling technique that combines Markov modeling with the power of arithmetic coding. It starts with a simple initial finite automaton and adds new states to it by an operation called *state cloning*.

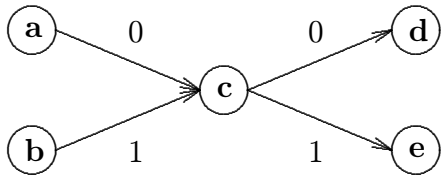


Figure 3.5: Fragment of finite automaton

In the finite automaton, frequency counts are added to each transition. These counts show how many times the transition has been used.

We will the operation of the dynamic Markov modeling on an example. In Figure 3.5, a fragment of a finite state automaton is presented. There are transitions from both states **a** and **b** to state **c**, and transitions from state **c** to both states **d** and **e**. Whenever the model enters state **c**, some contextual information is lost, because we forget whether we reached state **c** from state **a** or **b**. But is is quite possible that the choice of next state, **d** or **e**, is correlated with the previous state, **a** or **b**. A way to learn whether such a correlation exists is to duplicate state **c**, generating a new state **c'**. This creates a revised Markov model as shown in Figure 3.6. After this change to the model, the counts for transitions from **c** to **d** or **e** will be updated only when state **c** is reached from **a**, whereas the counts for **c'** to **d** or **e** will be updated only when **c'** is reached from **b**. Thanks to that, the model can now learn the degree of correlation between the **a**, **b** states and the **d**, **e** states.

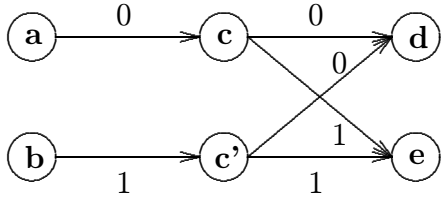


Figure 3.6: Finite automaton after state cloning

The cloning operation is applicable when two conditions are satisfied. The first condition is that the state must have been visited a reasonable number of times. If it has been

visited only a few times, the probability estimates for the next digit will be unreliable. After the cloning operation, the estimates would be even less reliable. The second condition is that there must be more than one predecessor for the state that is being cloned.

The last question concerns updating the frequency counts for the newly created transitions. The frequency counts from state **c'** to states **d** and **e** are divided propotionally to the counts from state **b** to **c'** and from state **a** to **c**, respectively.

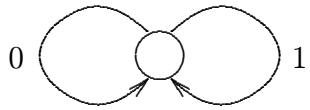


Figure 3.7: Initial one state model

Different initial models may be used. The simplest is the one-state model a shown in Figure 3.7. Regardless of the complexity of the initial model, all dynamic Markov models have proven to be finite-context models.

3.5.2 Prediction by partial matching (PPM)

The prediction by partial matching (PPM) proposed by Cleary and Witten [6] represents a state-of-the-art arithmetic coding method for compressing texts. In PPM, a suite of fixed order context models with different order k, from 0 up to some predetermined maximum, is used to predict upcoming characters.

For each model of order k, a note is kept of all characters that have followed every subsequence of length k observed so far in the input, and the number of times that each has occurred. Prediction probabilities are calculated from these counts.

The models are blended into a single one, and arithmetic coding is used to encode the characters. The combination is achieved through the use of escape probabilities. By default, the model with the largest k is used for coding. However, if a novel character is encountered in this context, an escape symbol is transmitted to instruct the decoder to switch to the model with the next smaller value of k. The process continues until a model is reached in which the character is not novel. To ensure that the process terminates, a (-1)-order model containing all characters in the coding alphabet is assumed.

Table 3.5 shows the state of the four models with $k = 2, 1, 0$, and -1 after the input string **abracadabra** has been processed. For each model, all previously-occurring contexts are shown with their associated predictions, along with occurrence counts c and the probabilities p that are calculated from them.

One potential problem is how to choose the probabilities of the escape events. The method used in the example, commonly called PPMC, gives a count to the escape event equal to the number of different symbols that have been seen in the context so far; thus, for example, in the order 0 column of Table 3.5, the escape symbol receives a count of 5 because five different symbols have been seen in that context.

Order $k=2$			Order $k=1$			Order $k=0$			Order $k=-1$		
Context	c	p	Context	c	p	Context	c	p	Context	c	p
ab $\rightarrow$ **r**	2	$\frac{2}{3}$	**a** $\rightarrow$ **b**	2	$\frac{2}{7}$	**a**	5	$\frac{5}{16}$	A	1	$\frac{1}{\lvert A\rvert}$
Esc	1	$\frac{1}{3}$	**c**	1	$\frac{1}{7}$	**b**	2	$\frac{2}{16}$			
			d	1	$\frac{1}{7}$	**c**	1	$\frac{1}{16}$			
ac $\rightarrow$ **a**	1	$\frac{1}{2}$	Esc	3	$\frac{3}{7}$	**d**	1	$\frac{1}{16}$			
Esc	1	$\frac{1}{2}$				**r**	2	$\frac{2}{16}$			
			b $\rightarrow$ **r**	2	$\frac{2}{3}$	Esc	5	$\frac{5}{16}$			
ad $\rightarrow$ **a**	1	$\frac{1}{2}$	Esc	1	$\frac{1}{3}$						
Esc	1	$\frac{1}{2}$									
			c $\rightarrow$ **a**	1	$\frac{1}{2}$						
br $\rightarrow$ **a**	2	$\frac{2}{3}$	Esc	1	$\frac{1}{2}$						
Esc	1	$\frac{1}{3}$									
			d $\rightarrow$ **a**	1	$\frac{1}{2}$						
ca $\rightarrow$ **d**	1	$\frac{1}{2}$	Esc	1	$\frac{1}{2}$						
Esc	1	$\frac{1}{2}$									
			r $\rightarrow$ **a**	2	$\frac{2}{3}$						
da $\rightarrow$ **b**	1	$\frac{1}{2}$	Esc	1	$\frac{1}{3}$						
Esc	1	$\frac{1}{2}$									
ra $\rightarrow$ **c**	1	$\frac{1}{2}$									
Esc	1	$\frac{1}{2}$									

Table 3.5: PPM model after processing the string **abracadabra**

Suppose that the character following **abracadabra** were **d**. This is not predicted from the current $k=2$ context **ra**. Therefore, an escape event occurs in context **ra**, which is coded with a probability of $\frac{1}{2}$, and then the $k=1$ context **a** is used. This does predict the desired symbol through the prediction **a** $\rightarrow$ **d**, with probability $\frac{1}{7}$.

If the next character were one that had never been encountered before, for example **t**, escaping would take place repeatedly right down to the base level $k=-1$. Once this level is reached, all symbols are equiprobable. Assuming a 256 character alphabet, the **t** is coded with probability $\frac{1}{256}$ at the base level.

It may seem that the performance of PPM should improve when the maximum context length is increased, because the predictions are more specific. However, it follows from the experimental results that the best compression is achieved when a maximum context length is 5 and that it deteriorates when the context is increased beyond this. The reason is that while longer contexts do provide more specific predictions, they often do not give any prediction at all. This causes the escape mechanism to be used more frequently to reduce the context length down. And each escape operation carries a penalty in coding effectiveness.

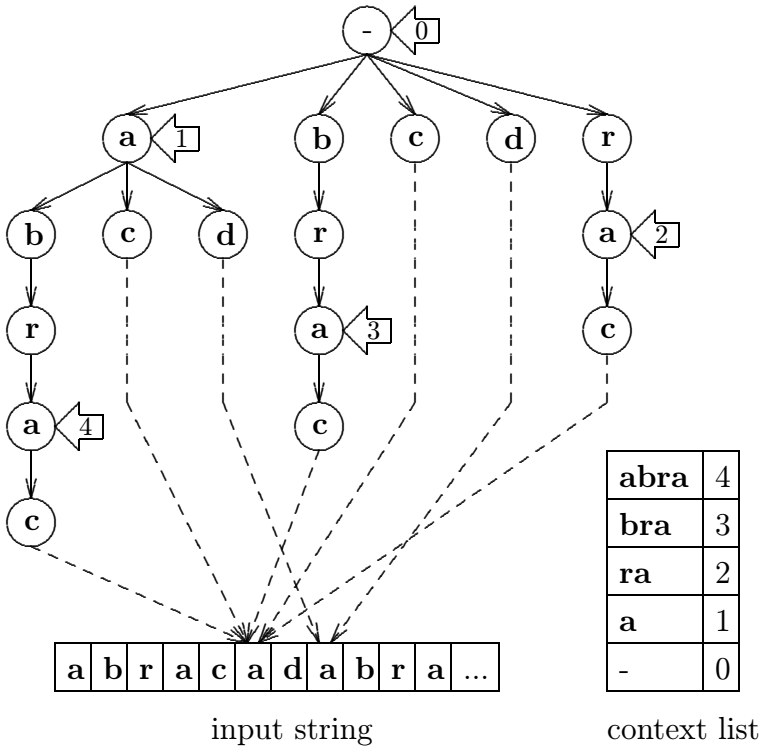


Figure 3.8: Context trie for the string **abracadabra**

Cleary, Teahan and Witten [5] have improved PPM to allow the context length to vary depending on the coding situation. This improved version, called PPM*, makes it possible to store the model in a way that gives rapid access to predictions based on any context, eliminating the need for an arbitrary bound to be imposed.

The main problem associated with the use of unbounded contexts is the amount of memory that is necessary to store them. It is impractical to extend PPM to models with a substantially higher order because of the exponential growth of the memory that is required as k increases. For PPM*, the problem is crucial, as it demands the ability to access all possible contexts right back to the very first character. As a solution to this, PPM* uses a trie structure combined with pointers back to the input string for the representation of the contexts. The resulting data structure, called a *context trie*, allows to save substantial space. Figure 3.8 shows the state of the context trie after the input string **abracadabra** has been processed.

It follows from the experimental results that PPM* provides a performance improvement over PPM, but at the expense of slower run.

3.5.3 Block-sorting compression

The interesting block-sorting compression algorithm introduced by Burrows and Wheeler [2] applies a reversible transformation to a block of input text. The transformation does not itself compress the data, but reorders it to make it easy to compress with simple algorithms (such as move-to-front coding). The algorithm is not adaptive; first the complete input sequence is transformed and then the resulting output is encoded. The algorithm is effective because the transformed string contains more regularities than the original one.

The block-sorting algorithm can be viewed as a context-based method, with no predetermined upper bound to context length.[3] To demonstrate the operation of the algorithm, we use **abraca** as the input string.

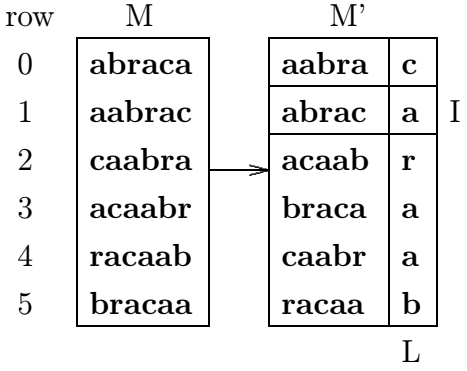


Figure 3.9: Block-sorting compression of **abraca**

The algorithm first generates the matrix of strings M (see Figure 3.9), then sorts the reversed strings alphabetically to produce the matrix M'. Two parameters are extracted from the sorted matrix. The first, I, is an integer that indicates which row number corresponds to the original string. The second, L, is the character string that constitutes the first column. In our example, $I = 1$ and $L = \mathbf{caraab}$. The input string is completely specified by I and L since there exists a reverse transformation for reconstructing the original (this transformation if described in [2] in detail). Moreover, L can be transmitted very economically because it has the property that the same letters often fall together in to long runs.

The vector L is encoded using a move-to-front strategy and Huffman or arithmetic coder. The move-to-front algorithm works as follows. Define a vector of integers $R[0]$, $\ldots$, $R[N-1]$, which are the codes for the characters $L[0]$, $\ldots$, $L[N-1]$. Initialize a list Y of characters to contain each character of the source alphabet exactly once. For each $i = 0, \ldots, N-1$, set $R[i]$ to the number of characters preceding character $L[i]$ in the list Y, then move character $L[i]$ to the front of Y. Taking $Y = [\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{r}]$ initially, and $L = \mathbf{caraab}$, we compute the vector $R = (2, 1, 3, 1, 0, 3)$.

[3] To see why is this true, refer for example to [5].

Huffman or arithmetic coding is applied to the elements of R, treating each element as a separate token to be coded. Any coding technique can be applied as long as the decompressor can perform the inverse operation. Call the output of this coding process OUT. The final output of the block-sorting algorithm is the pair (OUT, I), where I is the number of the row in the matrix M' computed previously.

3.6 Syntactical methods

If the source messages belong to some formal language, we can attempt to create a grammar that describes this language. With the help of such a grammar, it is possible to encode the source messages efficiently.

Two main approaches can be used in the syntactical compression. In one of these approaches, one fixes a grammar G, known to both encoder and decoder, such that the language generated by G contains all of the data strings to be compressed. To compress a particular data string, one then compresses the derivation tree. In the second approach, a different grammar G_x is assigned to each data string x, so that the language generated by G_x is $\{x\}$. If the data string x has to be compressed, the encoder transmits codebits to the decoder that allow the reconstruction of the grammar G_x.

We will focus on techniques that represent the second group of the syntactical methods. The following section is devoted to a detailed description of so called grammar-based codes, introduced recently in [16, 15]. After that, we briefly describe a similar compression technique that is represented by the Sequitur algorithm [24, 25].

3.6.1 Grammar-based codes

Recently, Kieffer and Yang [16, 15] have proposed a new class of lossless source codes called *grammar-based codes*. In the grammar-based code, a data sequence to be compressed is first converted into a context-free grammar by so called *grammar transform*, and then losslessly encoded using arithmetic code.

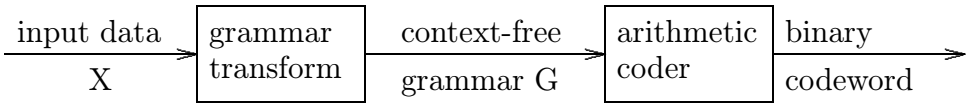


Figure 3.10: Structure of a grammar-based code

The class of grammar-based codes is broad enough to include various other codes, such as Lempel-Ziv types of codes, as special cases.

3.6.1.1 Basic notions

Let A be the source alphabet of size at least 2. Fix a countable set $S = \{s_0, s_1, s_2, \ldots\}$ of symbols, disjoint from A. Symbols in S will be called *variables*; symbols in A will be called *terminal symbols*. For any $j \geq 1$, let $S(j) = \{s_o, s_1, \ldots, s_{j-1}\}$.

For our purposes, a *context-free grammar* G is a mapping from $S(j)$ to $(S(j) \cup A)^+$ for some $j \geq 1$. The set $S(j)$ shall be called the variable set of G. The variable s_0 acts as the *start symbol*.

The mapping $s_i \to G(s_i)$ $(0 \leq i < j)$ is called a *production rule*. The grammar G is completely described by the set of its production rules.

Let G be a context-free grammar. If α and β are strings from $(S(j) \cup A)^+$, we shall write:

- $\alpha \overset{G}{\to} \beta$ if there are strings α_1, α_2 and a production rule $s_i \to G(s_i)$ such that $(\alpha_1, s_i, \alpha_2)$ is a parsing of α and $(\alpha_1, G(s_i), \alpha_2)$ is a parsing of β.
- $\alpha \overset{G}{\Rightarrow} \beta$ if there exist a sequence of strings $\alpha_1, \alpha_2, \ldots, \alpha_k$ such that $\alpha = \alpha_1 \overset{G}{\to} \alpha_2, \alpha_2 \overset{G}{\to} \alpha_3, \ldots, \alpha_{k-1} \overset{G}{\to} \alpha_k = \beta$.

The set $\{x \in A^+ : s_0 \overset{G}{\Rightarrow} x\}$ represents the *language generated by the grammar* G and shall be denoted as $L(G)$.

We say that a context-free grammar G is *admissible* if the following properties hold:

1. G is *determistic* (in other words, for each variable s_i in the variable set of G, there is exactly one production rule whose left member is s_i).
2. The empty string is not the right member of any production rule in G.
3. $L(G)$ is nonempty.
4. G has no useless symbols. This means that for each symbol $x \in S(j) \cup A, x \neq s_0$, there exist finitely many strings $\alpha_1, \alpha_2, \ldots, \alpha_n$ such that at least one of the strings contains x and $s_0 = \alpha_1 \overset{G}{\to} \alpha_2, \alpha_2 \overset{G}{\to} \alpha_3, \ldots, \alpha_{n-1} \overset{G}{\to} \alpha_n \in L(G)$.

For any deterministic grammar G, the language $L(G)$ is either empty or consists of exactly one string. If G is an admissible grammar, there exists a unique string $x \in A^+$ such that $L(G) = \{x\}$. We shall say that G *represents* x, and write $x \to G_x$.

Let G be an admissible grammar with variable set $S(j)$. The size $|G|$ of G is defined as the sum:

$$|G| \triangleq \sum_{v \in S(j)} |G(v)|$$

where $|G(v)|$ denotes the length of the string $G(v) \in (S(j) \cup A)^+$.

Example 3.6. Let $A = \{0, 1\}$. We present an admissible grammar G with the variable set $\{s_0, s_1, s_2, s_3\}$ which represents the sequence $x = 00101101101010101110$ and its size is equal to 14.

$$\begin{aligned} s_0 &\rightarrow 0s_3s_2s_1s_1s_310 \\ s_1 &\rightarrow 01 \\ s_2 &\rightarrow s_11 \\ s_3 &\rightarrow s_1s_2 \end{aligned}$$

□

Remark. Admissible grammars have one important property: we need only list the production rules to fully specify the grammar, because the terminal alphabet, as well as the variable set of the grammar and the start symbol, can be uniquely inferred from the production rules. The set of variables will consist of the left members of the production rules. The terminal alphabet will consist of the symbols which appear in the right members of the production rules and which are not variables. Finally, the start symbol is the unique variable that doesn't appear in the right members of the production rules.

3.6.1.2 Reduction rules

It is obvious that for a string $x \in A^+$, especially when $|x|$ is large, there are many admissible grammars that represent x. Some of these grammars can be more compact than others in the sense of having smaller size $|G|$. Consider for example the grammar from the Example 3.6 and a (rather simplistic) grammar with the only one production rule: $s_0 \rightarrow 00101101101010101110$. Both grammars represent the same data string, but the size of the second grammar is 20!

Since the size of G is influential in the performance of the grammar-based code, the grammars should be designed such that the following properties hold:

(a.1) The size $|G|$ should be small enough, compared to the length of x.

(a.2) Strings represented by distinct variables of G are distinct.

(a.3) The frequency distribution of variables and terminal symbols of G in the range of G should be such that effective arithmetic coding can be accomplished later on.

Kieffer and Yang have proposed a set of *reduction rules* which, when applied repeatedly to an admissible grammar G, lead to another admissible grammar G' which represents the same data string and satisfies the properties **(a.1)**, **(a.2)**, and **(a.3)** in some sense. These reduction rules will be described in the following text.

Reduction Rule 1. Let s be an variable of an admissible grammar G that appears only once in the range of G. Let $s' \to \alpha s \beta$ be the unique production rule in which s appears on the right. Let $s \to \gamma$ be the production rule corresponding to s. Reduce G to the admissible grammar G' obtained by removing the production rule $s \to \gamma$ from G and replacing the production rule $s' \to \alpha s \beta$ with the production rule $s' \to \alpha\gamma\beta$. The resulting admissible grammar G' represents the same sequence x as does G.

Example 3.7. Consider the grammar G with variable set $\{s_0, s_1, s_2\}$ given by $\{s_0 \to s_1 s_1, s_1 \to s_2 1, s_2 \to 010\}$. Applying Reduction Rule 1, one gets the grammar G' with variable set $\{s_0, s_1\}$ given by $\{s_0 \to s_1 s_1, s_1 \to 0101\}$. □

Reduction Rule 2. Let G be an admissible grammar possessing a production rule of form $s \to \alpha_1\beta\alpha_2\beta\alpha_3$, where the length of β is at least 2. Let $s' \in S$ be a variable which is not in G. Reduce G to the grammar G' obtained by replacing the production rule $s \to \alpha_1\beta\alpha_2\beta\alpha_3$ of G with $s \to \alpha_1 s'\alpha_2 s'\alpha_3$, and by appending the production rule $s' \to \beta$. The resulring grammar G' includes a new variable s' and represents the same sequence x as does G.

Example 3.8. Consider the grammar G with variable set $\{s_0, s_1\}$ given by $\{s_0 \to s_1 01 s_1 01, s_1 \to 11\}$. Applying Reduction Rule 2, one gets the grammar G' with variable set $\{s_0, s_1, s_2\}$ given by $\{s_0 \to s_1 s_2 s_1 s_2, s_1 \to 11, s_2 \to 01\}$. □

Reduction Rule 3. Let G be an admissible grammar possessing two distinct production rules of form $s \to \alpha_1\beta\alpha_2$ and $s' \to \alpha_3\beta\alpha_4$, where β is of length at least 2, either α_1 or α_2 is not empty, and either α_3 or α_4 is not empty. Let $s'' \in S$ be a variable which is not in G. Reduce G to the grammar G' obtained by doing the following: Replace rule $s \to \alpha_1\beta\alpha_2$ by $s \to \alpha_1 s''\alpha_2$, replace rule $s' \to \alpha_3\beta\alpha_4$ by $s' \to \alpha_1 s''\alpha_2$, and append the new rule $s'' \to \beta$.

Example 3.9. Consider the grammar G with variable set $\{s_0, s_1, s_2\}$ given by $\{s_0 \to s_1 0 s_2, s_1 \to 10, s_2 \to 0 s_1 0\}$. Applying Reduction Rule 3, one gets the grammar G' with variable set $\{s_0,\ s_1,\ s_2,\ s_3\}$ given by $\{s_0 \to s_3 s_2, s_1 \to 10, s_2 \to 0 s_3, s_3 \to 11\}$. □

Reduction Rule 4. Let G be an admissible grammar possessing two distinct production rules of the form $s \to \alpha_1\beta\alpha_2$ and $s' \to \beta$, where β is of length at least 2, and either α_1 or α_2 is not empty. Reduce G to the grammar G' obtained by replacing the production rule $s \to \alpha_1\beta\alpha_2$ with the production rule $s \to \alpha_1 s'\alpha_2$.

Example 3.10. Consider the grammar G with variable set $\{s_0, s_1, s_2\}$ given by $\{s_0 \to s_2 01 s_1, s_1 \to s_2 0, s_2 \to 11\}$. Applying Reduction Rule 4, one gets the grammar G' with variable set $\{s_0, s_1, s_2\}$ given by $\{s_0 \to s_1 1 s_1, s_1 \to s_2 0, s_2 \to 11\}$. □

Reduction Rule 5. Let G be an admissible grammar in which two variables s and s' represent the same subsequence of the data string represented by G. Reduce G to the

grammar G' obtained by replacing each appearance of s' in the range of G by s and deleting the production rule corresponding to s'. The grammar G' may not be admissible since some further variables of G' may become useless. If so, further reduce G' to the admissible grammar G'' obtained by deleting all production rules corresponding to useless variables of G'. Both G and G'' represent the same data string.

Remark. It is possible to define more reduction rules than Reduction Rules 1-5. For example, if the right members of the production rules of the admissible grammar G contain non-overlapping substrings $\alpha \neq \alpha'$ representing the same subsequence of the data string represented by G, one can reduce G by replacing α and α' with a new variable s, while introducing a new production rule (either $s \rightarrow \alpha$ or $s \rightarrow \alpha'$). However, this new rule is somewhat difficult to implement in practice. Kieffer and Yang limited themselves to Reduction Rules 1-5 because they are simple to implement, and yield grammars which are sufficiently reduced.

An admissible grammar G is said to be *irreducible* if none of Reduction Rules 1 to 5 can be applied to G to get a new admissible grammar. Each irreducible grammar G satisfies the following properties:

(b.1) Each variable of G other than s_0 (the start symbol) appears at least twice in the range of G.

(b.2) There is no non-overlapping repeated pattern of length greater than or equal to 2 in the range of G.

(b.3) Each distinct variable of G represents a distinct sequence from A.

Property **(b.3)** holds due to Reduction Rule 5 and is very important to the compression performance of a grammar-based code. A grammar-based code for which the transformed grammar does not satisfy the property **(b.3)**, may give poor compression results and can not be guaranteed to be universal.

3.6.1.3 Grammar transforms

Let x be a sequence from A which is to be compressed. A *grammar transform* is a transformation that converts x into an admissible grammar that represents x. For our purposes, we are interested particularly in a grammar transform that starts from the grammar G consisting of only one production rule $s_0 \rightarrow x$, and applies repeatedly Reduction Rules 1 to 5 in some order to reduce G into an irreducible grammar G'. Such a grammar transform is called an *irreducible* grammar transform. To compress x, the corresponding grammar-based code then uses a zero order arithmetic code to compress the irreducible grammar G'.

There are different grammar transforms because of the different orders via which the reduction rules are applied; this results in different grammar-based codes. However, all these grammar-based codes are universal, as proved by Kieffer and Yang:

Theorem 1. *Let G be an irreducible grammar representing a sequence x from A. The size $|G|$ of G divided by the length $|x|$ of x goes to 0 uniformly as $|x|$ increases. Specifically,*

$$max\{|G| : G \text{ is irreducible grammar representing } x,\ x \in A^n\} = o(n)$$

Proof. Refer to [16]. □

Theorem 2. *Any grammar-based code with an irreducible grammar transform is universal in the sense that for any stationary, ergodic source $\{X_i\}_{i=0}^{\infty}$, the compression rate resulting from using the grammar-based code to compress the first segment $X_1X_2\ldots X_n$ of length n converges, with probability one, to the entropy rate of the source as n goes to infinity.*

Proof. Refer to [16]. □

The irreducible grammar-based codes combine the power of string matching (Reduction Rules 2 to 4) with that of arithmetic coding, which is the main reason why they are universal.

3.6.1.4 A greedy irreducible grammar transform

In this section, we describe the greedy irreducible grammar transform proposed by Kieffer and Yang, which can construct sequentially a sequence of irreducible grammars from which the original data sequence can be recovered incrementally.

Let $x_1x_2\ldots x_n$ be a sequence from A which has to be compressed. The greedy irreducible grammar transform parses the sequence x sequentially into non-overlapping substrings $\{x_1,\ x_2\cdots x_{n_2},\ \ldots,\ x_{n_{t-1}+1}\cdots x_{n_t}\}$ and builds sequentially an irreducible grammar for each $x_1\cdots x_{n_i}$, where $1 \le i \le t$, $n = 1$, and $n_t = n$. The first substring is x_1 and the corresponding irreducible grammar G_1 consists of only one production rule $s_0 \to x_1$.

Suppose that $x_1, x_2\cdots x_{n_2}, \ldots, x_{n_{i-1}+1}\cdots x_{n_i}$ have been parsed off and the corresponding irreducible grammar G_i for $x_1\cdots x_{n_i}$ has been built. Suppose that the variable set of G_i is equal to $S(j_i) = \{s_0, s_1, \ldots, s_{j_i-1}\}$, where $j_1 = 1$. The next substring $x_{n_i+1}\cdots x_{n_{i+1}}$ is the longest prefix of $x_{n_i+1}\cdots x_n$ that can be represented by s_j for some $0 < j < j_i$ if such a prefix exists. Otherwise, $x_{n_i+1}\cdots x_{n_{i+1}} = x_{n_i+1}$ with $n_{i+1} = n_i + 1$. If $n_{i+1} - n_i > 1$ and $x_{n_i+1}\cdots x_{n_{i+1}}$ is represented by s_j, then append s_j to the right end of $G_i(s_0)$; otherwise, append the symbol x_{n_i+1} to the right end of G_is_0.

The resulting grammar is admissible, but not necessarily irreducible. Apply Reduction Rules 1 to 5 to reduce the grammar to an irreducible grammar G_{i+1}. Then G_{i+1} represents $x_1\cdots x_{n_{i+1}}$. Repeat the described procedure until the whole sequence x is processed. The final irreducible grammar G_t represents x.

Define a function $I : \{1, \ldots, t\} \to \{0, 1\}$ as follows: $I(1) = 0$, and for any $i > 1$, $I(i)$ is equal to 0 if G_i is equal to the apended G_{i-1}, and 1 otherwise.

Since only one symbol from $S(j_i) \cup A$ is appended to the end of $G_i(s_0)$, not all reduction rules can be applied to get G_{i+1}. In fact, the order via which reduction rules are applied is unique. The following theorem shows why this is the case.

Theorem 3. *Let α be the last symbol of $G_i(s_0)$. Let β be the symbol s_j that represents $x_{n_i+1} \cdots x_{n_{i+1}}$ if $n_{i+1} - n_i > 1$, and x_{n_i+1} itself otherwise. Let G'_i be the admissible grammar obtaining by appending β to the end of $G_i(s_0)$. Then the following steps specify how to get G_{i+1} from G'_i.*

Case 1 *The pattern $\alpha\beta$ does not appear in two non-overlapping positions in the range of G'_i. In this case, G'_i is irreducible and hence G_{i+1} is equal to G'_i.*

Case 2 *The pattern $\alpha\beta$ appears in two non-overlapping positions in the range of G'_i and $I(i) = 0$. In this case, apply Reduction Rule 2 once if the pattern $\alpha\beta$ repeats itself in $G'_i(s_0)$, and Reduction Rule 3 once otherwise. The resulting grammar is irreducible and hence equal to G_{i+1}. The variable set of G_{i+1} is $S(j_{i+1})$ with $j_{i+1} = j_i + 1$, and the newly created production rule is $s_{j_i} \to \alpha\beta$.*

Case 3 *The pattern $\alpha\beta$ appears in two non-overlapping positions in the range of G'_i and $I(i) = 1$. In this case, apply Reduction Rule 2 followed by Reduction Rule 1 if the pattern $\alpha\beta$ repeats itself in $G'_i(s_0)$, and Reduction Rule 3 followed by Reduction Rule 1 otherwise. The resulting grammar is irreducible and hence equal to G_{i+1}. The variable set of G_{i+1} is the same as that of G_i with $j_{i+1} = j_i$, and $G_{i+1}(s_{j_{i+1}-1})$ is obtained by appending β to the end of $G_i(s_{j_i-1})$*

Proof. Refer to [15]. □

Example 3.11. We demonstrate the greedy irreducible transform on an example. Let $A = \{0, 1\}$ and $x = 1001110001000111000011111111000$. It is easy to see that the first three parsed phrases are 1, 0, and 0. The corresponding irreducible grammars G_1, G_2, and G_3 are given by $\{s_0 \to 1\}$, $\{s_0 \to 10\}$, and $\{s_0 \to 100\}$, respectively. Since $j_3 = 1$, the fourth parsed phrase is $x_4 = 1$. Appending the symbol 1 to the end of $G_3(s_0)$, we get an admissible grammar G'_3 given by $\{s_0 \to 1001\}$. G'_3 itself is irreducible, so none of Reduction Rules 1 to 5 can be applied and G_4 is equal to G'_3. Similarly, the fifth and sixth parsed phrases are $x_5 = 1$ and $x_6 = 1$, respectively; G_5 and G_6 are given respectively by $\{s_0 \to 10011\}$ and $\{s_0 \to 100111\}$. The seventh parsed phrase is $x_7 = 0$. Appending the symbol 0 to the end of $G_6(s_0)$, we get an admissible grammar G'_6 given by

$$s_0 \;\to\; 1001110$$

G'_6 is not irreducible any more since there is a non-overlapping repeated pattern 10 in the range of G'_6. At this point, only Reduction Rule 2 is applicable. Applying Reduction Rule 2 once, we get an irreducible grammar G_7 given by

$$\begin{aligned} s_0 &\to s_1011s_1 \\ s_1 &\to 10 \end{aligned}$$

Since the sequance from A represented by s_1 is not a prefix of the remaining part of x, the next parsed phrase is $x_8 = 0$. Appending the symbol 0 to the end of $G_7(s_0)$, we get an admisible grammar G_7' given by

$$\begin{aligned} s_0 &\rightarrow s_1011s_10 \\ s_1 &\rightarrow 10 \end{aligned}$$

G_7' is not irreducible. Applying Reduction Rule 2 once, which is the only applicable reduction rule at this point, we get a grammar G_7''

$$\begin{aligned} s_0 &\rightarrow s_s11s_s \\ s_1 &\rightarrow 10 \\ s_2 &\rightarrow s_10 \end{aligned}$$

In the above, the variable s_1 appears only once in the range of G_7''. APplying Reduction Rule 1 once, we get the irreducible grammar G_8:

$$\begin{aligned} s_0 &\rightarrow s_111s_1 \\ s_1 &\rightarrow 100 \end{aligned}$$

From G_7 to G_8, we have applied Reduction Rule 2 followed by Reduction Rule 1. Based on G_8, the next two parsed phrases are $x_9 = 0$ and $x_{10}x_{11}x_{12} = 100$, respectively. The irreducible grammar G_9 is given by

$$\begin{aligned} s_0 &\rightarrow s_111s_10 \\ s_1 &\rightarrow 100 \end{aligned}$$

and the grammar G_{10} is given by

$$\begin{aligned} s_0 &\rightarrow s_111s_10s_1 \\ s_1 &\rightarrow 100 \end{aligned}$$

Note that from G_9 to G_{10}, we simply append the symbol s_1 to the end of $G_9(s_0)$ since the phrase $x_{10}x_{11}x_{12}$ is represented by s_1. The eleventh parsed phrase is $x_{13} = 0$. Appending 0 to the end of $G_{10}(s_0)$ and then applying Reduction Rule 2 once, we get G_{11}

$$\begin{aligned} s_0 &\rightarrow s_111s_2s_2 \\ s_1 &\rightarrow 100 \\ s_2 &\rightarrow s_10 \end{aligned}$$

The twelfth parsed phrase is $x_{14} = 1$ and G_{12} is obtained by simply appending 1 to the end of $G_{11}(s_0)$. The thirteen parsed phrase is $x_{15} = 1$. Appending 1 to the end of $G_{12}(s_0)$ and then applying Reduction Rule 2 once, we get G_{13}

$$\begin{aligned} s_0 &\rightarrow s_1s_3s_2s_2s_3 \\ s_1 &\rightarrow 100 \\ s_2 &\rightarrow s_10 \\ s_3 &\rightarrow 11 \end{aligned}$$

The fourteen parsed phrase is $x_{16}x_{17}x_{18}x_{19} = 1000$, which is represented by s_2. Appending s_2 to the end of $G_{13}(s_0)$ and then applying Reduction Rule 2 followed by Reduction Rule 1, we get G_{14}

$$\begin{aligned} s_0 &\rightarrow s_1 s_3 s_2 s_3 \\ s_1 &\rightarrow 100 \\ s_2 &\rightarrow s_1 0 \\ s_3 &\rightarrow 11 s_2 \end{aligned}$$

The fifteenth parsed phrase is $x_{20} = 1$, and G_{15} is obtained by appending the symbol 1 to the end of $G_{14}(s_0)$. The sixteenth parsed phrase is $x_{21} = 0$. Appending the symbol 1 to the end of $G_{15}(s_0)$ and then applying Reduction Rule 3 once, we get G_{15}

$$\begin{aligned} s_0 &\rightarrow s_1 s_3 s_2 s_3 s_4 \\ s_1 &\rightarrow 100 \\ s_2 &\rightarrow s_1 0 \\ s_3 &\rightarrow s_4 s_2 \\ s_4 &\rightarrow 11 \end{aligned}$$

The seventeenth parsed phrase is $x_{22}x_{23} = 11$ and G_{17} is obtained by appending s_4 to the end of $G_{16}(s_0)$. The final parsed phrase is $x_{24} \cdots x_{29} = 111000$ and G_{18} is obtained by appending s_3 to the end of $G_{17}(s_0)$. In summary, the string x is parsed into {1, 0, 0, 1, 1, 1, 0, 0, 0, 100, 0, 1, 1, 1000, 1, 1, 11, 111000} and transformed into the irreducible grammar G_{18}

$$\begin{aligned} s_0 &\rightarrow s_1 s_3 s_2 s_3 s_4 s_4 s_3 \\ s_1 &\rightarrow 100 \\ s_2 &\rightarrow s_1 0 \\ s_3 &\rightarrow s_4 s_2 \\ s_4 &\rightarrow 11 \end{aligned}$$

□

Kieffer and Yang proposed three algorithms[4] based on the greedy irreducible grammar transform: a hierarchical algorithm, a sequential algorithm, an an improved sequential algorithm. Each of the algorithms uses the irreducible grammar transform to construct the irreducible grammar G_t which is encoded by the arithmetic coding with dynamic alphabet.

The following theorem gives the redundancy estimates for the hierarchical, the sequential, and the improved sequential algorithms.

[4]Prior to that, they had sketched yet another algorithm based on enumerative coding (see [23] for further details) in [16]. However, this algorithm is less intuitive and not very efficient.

Theorem 4. *The worst case redundancies of the hierarchical, the sequential, and the improved sequential algorithms, among all individual sequences $x \in A^+, |x| = n$, are upper bounded by*

$$c\frac{\log\log n}{\log n}$$

where c is a constant.

Proof. Refer to [15]. □

It follows from simulation results[5] in [15] that the hierarchical algorithm is greatly outperformed by both the sequential and the improved sequential algorithms. Moreover, the sequential algorithms make it possible to parse and encode the input sequence simultaneously, as opposed to the hierarchical algorithm which encodes the input sequence only after it was completely processed. The improved sequential algorithm yields the best results, but is the most difficult to implement efficiently.

In the following sections, we describe the principles of the three algorithms.

3.6.1.5 Hierarchical algorithm

Let $x \in A^+$ be a sequence to be compressed. Let G_t be the final irreducible grammar for x furnished by the irreducible grammar transform. In the hierarchical algorithm, a zero order arithmetic code with a dynamic alphabed is used to encode G_t (or its equivalent form). After receiving the binary codeword, the decoder recovers G_t (or its equivalent form) and reconstructs x from it.

To illustrate the operation of the hierarchical algorithm, we refer to Example 3.11. The final irreducible grammar for the sequence x in Example 3.11 is G_{18}. To encode G_{18} efficiently, it is first converted into its canonical form G_{18}^g given by

$$\begin{aligned} s_0 &\rightarrow s_1 s_2 s_3 s_2 s_4 s_4 s_2 \\ s_1 &\rightarrow 100 \\ s_2 &\rightarrow s_4 s_3 \\ s_3 &\rightarrow s_1 0 \\ s_4 &\rightarrow 11 \end{aligned}$$

Note that G_{18}^g and G_{18} still represent the same sentence. The difference between G_{18}^g and G_{18} is that the following property holds for G_{18}^g, but not for G_{18}:

(c.1) If we read $G_{18}^g(s_i)$ from left to right and from top ($i = 0$) to bottom ($i = 4$), then for any $j \geq 1$, the first appearance of s_j always precedes that of s_{j+1}.

[5] There was no implementation of the individual algorithms available by the time when [15] was published.

To encode G_{18}^g, we concatenate $G_{18}^g(s_0), G_{18}^g(s_1), \ldots, G_{18}^g(s_4)$ in the indicated order, inserting symbol e at the end of $G_{18}^g(s_0)$, and for any $j \geq 1$ satisfying $|G_{18}^g(s_i)| > 2$, inserting symbol b at the beginning of $G_{18}^g(s_i)$ and symbol e at the end of $G_{18}^g(s_i)$, where symbols b and e are assumed not to belong to $S \cup A$. This gives rise to the following sentence from the alphabet $S \cup A \cup \{b, e\}$:

$$s_1 s_2 s_3 s_2 s_4 s_4 s_2 e b 100 e s_4 s_3 s_1 011 \tag{3.1}$$

Since G_{18}^g satisfies the property **(c.1)**, we can take advantage of this. Let s be a symbol which is not in $S \cup A \cup \{b, e\}$. For each $i \geq 1$, replace the first occurrence of s_i in the sequence 3.1 by s:

$$ssss_2 ss_4 s_2 e b 100 e s_4 s_3 s_1 011 \tag{3.2}$$

This sequence will be called the sequence generated from G_{18} (or its canonical form G_{18}^g). We can get the sequence 3.1 from 3.2 by simply replacing the ith s in 3.2 by s_i. And from 3.1, it is easy to reconstruct the grammar G_{18}^g.

To compress G_{18}^g (or x), we encode the sequence generated from G_{18} using a zero order arithmetic coder with dynamic alphabet. We associate each symbol $\beta \in S \cup A \cup \{b, e, s\}$ with a counter $c(\beta)$. Initially, $c(\beta)$ is set to 1 if $\beta \in A \cup \{b, e, s\}$ and 0 otherwise. The initial alphabet used by the arithmetic coder is $A \cup \{b, e, s\}$. Each symbol β in the sequence generated from G_{18} is encoded as follows:

1. Encode β by using the probability

$$c(\beta) / \sum_{\alpha} c(\alpha)$$

   where the summation is taken over $A \cup \{b, e, s\} \cup \{s_1, \ldots, s_i\}$, and i is the number of times that s occurs before the position of this β.

2. Increase the counter $c(\beta)$ by 1.

3. If $\beta = s$, increase the counter $c(s_(i+1))$ from 0 to 1, where i is defined in Step 1.

Repeat this procedure until the whole sequence x is processed and encoded.

3.6.1.6 Sequential algorithm

In the sequential algorithm, the data sequence x is encoded sequentially by using a zero order arithmetic code with a dynamic alphabet to encode the sequence of parsed phrases $x_1,\ x_2 \cdots x_{n_2},\ \ldots,\ x_{n_{t-1}+1} \cdots x_{n_t}$. We associate each symbol $\beta \in S \cup A$ with a counter $c(\beta)$. Initially, $c(\beta)$ is set to 1 if $\beta \in A$ and 0 otherwise. At the beginning, the alphabet used by the arithmetic code is A. The first parsed phrase x_1 is encoded using a probability $c(x_1) / \sum_{\beta \in A} c(\beta)$. The the counter $c(x_1)$ increases by 1.

Suppose that $x_1, x_2 \cdots x_{n_2}, \ldots, x_{n_{i-1}+1} \cdots x_{n_i}$ have been parsed off and encoded and that all corresponding counters have been updated. Let G_i be the corresponding irreducible grammar for $x_1 \cdots x_{n_i}$ with the variable set $S(j_i) = \{s_0, s_1, \ldots, s_{j_i-1}\}$. Let $x_{n_i+1} \cdots x_{n_{i+1}}$ be parsed off by the greedy sequential grammar transform and represented by $\beta \in \{s_1, \ldots, s_{j_i-1}\}$. Encode β and update the relevant counters according to the following steps:

1. The alphabet used at this point by the arithmetic code is $\{s_1, \ldots, s_{j_i-1}\} \cup A$. Encode $x_{n_i+1} \cdots x_{n_{i+1}}$ by using the probability

$$c(\beta) / \sum_{\alpha \in S(j_i) \cup A} c(\alpha)$$

2. Increase $c(\beta)$ by 1.

3. Get G_{i+1} from the appended G_i as in the greedy irreducible grammar transform.

4. If $j_{i+1} > j_i$, i.e G_{i+1} includes new variable s_{j_i}, increase the counter $c(s_{j_i})$ by 1.

Repeat this procedure until the whole sequence x is processed and encoded.

3.6.1.7 Improved sequential algorithm

The encoding of the sequence of parsed phrases in the sequential algorithm does not utilize the structure of the irreducible grammar $G_i, 1 \leq i \leq t$. Since G_i is known to the decoder before encoding the $(i+1)$th parsed phrase, the structure of G_i can be used as context information to reduce the codebits for the $(i+1)$th parsed phrase.

Each symbol $\gamma \in S \cup A$ is associated with two lists $L_1(\gamma)$ and $L_2(\gamma)$. The list $L_1(\gamma)$ consists of all symbols $\eta \in S \cup A$ such that the following properties are satisfied:

(d.1) The pattern $\gamma\eta$ appears in the range of G_i.

(d.2) The pattern $\gamma\eta$ is not the last two symbols of $G_i(s_0)$.

(d.3) There is no variable s_j of G_i such that $G_i(s_j)$ is equal to $\gamma\eta$.

The list $L_2(\gamma)$ consists of all symbols $\eta \in S \cup A$ such that properties **(d.1)** and **(d.2)** hold. Let α be the last symbol of $G_i(s_0)$. Let the $(i+1)$th parsed phrase $x_{n_i+1} \cdots x_{n_{i+1}}$ be represented by $\beta \in \{s_1, \ldots, s_{j_i-1}\} \cup A$. Recall from Section 3.6.1.4 that $I(i) = 0$ is equal to 0 if G_i is equal to the appended G_{i-1}, and 1 otherwise. From Theorem 3, it follows that when $I(i) = 0$ and $I(i+1) = 0$, the symbol β appears in the list $L_1(\alpha)$, and hence one can simply send the index of β in the list $L_1(\alpha)$ to the decoder. When $I(i) = 1$ and $I(i+1) = 1$, β is the only element in the list $L_1(\alpha)$ and thus no information has to be sent to the decoder. Therefore, if we transmit the bit $I(i+1)$ to the decoder, we can use the bit $I(i+1)$ and the structure of G_i to improve the encoding of β.

In addition to the counters $c(\gamma), \gamma \in S \cup A$, improved sequential algorithm defines new counters $c(0,0)$, $c(0,1)$, $c(1,0)$, $c(1,1)$, and $\hat{c}(\gamma)$. The counters $c(0,0)$, $c(0,1)$, $c(1,0)$, and $c(1,1)$ are used to encode the sequence $\{I(i)\}_{i=1}^{t}$. Initially, they are all equal to 1. The $(i+1)$th parsed phrase is encoded by the counters $\hat{c}(\gamma)$ whenever $I(i) = 0$ and $I(i+1) = 1$, and by by the counters $c(\gamma)$ whenever $I(i+1) = 0$. As in the case of $c(\gamma)$, initially $\hat{c}(\gamma)$ is 1 if $\gamma \in A$ and 0 if $\gamma \in S$.

The first three parsed phrases are encoded as in the sequential algorithm since they are x_1, x_2, and x_3. Also, $I(1)$, $I(2)$, and $I(3)$ are all 0 and hence there is no need to encode them. Starting with the fourth phrase, we first encode $I(i+1)$, and then use $I(i+1)$ as side information and the structure of G_i as context information to encode the $(i+1)$th parsed phrase.

Suppose that $x_1, x_2 \cdots x_{n_2}, \ldots, x_{n_{i-1}+1} \cdots x_{n_i}$ have been parsed off and encoded and that all corresponding counters have been updated. Let G_i be the corresponding irreducible grammar for $x_1 \cdots x_{n_i}$ with the variable set $S(j_i) = \{s_0, s_1, \ldots, s_{j_i-1}\}$. Let α be the last symbol of $G_i(s_0)$. Let $x_{n_i+1} \cdots x_{n_{i+1}}$ be parsed off by the greedy sequential grammar transform and represented by $\beta \in \{s_1, \ldots, s_{j_i-1}\}$. Encode $I(i+1)$ and β, and update the relevant counters according to the following steps:

1. Encode $I(i+1)$ by using the probability

$$\frac{c(I(i), I(i+1))}{c(I(i), 0) + c(I(i), 1)}$$

2. Increase $c(I(i), I(i+1))$ by 1.

3. If $I(i+1) = 0$, encode β by using the probability

$$c(\beta) / \sum_{\gamma \in S(j_i) \cup A - L_2(\alpha)} c(\gamma)$$

and increase $x(\beta)$ by 1. If $I(i) = 0$ and $I(i+1) = 1$, encode β by using the probability

$$\hat{c}(\beta) / \sum_{\gamma L_1(\alpha)} \hat{c}(\gamma)$$

and then increase $\hat{c}(\beta)$ by 1. On the other hand, if $I(i) = 1$ and $I(i+1) = 1$, no information is sent since $L_1(\alpha)$ contains only one element and the decoder knows what β is.

4. Get G_{i+1} from the appended G_i as in the greedy irreducible grammar transform. Update $L_1(\gamma)$ and $L_2(\gamma)$ accordingly, where $\gamma \in S(j_{i+1}) \cup A$.

5. If $j_{i+1} > j_i$, i.e. G_{i+1} includes new variable s_{j_i}, increase both $c(s_{j_i})$ and $\hat{c}(s_{j_i})$ by 1.

Repeat this procedure until the whole sequence x is processed and encoded.

3.6.2 Sequitur

In [24, 25], Nevill-Manning and Witten presented a syntactical compression method that is in many ways similar to the grammar-based codes discussed in the previous section. Their algorithm, called Sequitur, sequentially generates a context-free grammar that uniquely represents the input string. Compared to the grammar-based coding, the algorithm is rather simple and can be stated concisely in the form of two constraints on the generated context-free grammar:

1. No pair of adjacent smbols appears more than once in the grammar.
2. Every production rule is used more than once.

Property 1 says that every digram (i.e. a pair of two adjacent symbols) in the grammar is unique, and is referred to as *digram uniqueness*. Property 2 ensures that the production rules are useful, and is called *rule utility*.

Sequitur processes the input data incrementally, and its operation consists of ensuring that both constraints hold. When a new symbol is observed, it is appended to the root production rule of the grammar. If the new symbol causes the digram uniqueness constraint to be violated, a new production rule is formed. In the case that the rule utility constraint is violated, the useless production rule is deleted.

Example 3.12. We illustrate the operation of Sequitur on the string **abcdbcabcd**. Suppose that the string **abcdbcabc** has been parsed off. The grammar representing this string looks as follows:

$$\begin{aligned} S &\rightarrow B\mathbf{da}B \\ A &\rightarrow \mathbf{bc} \\ B &\rightarrow \mathbf{a}A \end{aligned}$$

After appending the symbol **d**, we get the grammar given by

$$\begin{aligned} S &\rightarrow B\mathbf{da}Bd \\ A &\rightarrow \mathbf{bc} \\ B &\rightarrow \mathbf{a}A \end{aligned}$$

The digram $B\mathbf{d}$ appears two times in the grammar, and the digram uniqueness constraint is violated. Therefore a new production rule C is introduced:

$$\begin{aligned} S &\rightarrow CAC \\ A &\rightarrow \mathbf{bc} \\ B &\rightarrow \mathbf{a}A \\ C &\rightarrow B\mathbf{d} \end{aligned}$$

This time, the rule utility constraint is violated since the production rule B is used only once. Therefore B is removed from the grammar, and its right-hand side is substituted in the one place where it occurs:

$$\begin{aligned} S &\rightarrow CAC \\ A &\rightarrow \mathbf{bc} \\ C &\rightarrow \mathbf{a}A\mathbf{d} \end{aligned}$$

□

After the grammar is formed, it is encoded using a so called *implicit encoding* technique. In this technique, the root production rule is encoded. When a variable is encountered in the root production rule, it is treated in three different ways depending on how many times it has been seen. The first time it occurs, its contents are sent. On its second occurrence, a pointer that identifies the contents of the rule is sent. The pointers are similar to that used in LZ77 (see Section 3.4.1) and consist of an offset from the beginning of the root production rule, and the length of the match. The decoder numbers production rules in the order in which they are received, and the encoder keeps track of this numbering. On the third and subsequent occurrences of the variable, the number of the production rule is used to identify it.

Example 3.13. The grammar from Example 3.12 is implicitly encoded as $\mathbf{abcd}(1,2)(0,3)$. Because both production rules A and C only occur twice in the grammar, no variables appear in the encoding. The encoded sequence represents the root production rule which consists of two instances of production rule C and one instance of A. The first symbol **a** is encoded normally. The first time production rules C and A are encountered, their contents are sent. After C has been encoded, the production rule A occurs for the second time, and the pointer $(1,2)$ is sent. The first element of the pointer is the distance from the start of the sequence to the start of the first occurence of **bc**. The second element of the pointer is the length of the repetition. After receiving the pointer, the decoder forms a production rule $A \rightarrow \mathbf{bc}$, and replaces both instances of **bc** in the sequence with A. The second occurrence of production rule C is encoded by the pointer $(0,3)$. The repetition starts at the beginning of the sequence, and continues for 3 symbols ($\mathbf{a}A\mathbf{d}$). □

In [15], Kieffer and Yang showed that the code generated by the Sequitur algorithm is not universal, as opposed to the grammar-based codes. The main reason is that the grammars generated by Sequitur do not satisfy the property **(b.3)** (see page 35), i.e. they are not irreducible.

Chapter 4

Existing XML-conscious compressors

We are aware of several XML-conscious compressors. In this chapter, we discuss the main principles in these tools, and summarize their performance in the context of other XML-conscious compressors as well as the general-purpose compressors. Because not all of the compressors have technical papers or documentation, we were not able to present the exact details on the compression techniques used in some occasions.

4.1 XMill

XMill [20] is an XML compressor based on Gzip, which can compress about twice as good, and at about the same speed. It allows to combine existing compressors in order to compress heterogeneous XML data. Further, it is extensible with user-defined compressors for complex data types, such as DNA sequences, etc.

```
<book>
  <title lang="en">Views</title>
  <author>Miller</author>
  <author>Tai</author>
</book>
```

Figure 4.1: Sample XML document

XMill parses XML data with a SAX parser, and transforms it by splitting the data into three types of containers: one container for the element and attribute symbols, one for the document tree structure, and several containers for storing the character data. By default, each element or attribute is assigned one data container. XMill employs a path processor that is driven by so called container expressions. The container expressions are based on the XPath language [32], and allow experienced users to group the data within a certain

set of elements into one container to improve compression efficiency. In the output file, the individual containers are compressed using Gzip.

XMill applies three principles to compress XML data:

Separate structure from data. The structure, represented by XML tags and attributes, and the data are compressed separately.

XMill uses numeric tokens to represent the XML structure. Start-tags are dictionary encoded, i.e. assigned an integer value, while all end-tags are replaced by the token `/`. Data values are replaced with their container number. When complete document is processed, the token table, the structure container and the data containers are compressed using Gzip. The tokens are represented as integers with 1, 2, or 4 bytes; tags and attributes are positive integers, `/` is 0, and container numbers are negative integers.

To illustrate the tokenization of the structure, consider the sample XML document in Figure 4.1. After the document is processed, the structure will be tokenized as:

```
T1 T2 T3 C3 / C4 / T4 C5 / T4 C5 / /
```

and the following dictionary is created: `book = T1`, `title = T2`, `@lang = T3`, `author = T4`. Data values are assigned containers `C3`, `C4`, and `C5` depending on their parent tag.

Group data items with related meaning. The data items are grouped into containers, which are compressed separately. XMill groups the data items based on the element type, but this can be overridden through the container expressions. By grouping similar data items, the compression can improve substantially.

The container expression describe the mappings from paths to containers. Consider the following regular expressions derived from XPath:

```
e ::= label | * | # | e1/e2 | e1//e2 | (e1|e2) | (e)+
```

Except for `(e)+` and `#`, all are XPath constructs: `label` is either `tag` or an `@attribute`, `*` denotes any tag or attribute, `e1/e2` is concatenation, `e1//e2` is concatenation with any path in between, and `(e1|e2)` is alternation. To these constructs, `(e)+` has been added, which is the strict Kleene-Closure. The construct `#` stands for any tag or attribute (much like `*`), but each match of `#` will determine a new container.

The container expression has the form `c ::= /e | //e`, where `/e` matches `e` starting from the root of the XML tree while `//e` matches `e` at arbitrary depth of the tree. `//*` is abbreviated by `//`.

Example 4.1. `//Name` creates one container for all data values whose paths end in `Name`. `//Person/Title` creates a container for all `Person`'s titles. `//#` creates a family of containers—one for each ending tag or attribute—and characterizes the default behavior of XMill. □

Apply semantic compressors to containers. Because the data items can be of different types (text, numbers, dates, etc.), XMill allows the users to apply different specialized

semantic compressors to different containers. At first, the items in the container are processed by the semantic compressor, and then they are passed to Gzip.

There are 8 different semantic compressors in XMill. These can be used to encode integers, enumerations and texts, or the sequences or the repetitions of them more efficiently. For example, positive integers are binary encoded as follows: numbers less than 128 use one byte; those less than 16384 use two bytes, otherwise they use four bytes. The most significant one or two bits describe the length of the sequence.

Semantic compressors are specified on the command line using the syntax `c=>s` where `c` is a container expression and `s` is a semantic compressor.

It is possible to write his own semantic compressor (for example, for encoding the DNA sequences) and link it into XMill. The list of semantic compressors can be extended by the users.

Under the default setting, XMill compresses 40%-60% better than Gzip. With the user assistance (grouping related data, applying semantic compressors), it is possible to further improve the compression by about 10%.

The main disadvantage of XMill is that it scatters parts of the documents, making incremental processing impossible.

4.2 XMLZip

Java-based XMLZip is a creation of XML Solutions [34]. It operates in a rather interesting way. The compression is driven by the level parameter l. Based on this parameter, XMLZip processes the document using the DOM interface, and breaks the structural tree into multiple components: a root component containing the elements up to the level l, and one component for each of the remaining subtrees starting at level l. The root component is modified by adding references to the subtrees, and the individual components are then compressed using the Java Zip/Deflate library (which uses a variant of the LZSS method).

```
<root>
  <child id="1">
    ...
  </child>
  <child id="2">
    ...
  </child>
</root>
```

Figure 4.2: Sample XML document

Consider the XML document in Figure 4.2. Suppose that $l = 2$. XMLZip splits

the original document into three components, as displayed in Figure 4.3. In the root component, <xmlzip> tags are inserted to reference the detached subtrees. After that, the individual components are compressed.

```
<root>                  <child id="1">      <child id="2">
  <xmlzip id="1"/>        ...                 ...
  <xmlzip id="2"/>      </child>            </child>
</root>
```

Figure 4.3: Decomposition of the XML document with $l = 2$

The compression efficiency depends on the value of l. In most occasions, increasing l causes the performance to deteriorate, since the redundancies across the separated subtrees cannot be used in the compression.

In a comparison to other XML compressors, XMLZip yields considerably worse results. It is often outperformed even by the ordinary text compressors, such as Gzip. However, the main benefit of XMLZip is that it allows limited random access to the compressed XML documents without storing the whole document uncompressed or in the memory. Only the portion of the XML tree that needs to be accessed is uncompressed. It is possible to implement a DOM-like API to control the amount of memory required, or to speed up the access for queries, for example.

XMLZip can only be run on entire XML documents, and therefore the compression is off-line.

4.3 XMLPPM

In [4], the PPM compression has been adapted to compress XML. The compressor—called XMLPPM—uses so called *multiplexed hierarchical modeling* (MHM) for modeling the structure of XML. In MHM, several PPM models are multiplexed together, and switches among them are performed based on the syntactic context supplied by the parser.

XMLPPM uses four PPM models: one for element and attribute names, one for element structure, one for attributes, and one for character data. Each model maintains its own state but all share the access to one underlying arithmetic coder. Element start tags, end tags, and attribute names are dictionary encoded using numeric tokens. Whenever a new symbol is encountered, the encoder sends the symbol name and the decoder enters it to the corresponding dictionary. Some tokens are reserved, and are used to encode the events such as the start of the character data, the element end tag, etc.

To demonstrate the operation of XMLPPM, we encode the following XML fragment:

```
<elt att="abcd">XYZ</elt>
```

Suppose that the tag `elt` has been seen before, and is represented by the token `10`, but the attribute `att` has not, and the next available token for attribute names is `0D`. The state of the individual MHM models after processing the XML fragment is shown in Table 4.3.

	<elt	att=		"abcd"	>	XYZ		</elt>
Elt:	10					FE		FF
Att:		10 0D		asdf00	10 FF			
Char:							10 XYZ 00	
Sym:			att00					

Table 4.4: MHM models after processing the example XML fragment

Notice the token 10 that has been "injected" into the attribute and character data models. This token is not encoded; instead, it is used to indicate the cross-class sequential dependencies within the XML document. A common case for these dependencies is a strong correlation between the enclosing element tag and the enclosed data. Using the "injection" mechanism, these correlation can be exploited by MHM.

The authors of XMLPPM have also implemented a variant of MHM that uses PPM* instead of PPM, which they call MHM*. Compared to MHM, MHM* performs slightly better on average, but is considerably slower. On structured documents, MHM performs much worse (about 20%-40%) than MHM*; on the other hand, MHM* is worse on textual documents.

XMLPPM (using either MHM or MHM*) compresses extremely well, outperforming most of the concurrent compressors. The compression is on-line, and therefore the XML data can be processed incrementally.

4.4 Millau

Millau [10] is an on-line XML compression method that is suitable for compression and streaming of small XML documents (smaller than 5 kilobytes). Millau can make use of the associated schema (if available) in the compression of the structure.

The encoding is based on the Wireless Binary XML format (WBXML) proposed by the Wireless Application Protocol Forum [31] which losslessly reduces the size of XML documents. This method uses a table of tokens to encode the XML tags and the attribute names. Some tokens are reserved, and are used to indicate events such as the character data, the end of element, etc. The meaning of a particular token is dependent on the context in which it is used. There are two basic types of tokens: global tokens and application tokens. Global tokens are assigned to fixed set of codes in all contexts and are unambiguous in all situations. Global codes are used to encode inline data (such as strings, entities, etc.) and to encode a variety of miscellaneous control functions. Application tokens have a context-dependent meaning and are split into two overlapping *code spaces*: the tag code

space and the attribute code space. The tag code space represents specific tag names and the attribute code space comprises of attribute-start token and attribute-value token.

Since the set of tags and attributes is known in advance in the WAP protocol, the table is fixed and does not have to be contained in the encoded data. The output data is a stream of tokens and uncompressed character data. In this stream, the structure of the original XML document is preserved.

Millau improves on the WBXML scheme, making it possible to compress the character data: the structure and the character data are separated into two streams, and the character data stream is compressed using conventional text compressors. In the structure stream, special tokens are inserted to indicate the occurrences of compresssed data.

Since the set of elements and attributes is not known in advance in the case of ordinary XML documents, Millau sets out a strategy for building the token table. If the DTD exists, Millau constructs the table based upon it; otherwise, the document is pre-parsed and the tokens are assigned to the encountered elements and attributes. The table of tokens is contained in the encoded data.

An XML parser for processing the Millau streams is implemented using both DOM and SAX. Because binary tokens are processed—instead of strings in the case of uncompressed XML documents—, the parser usually operates very fast.

Although Millau is outperformed by the traditional text compression algorithms on large XML files, it achieves better compression for file sizes between 0-5 kilobytes, which is the typical file size for eBusiness transactions, such as orders, bill payments, etc.

4.5 Other compressors

We are aware of two more XML compressors: XGrind and XML-Xpress. XGrind is a compressor that makes direct queries of the compressed data possible. XML-Xpress is a commercial application that is schema-aware and is reported to achieve extremely high compression ratios. In subsequent paragraphs, the principles of the two compressors are briefly discussed.

XGrind [28] is a query-friendly XML compressor that encodes the data as a mixture of numeric tokens that represent the structure (the mechanism of tokens is similar to that of the XMill compressor) and of compressed character data. The structure of the original document is preserved in the output. To make querying of the compressed data possible, the character data is encoded using non-adaptive Huffman coding. Thanks to that, it is possible to locate occurrences of a given string in the compressed document, without decompressing it. The non-adaptive coding requires two passes over the input data: in the first pass, the statistics are gathered, and in the second pass, the document is compressed.

Commercially available XML-Xpress [14] is a schema-aware compressor that can work either with the DTD or XML Schema [33]. When the schema is known to XML-Xpress, XML tags can be encoded very efficiently. For example, if an element is defined in the schema as having only one of two sub-elements `(A|B)`, only a binary decision needs to be included in the encoded file to determine which of `A` or `B` is present. The compression can

be further improved by using XML Schema instead of DTD, because the information about the data types of the element data is utilized in the compression. In [14], compression ratios exceeding 30 : 1 are reported.

The main disadvantage of XML-Express is that it is primarily a schema-specific compressor, and thus the above average compression ratios are dependent on the presence of a known schema. In the absence of such a schema, XML-Express uses a general-purpose compressors, and the outstanding compression performance is lost.

Chapter 5

Principles of the proposed XML compression scheme

In Chapter 4, a number of XML-conscious compressors were discussed. Several compression techniques have been adopted in these tools, and with varying results. Unfortunately, it is not easy to judge which compressor is the best, or which is the worst. Such a ranking would be necessarily unfair, since each of the compressors that we are aware of differs from the others in some way, and is suited for different purposes. For example, some of the compressors allow transparent access to the compressed data via the SAX or DOM interfaces (XMLZip or Millau, for instance), but achieve poor compression results. On the other hand, there are XMill and XMLPPM that compress very well, but there is no way to parse the compressed data unless it is decompressed first. Also, some compressors work incrementally and allow on-line processing of the data (XMLPPM), while some are off-line by their design (XMill). There is no clear winner—no "ideal" XML compressor—, and one has to formulate his requirements first before he picks and applies the compressor that seems to be the most suitable.

We believe that there are still many paths to be explored in the field of XML data compression. In this work, we present a syntactical compression scheme that is based on probabilistic modeling of XML structure. It does not need the DTD since it infers all the necessary information directly from the input XML data. Moreover, it works incrementally, making on-line compression and decompression possible. Transparent parsing of the compressed data using the SAX interface is possible.

In the following text, the main principles and features of our compression scheme are discussed.

Syntactical compression. Our approach is based on the observation that XML has a fairly restrictive context-free nature. Indeed, the structure of XML documents, as defined in the DTD, can be described by a special form of a context-free grammar. These grammars have been extensively studied (for example in [9]) and are sometimes referred to as *XML grammars*. Consider for example the (rather simplistic) DTD:

```
<!DOCTYPE a [
    <!ELEMENT a ((a|b), (a|b))>
    <!ELEMENT b (b)*>
]>
```

According to [9], this DTD can be rewritten to the corresponding XML grammar:

$$\begin{aligned} X_a &\rightarrow a(X_a|X_b)(X_a|X_b)\overline{a} \\ X_b &\rightarrow b(X_b)^*\overline{b} \end{aligned}$$

It can be seen that XML grammars correspond to DTDs in a natural way, and vice versa. This led us to the thought of employing some kind of grammar-inferring technique to compress the XML data.

The syntactical compression techniques that caught our attention were the Sequitur algorithm by Nevill-Manning and Witten (see Section 3.6.2 or [24, 25]), and grammar-based codes recently proposed by Kieffer and Yang (see Section 3.6.1 or [16, 15]). Both schemes work in a similar manner: they parse the input data and infer a context-free grammar that uniquely describes it. To compress the data, the grammar is encoded using an arithmetic code.

We have experimented with the implementation of Sequitur that was designed to be used on text data, and we find its results on XML to be very promising. In many occasions, it greatly outperformed other general-purpose text compressors such as Gzip, demonstrating its ability to identify the hierarchical structure of the input.

The main difference between Sequitur and the grammar-based codes is in the way how the grammar is constructed and encoded. In both schemes, the grammar is constructed incrementally during the processing of the data. However, the grammar-based coding scheme allows us to parse the data and to encode the grammar simultaneously, whereas in Sequitur, complete grammar has to be formed first before it is encoded.

The constraints on the generated grammar are more restrictive in the case of the grammar-based codes. The properties of the generated grammar are influential in the performance of both schemes: while the grammar-based codes are proven to be universal for wide variety of sources, the Sequitur code is not guaranteed to be universal code at all. For further details, please refer to [15].

We decided to employ the grammar-based coding in our scheme, although we were not aware of any previous implementation of it. This step into the *terra incognita* was motivated by the results observed with the Sequitur algorithm, and by our belief in the potentials of the grammar-based codes. Moreover, it was also an interesting test on how well this new and promising technique performed.

Remark. The efficient implementation (in the sense of time and memory sparing issues) of the grammar-based coding scheme is a complicated task in itself, and requires many

practical problems to be solved. Our goal was rather to test the behavior of the grammar-based compression on semi-structured data than to present a "perfect" implementation.

Analysis of SAX events. Exalt uses an underlying SAX parser to access the XML content.[1] The event-driven nature of the SAX interface has several advantages over the standard DOM interface: most importantly, it is not necessary to store the whole document in the system memory. This is crucial for the compression of large documents. Also, thanks to the the sequential nature of the SAX interface, the data can processed incrementally, which makes on-line compression possible.

A typical SAX parser "emits" a stream of SAX events that are processed by the application. In our compression scheme, the SAX events serve as the basis for the modeling of XML structure (see below).

One possible problem with the SAX interface is that, in some occasions, the data presented by SAX parser may not be the same as the original data. For example, consider the following fragment of character data: "`rhythm &amp; blues`". The data that the SAX parser delivers to the application is: "`rhythm & blues`". Notice that the standard entity reference `&amp;` has been replaced by the character `&`. While this is no real problem for the ordinary XML processing (in fact, it is an expected behavior to replace the reserved entity references with their equivalents), it has important consequences for the XML compression. If we store the character `&` "as is", the resulting XML document may not be well-formed. The partial solution is to replace each occurrence of this character by the corresponding entity reference every time it would cause the resulting XML document not to be well-formed. The consequences of this are not far to seek: if we compress an XML document using the SAX event processing, after the decompression we may get a document that is not identical to the original, since some characters in the original document may be incorrectly replaced by the reserved entities that represent them.

Accessing the XML document via the SAX interface requires more operations to be performed than reading the document directly. It is therefore obvious that general-purpose text compressors run faster. However, this handicap shows to be negligible in most occasions.

Probabilistic modeling of XML structure. XML data contain a lot of redundant information that is present in the XML structure. For the ordinary text compressors, this type of redundancy is often difficult to discover, which is the main reason why they yield only suboptimal results. By discovering and utilizing the structural redundancy, one can substantially improve the compression efficiency.

Several techniques of structure modeling have been adopted in the XML-conscious compressors. For example, XMill and XMLPPM, which are both based on existing text compressors, use numeric tokens to represent the XML structure. The tokenized representation of the structure exposes the redundancy much more, and is therefore better suited for compression using general-purpose text compressors.

[1] We use the Expat XML parser [8].

We have proposed two modeling strategies which we call the *simple modeling* and the *adaptive modeling*, respectively. In both cases, numeric tokens are used to represent the structure. The input XML data is transformed into a sequence of characters and numeric tokens, and then passed to the grammar-based coder. The numeric codes represent the units of the XML structure that contain a lot of redundant information, such as repeated occurrences of known tags and attributes, or the end-of-element tags. Furthermore, some tokens are reserved to encode the events such as comments, processing instructions, entity declarations, etc.

The simple modeling does not bring anything really novel to the XML data compression. However, due to its simplicity, it can be used as a suitable foundation for future enhancements. One of the ways how to improve on it is demonstrated in the adaptive modeling. In the adaptive modeling, we try to learn as much as possible about the structure of the input document. This knowledge can be used for the prediction of the subsequent data. In most occasions, the elements have fairly regular structure—and once we discover this structure, we are able to predict the "behavior" of the document in the future. As a result, the amount of the data to be compressed can be substantially reduced.

The adaptive modeling gives us resources that allow us to measure the complexity of the XML documents. We define so called *structural entropy* which characterizes the amount of the information present in the structure of the element. By weighting the structural entropies of the individual elements of the document, the complexity of the entire document can be measured.

The detailed description of both the simple modeling and the adaptive modeling can be found in Chapter 6.

Remark. We wanted our compression scheme to be on-line, making incremental processing of the data possible. The document transformation that we use is similar to that of the XMill compressor, except that it is not container-based. The character data and the tokenized structure are compressed together. We realize that this approach somewhat "mixes apples and oranges", and probably hinders better results. In XMLPPM, for example, the data is compressed using four multiplexed PPM models, each devoted to different class of data (element and attribute names, attribute values, character data, and structure). Multiplexing of several models is probably the right way, and we wanted to go that direction initially. Sadly, it soon became obvious that this approach is not applicable in our compression scheme. While switching among the PPM models is quite easy, the nature of the greedy irreducible grammar transform makes it impossible for the grammar-based coding. If we used several grammars for representing different classes of data, it would be extremely difficult to keep them in a consistent state. The main reason is that the greedy irreducible grammar transform doesn't ensure that the data passed to it is processed. Actually, the input data is enqueued and only a small portion of it (if any) may be processed. Very often, there are production rules in the grammar that represent more than the enqueued data, and the greedy irreducible grammar transform waits until next data is supplied to see if these rules are applicable.

The independence on the DTD. We designed our compression scheme to be independent on the DTD. The reason is that while the information contained in the DTD can be used for extremely efficient compression of XML (as sketched in [19]), the use of the DTD brings many problems that need to be solved.

Probably the most important problem is that the DTD may not be always available. Often, it is located on a remote machine (which may be unaccessible), or the document simply does not have any DTD at all.

Other problem is how to ensure that the DTD is available also to the decoder. If we use some local DTD for the compression, and then copy the compressed file to another machine, we will be unable to decompress it there, because the DTD will be unaccessible to the decoder. To solve this, the DTD (probably compressed in some way) should be attached to the compressed data.

Because of the above mentioned problems, and the fact that there are many documents that do not have any associated schema, we decided not to rely on the DTD. However, the support for the schema in our compressor surely is a motivating subject for future research.

SAX event based decompression. Our decompressor can act as an ordinary SAX parser on the compressed data. It can read the compressed XML data and emit SAX events to the application. Actually, the decompressor can be divided into two components: one components decodes the stream of the compressed data and emits corresponding SAX events, and the second component uses these events for the reconstruction of the original XML document. Thanks to this approach, the SAX events can be redirected to the application in an natural fashion. The application can handle only selected SAX events (for example, the start-of-element and end-of-element events), and ignore the others.

The SAX interface of Exalt is similar to that of the Expat XML parser [8].

Support for various types of inputs and outputs. In most occasions, the data that is processed is stored in files. But we are not limited to use files only. In fact, it is possible to work with virtually any "device", such as the network or some database, for example. In our compression scheme, a higher level abstraction to inputs and outputs is introduced, called input/output devices (or IO devices, for short). At present, only files are supported, but programmers are allowed to create custom IO devices which can be transparently used by both the compressor and the compressor.

During the compression, the XML data is usually read from some input device and then written to some output device. We refer to this default interface as the PULL interface. In the PULL interface, the coder works with the input device by itself; once the compression starts, there is no way to stop it unless complete document is processed. In some occasions, a different interface may be useful. We call this the PUSH interface. In the PUSH interface, it is the application that supplies the data to the coder. The PUSH interface may be used for the compression of the XML data that is dynamically generated, for example.

Chapter 6

Modeling of XML structure

The modeling of XML structure plays an important role in the process of compression. Since there is a lot of redundant information arising from the structure of XML, the goal of such a modeling is to expose this redundancy and to improve the performance of the underlying compressor. We can also make use of the model of the structure to make predictions that may substantially reduce the amount of the data that has to be compressed. Thanks to that, the compresion performance may further increase.

We have devised and implemented two modeling strategies called the simple modeling and the adaptive modeling, respectively. While the first technique is quite simple and serves rather as a foundation for further enhancements, the latter improves on it and is much more sophisticated.

Both modeling strategies are adaptive and make on-line processing possible.

6.1 Structural symbols

In both modeling strategies, the structure and character data content are compressed together (there are no containers as in the case of XMill). The structure is represented—and distinguished from the character data—by numeric tokens. We call these tokens *structural symbols*. There are two types of structural symbols: *reserved symbols* and *identifiers*. The reserved symbols represent events such as new start-element tag or attribute, known start-element tag or attribute, end-element tag, start of CDATA section, etc. The identifiers are used in conjunction with the reserved symbols and complete the information in some situations. Very often, the identifiers act as indices to certain dictionaries.

The set of the reserved symbols is fixed, while the identifiers are assigned values in an adaptive way, during the processing of the input document. Therefore, reserved symbols and identifiers have to be encoded differently.

In the case of the reserved symbols, we made use of the fact that in the XML specification [30], the valid character data is defined such that there is a gap of characters with Unicode values `#00` to `#1F` (except the whitespace characters represented by codes `#09`, `#0A`, and `#0D`). We used some of these values to represent the reserved symbols. The

assignment of the values, as well as the meaning of the reserved symbols, is different in the simple and the adaptive modeling strategies. The exact details can be found in Sections 6.2 and 6.3, respectively.

To encode the identifiers, a mechanism that allows to represent *any* value is required. In our scheme, we decided to use order-2 Fibonacci codes (see Section 3.2.1) for encoding the identifiers.

6.2 Simple modeling

The policy of the simple modeling strategy is onefold and resembles the transformation that has been used in XMill. The input XML data is transformed into a sequence of characters and structural symbols (reserved symbols and identifiers), and then passed to the grammar-based coder. As opposed to XMill, there are no data containers. The identifiers are assigned in an adaptive way, therefore the dictionary of them does not have to be transmitted with the compressed message.

In the simple modeling, the start-element tags and attribute names are dictionary encoded using unique identifiers. The attribute names are stored in the same dictionary as the names of the start-element tags.

Symbol	Description	Parameter
0	End of data	-
1	New tag	String
	New attribute	String
2	Known tag	Index to the dictionary of elements
	Known attribute	Index to the dictionary of elements
3	End tag	-
4	Comment	String
5	Processing instruction	Sequence of strings
6	DTD	Strings and reserved symbols
7	XML declaration	Sequence of strings
8	CDATA section	String
11	Start of attributes	Reserved symbols
14	Entity declaration	Sequence of strings
15	Notation declaration	Sequence of strings
16	Default data	String

Table 6.1: Reserved symbols in the simple modeling

To encode the structure of the document, a set of reserved structural symbols is used. The complete list of reserved symbols is summarized in Table 6.1. Along with the numeric value of each symbol, also its purpose and type of prospective parameters are described.

The most significant symbols are those that are used for reporting new and known start-element tags (or attributes). If a new start-element tag (attribute) is encountered during the processing of the input document, the NEW TAG symbol is produced, followed by the name of the element (attribute). The name is inserted into the dictionary and assigned a unique identifier. Any subsequent occurrence of the tag (attribute) is encoded using the KNOWN TAG symbol, followed by its identifier. The identifiers are encoded using order-2 Fibonacci codes.

If attributes and start-element tags are encoded the same way, one possible problem is how to distinguish between them. In the simple modeling, we use a special ATTRIBUTE SECTION symbol that informs the decoder that a list of attributes and their values follows. The list is terminated with the END OF DATA symbol right after the value of the last attribute. To make the operation of the decoder simpler, the list of attributes of an element always precedes the element identification in the data stream. After buffering the names and the values of the attributes, the decoder simply recovers the name of the element from the subsequent data, and attaches the attributes to it.

The end of element is reported by special END TAG symbol. This symbol does not require any additional parameters. Since the elements are properly nested in well-formed XML documents, it is always clear which element is ending.

```
<lib>
  <book>
    <author>...</author>
    <title>...</title>
  </book>
  <book lang="...">
    <author>...</author>
    <author>...</author>
    <title>...</title>
  </book>
</lib>
```

Figure 6.1: Sample XML document

To illustrate the transformation of the XML document, consider the XML document in Figure 6.1. By using the simple modeling strategy, the document gets transformed into:[1]

N *lib* N *book* N *author* ... / N *title* ... / / A N *lang* ... E K I2 K I3 ... / K I3 ... / K I4 ... / / /

Following notation is used. The characters N and K represent the reserved symbols NEW TAG and KNOWN TAG, respectively. The character A stands for the ATTRIBUTE

[1]Newline characters and whitespace after or before the start-element tags are omitted in this example. In the real implementation, these characters are compressed, too.

SECTION symbol, and the character `E` for the END OF DATA symbol. The END TAG symbol is represented by the character `/`. Finally, the we use `I2`, `I3`, and `I4` to represent the identifiers assigned to the tags `book`, `author`, and `title`, respectively.

Remark. If we compared the representation of the structure obtained by the simple transformation to that obtained by XMill, we would see that it would be shorter in the case of XMill, because XMill does not need any tokens for reporting new and known tags. It encodes the tags using only their identifiers. This is possible because of two reasons. First, the character data and the structure are compressed separately (in different containers) in XMill, and therefore there is no need to distinguish between the character data and structure by special-purpose tokens. Second, XMill assigns the identifiers in a semi-adaptive fashion. The dictionary of the tags is a part of the compressed message, and the decoder does not have to learn the assignment of the identifiers.

6.3 Adaptive modeling

In adaptive modeling, we try to learn as much as possible about the structure of the input XML document during its processing. This knowledge can be used for the prediction of the subsequent data. In most occasions, the elements have fairly regular structure. Recalling the XML document in Figure 6.1, we can see that the element `book` contains one or two subelements `author` and one subelement `title` (in that order). Once we discover this structure, we are able to predict the "behavior" of the element `book` in the future. As a result, the amount of the data to be compressed can be substantially reduced.

Similarly to the simple modeling, structural symbols (i.e. reserved symbols and identifiers) are used to encode the XML structure. However, the meaning of the reserved symbols is context-dependent in the adaptive modeling. Thanks to that, many symbols can be assigned the same value. Therefore, the range of the reserved numeric values is smaller, and the probability of better compression higher, because of the increased homogenity of the transformed data.

The reserved symbols for the adaptive modeling are listed in Table 6.3.

We observed that the names of the attributes collided with the names of the elements only very rarely. Therefore, we decided to use two distinct dictionaries to represent the start-element tags and attribute names in the adaptive modeling—as opposed to one dictionary in the simple modeling. Since both of the dictionaries are usually shorter than one common dictionary, the range of the identifiers can be reduced.

In the simple modeling, the structural symbols are passed to the compressor whenever any structural information is supplied by the XML parser (end of an element, start of an element, character data, etc.). The adaptive modeling is more economic: it generates the structural symbols only if they are necessary. If the structure has been predicted, no symbols need to be generated. Suppose, for example, that we are processing element `author` from Figure 6.1. Suppose that this element has been seen in the past so that the adaptive modeling is aware of its structure. After the character data in the element

Symbol	Description	Parameter
0	End of data	-
	Nack	Distance from clue transition
	Transition	Identifier of the transition
1	New tag	String
	New attribute	String
	Comment	String
2	Know tag	Index to dictionary of elements
	Known attribute	Index to dictionaty of attributes
	Processing instruction	Sequence of strings
	XML declaration	Sequence of strings
3	End tag	-
	CDATA section	String
	DTD	Strings and reserved symbols
4	Characters	Strings and reserved symbols
5	Default data	String
6	Entity declaration	Sequence of strings
7	Notation declaration	Sequence of strings

Table 6.3: Reserved symbols in the adaptive modeling

`author`, the element ends. To report this, the END TAG symbol is generated in the simple modeling. In the adaptive modeling, no symbol needs to be generated, because `author` was expected to end at this point.

There are, however, many situations when the predictions may fail in the adaptive modeling. For example, if some element has very irregular structure, it is often very difficult to predict it. To handle these situations, a rather elaborate escape mechanism is required. This escape mechanism should make it possible both to inform the decoder that the prediction failed (and where it failed), and to instruct the decoder what actions should be performed.

In the following sections, the principles of the adaptive modeling are described in detail.

6.3.1 Model of element

In the adaptive modeling, each element e is assigned a finite state automaton M_e which describes its structure. We call the automaton a *model of element* e.

The states of the automaton can be of four types: besides the *initial state* and the *accepting states*, there are also so called *character states* and *element states*. In an analogy to the SAX parsing interface, the initial state can be seen as a parallel to the "start element" event, and the accepting states as a parallel to the "end element" event. The character states indicate the presence of the character data in the element, and the element states represent the nested elements. Each element state carries the name of the nested element

and acts as a reference to the model describing that element.

The transitions in the automaton characterize the arrangement of the nested elements and character data within the element. Each transition t is assigned a frequency count f_t that indicates how many times t has been used. From these counts, the probabilities of the transitions are calculated. For each state s, define $T(s)$ to be the set of the transitions that lead out of s. Then the probability of the transition $t \in T(s)$ is defined as

$$p_t = \frac{f(t)}{\sum_{t' \in T(s)} f(t')}$$

For each state s, the most probable transition that leads out of s shall be called *deterministic*; the remaining transitions shall be called *nondeterministic*. We shall say that the state s *predicts* the state s', if the deterministic transition leading out of s ends in s'.

Besides the frequency counts, the transitions are also assigned a numeric identifier which uniquely identifies them with respect to their start state. More formally, for each state s, the set $T(s)$ is finite and equal to $\{t_1, \ldots, t_{s_n}\}$. For each transition $t_i \in T(s)$ $(1 \leq i \leq t_{s_n})$, its identifier is defined as $id(t_i) = i$.

Each model represents exactly one element, but the models are "linked" together by means of the states that represent the nested elements.

The element models are built incrementally during the compression (or decompression) and reflect the present structure of the individual elements. The initial automaton for each element consists of one (initial) state with no transitions defined. During the processing of the data, new states and transitions are being added to the automaton, and frequency counts are adjusted.

Because of the way how the models are being built (see Section 6.3.2.1), they are always acyclic, i.e. they do not contain a sequence of transitions that forms a cycle. In fact, the models have a structure of a tree, with leaves representing the accepting states.

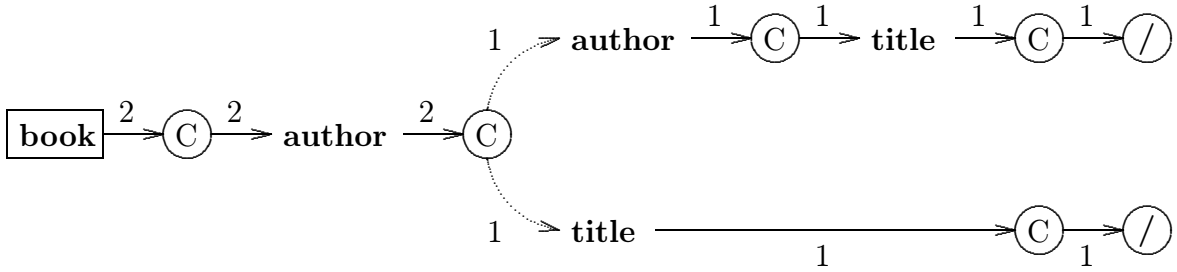


Figure 6.2: Model of element `book`

Example 6.1. To illustrate how the element models look like, we recall the XML document in Figure 6.1 (page 60). The document contains four types of elements (`lib`, `book`, `author`, and `title`), therefore four element models will be created during its processing. In Figure 6.2, the final model of the element `book` is displayed. In the model, the initial state is

depicted as **book**, the accepting states as (/) , and the character states as (C) .[2] The element states are represented simply by the names of the elements. The nondeterministic transitions are dotted in the figure.

The element `book` appeared two times in the document. In its first occurrence, it contained one `author` subelement, and one `title` subelement (in that order). In its second occurrence, it contained two `author` subelements, followed by one `title` subelement. If we look at the model of `book`, we will see that the model permits both possibilities. In the second character state, transitions leading both to the `author` and the `title` element states are defined. □

Remark. There is an obvious similarity between the element models and the element declarations in the DTD. Indeed, the information that we try to describe by the construction of the models is already present in the DTD. So why not to use the element declarations and construct the models based upon them? There are several possible answers to this. First, we designed our compression scheme to be independent on the DTD, therefore we cannot use it on principle. Second, the DTD often describes a whole class of documents, and this class may be unnecessarily broad. For our purposes, we want to create a model of just one distinct document. We believe that such a model is more suited for the compression, because it is more specific and describes the data more accurately. Third, the whitespace characters are ignored in the DTD. Our models have to work with this data to keep the original indentation of the source.

6.3.2 Modeling algorithms

During the compression and the decompression, we make use of the element models to make predictions of the structure of the elements. For each processed element, the prediction is achieved by the movement along the deterministic transitions in the corresponding model. Suppose that in the current state of the model, a certain element is expected to occur. In other words, the current deterministic transition ends in an element state that represents the expected element. If the expected element really occurs, the only thing we have to do is to move along the deterministic transition, and to enter the referenced model. The point is that *no information* needs to be sent to the decompressor.

However, the situation is not always that simple. The most important problem is that we do not have any element models in the beginning. The models are being created during the processing of the data. Each time a new element is encountered, a simple initial model is assigned to it. This model gives no predictions at all; it consists of only one state and no transitions. During the processing of the element, new states and transitions are added to

[2]The element `book` does not contain any character data in the common sense, only newline and whitespace characters before or after the nested elements. From our point of view, however, these characters are considerd to be character data, too. If we ignored them, the compressed data would not represent the original document, since all the indentation would be lost.

the model to reflect the structure of the element. When the element is encountered again in the document, its structure is compared to that predicted by its model. If the model does not describe the present structure accurately, it is updated in an appropriate way. In other words, the models are adaptive, and are constantly being refined to reflect the structure of the corresponding elements. Therefore, the predictions and the updating of the models have to be performed simultaneously.

Another problem is that the models may not always give good predictions. In case that the element has an irregular and varying structure, which is difficult to predict, the model may be mistaken. It may easily happen that the character data is predicted by the model, for example, but a nested element actually occurred. Therefore, an escape mechanism is required.

Because the modeling is adaptive, the decompressor must be able to maintain the same models as the compressor. Moreover, it must be able to make the same predictions, and to recover in case that an escape event occurs. However, to keep the decompressor synchronized with the compressor is not as simple as it may seem. As noted before, the prediction of the structure is realized by the movement along the deterministic transitions in the element models. Thanks to these predictions, the compressor can operate without sending any data to the decoder very often. While this may improve the compression performance substantially, it makes the cooperation with the decompressor rather complicated. For example, if an escape event occurs during the compression, the decompresor is often unable to locate where (i.e. in which model, and in which state) it actually occurred.

Fortunately, it shows that during the compression, there are states in the models that can be uniquely identified by both the compressor and the decompressor—for example, the initial state of the root element. We refer to these states as to *clue states*. Thanks to these states, the decompressor can be always synchronized with the compressor. If the compressor emits an escape event, it also sends the distance (that is, the number of deterministic transitions) from the clue state. The decoder then moves along the specified number of deterministic transitions, and performs an appropriate recover action. After that, both the compressor and the decompressor set the current state to be the next clue state.

In the following text, we present the algorithms that characterize the basic operation of the compressor and the decompressor, respectively.

6.3.2.1 Compression

In this section, we present an algorithm for modeling and encoding the structure of one element, as it is implemented in the compressor. The algorithm operates recursively, making it possible to model and encode also the nested elements in a natural fashion. It constructs the model for the element, and transforms its structure depending on the predictions provided by the model.

Well-formed XML documents contain one root element, and the other elements are nested. Therefore, the structure of the whole document can be modeled and encoded by applying the algorithm to the root element.

The algorithm takes the name of an element as the parameter. Besides that, it uses an external variable *dist* for measuring and reporting the distance from the clue states. Initially, the compressor sets this variable to 0. If it necessary to send the value of *dist* to the decompressor, it is encoded using order-2 Fibonacci code.[3]

The structure of the root element e and its nested elements is modeled and encoded as follows:

1. If M_e does not exist (i.e. the element e was encountered for the first time), create an initial model for e.

2. Set s to be the initial state of M_e.

3. Read the data at the current position in the element e. Depending on the type of the data (character data, name of a nested element, or end of element), and on the properties of s, perform one of the four possible actions:

   - There is no transition leading out of s. Then create a new state s' which has the same type as the current data (that is, a character state for character data, an element state for the nested element, and an accepting state for the end of element), and a new transition $s \to s'$. Set $dist = 0$ and encode the data using the structural symbols:
     - Character data: CHARACTERS `data`
     - A name of a new (previously unseen) element: NEW TAG `name`
     - A name of a known (previously seen) element: KNOWN TAG `identifier`
     - End of element: END TAG
   - There are some transitions leading out of s, but none of them ends in a state with the same type as the data. Then output the NACK reserved symbol followed by the value of the distance counter *dist*. After that, create a new state s' which has the same type as the data, and a new transition $s \to s'$. Set $dist = 0$ and encode the data using the structural symbols.
   - The deterministic transition leading out of s ends in a state s' which has the same type as the data. Increase the frequency count of the transition, as well as the the distance counter *dist* by 1. In case of character data, encode it as follows: CHARACTERS `data`
   - Nondeterministic transition t leading out of s ends in a state s' with the same type as the data. Then output the NACK reserved symbol followed by the value of the distance counter *dist*, and the TRANSITION reserved symbol followed by $id(t)$. Increase the frequency count of the transition t and set $dist = 0$. In case of character data, encode it as follows: CHARACTERS `data`

[3]Because it is only possible to encode integers greater than 0 using Fibonacci codes, $dist + 1$ is actually encoded. After the decompressor decodes the value, 1 is subtracted.

4. The new current state is $s = s'$. Depending on the type of s, perform one of the two possible actions:
   - The state s represents an element e'. Then apply the complete algorithm recursively to e'.
   - The state s is an accepting state. Then the algorithm ends.
5. Continue with step 3.

Remark. Besides start tags, end tags, and character data, XML documents may contain additional XML mark-up, such as comments, processing instructions, or CDATA sections. The above described algorithm treats this mark-up as a part of the character data. Each time a comment, processing instruction, or a CDATA section is encountered, it is encoded using appropriate structural symbols (COMMENT, PI, CDATA), and inserted in a character data section. For example, consider the following XML fragment:

```
<e>hello <!-- world --></e>
```

The content of element `e` is encoded as one character data block:

CHARACTERS `hello` COMMENT `world`

6.3.2.2 Decompression

In the decompressor, the structure and the content of the compressed document is reconstructed. Compared to the algorithm of the compressor, it operates in a much simpler manner.

The algorithm can be stated in a form of a simple loop. In each step of the algorithm, a block of the encoded data is read and analyzed. The block consists of a reserved structural symbol, optionally followed by its parameter. Based upon the information contained in the block, the models of the elements are constructed and updated.

The encoded data starts with the NEW TAG reserved symbol, followed by the name of the root element. The decompressor first creates an initial model for the root element, and enters the initial state of the model. After that, the decompressor reconstructs the content of the root element and the nested elements according to the following steps;

1. Read a block of data. Depending on the structure of the block, perform one of the following actions:
   - NEW TAG `name`: Insert `name` into the dictionary of elements and move along the deterministic edges (adjusting their frequency counts and possibly visiting the nested element models) until a state s with no outgoing transitions is reached. Create an initial model for the element. After that, create a new element state s' to represent the element, and a new transition $s \rightarrow s'$.

- KNOWN TAG `identifier`: Resolve the name of the element represented by `identifier` and move along the deterministic edges (adjusting their frequency counts and possibly visiting the nested element models) until a state s with no outgoing transitions is reached. Create a new element state s' to represent the element, and a new transition $s \to s'$.
- CHARACTERS `data`: Read `data` and move along the deterministic edges (adjusting their frequency counts and possibly visiting the nested element models) until a character state or a state with no outgoing transitions is reached. Denote this state as s. Create a new character state s', and a new transition $s \to s'$.
- END TAG: Move along the deterministic edges (adjusting their frequency counts and possibly visiting the nested element models) until a state s with no outgoing transitions is reached. Create a new accepting state s', and a new transition $s \to s'$.
- NACK `distance`. Move along the specified number of deterministic edges (adjusting their frequency counts and possibly visiting the nested element models), ending in a state s. Depending on the subsequent data, perform one of the four possible actions:
  - NEW TAG `name`: Insert `name` to the dictionary of elements, and create an initial model for the element. After that, create a new element state s' to represent the element, and a new transition $s \to s'$.
  - KNOWN TAG `identifier`: Resolve the name of the element given by `identifier` and create a new element state s' to represent the element, and a new transition $s \to s'$.
  - END TAG: Create a new accepting state s', and a new transition $s \to s'$.
  - CHARACTERS `data`: Read `data` and create a new character state s', and a new transition $s \to s'$.
  - TRANSITION `identifier`: Move along the specified transition to a state s' and increase the frequency count of the transition by 1.

2. Depending on the type of s', perform one of the two possible actions:
   - The state s' represents an element e. Then enter the initial state of M_e.
   - The state s' is an accepting state of the model. If the model was nested in another model, return to the corresponding state of the parent model. If the model was not nested (it was a "root model"), the algorithm ends.
3. Continue with step 1.

Remark. In the above described algorithm, only the models of the elements are constructed. In a real implementation, however, the algorithm also emits SAX events depending on its current position within the models. For example, each time an initial state of

some model is encountered, the "start element" SAX event is emitted. Similarly, each time an acepting state is visited, the "end of element" SAX event is emitted. In the case of character data, it is parsed first and possible comments, processing instructions, CDATA sections and the remaining character data are extracted from it. After that, corresponding SAX events are emitted.

6.3.2.3 Attributes

If we look closely at the algorithms presented in Sections 6.3.2.1 and 6.3.2.2, we will see that they do not count with attributes. The problem with attributes is that they are unordered, as opposed to elements which form an ordered sequence. In other words, `<e a="..." b="...">` is equivalent to `<e b="..." a="...">`. Because of that, it is not possible to represent the set of the attributes in the element models.

As a solution to this, we use so called *enhanced element models* in the implementation. The enhanced models are the same as the models discussed so far, except that they contain one additional, special purpose, state. We refer to this state as to the *attribute state*. There is exactly one attribute state in the model, and it follows immediately after the initial state. The initial model for each element contains one initial state connected with one attribute state.

The attribute state is assigned two counters to store the number of times the element has been seen with attributes, and without them, respectively. Depending on which of the counters is larger, the attributes are either expected to occur in the element, or not. Initially, the elements are expected not to have any attributes.

Recall the model for element `book` in Figure 6.2 (page 63). A fragment of the enhanced model for the element is shown in Figure 6.3.

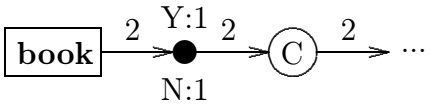


Figure 6.3: Enhanced model of element `book`

The attributes are encoded as a sequence of pairs: (attribute identification, value). If the attribute occurs for the first time, it is encoded as follows:

NEW TAG `name value`

and `name` is inserted into the dictionary of attributes. If the attribute has been seen before, it is encoded as follows:

KNOWN TAG `identifier value`

where `identifier` is an index to the dictionaty of attributes. The list of attributes is terminated by the END OF DATA reserved symbol.

In order to make it possible to work with the enhanced models, the algorithms described in Sections 6.3.2.1 and 6.3.2.2 have to be slightly altered.

Each time an element is encountered, the compressor enters the initial state of the corresponding model, moves to the attribute state, and increases the value of the distance counter by 1. Based upon the values of the counters in the attribute state, and upon the properties of the element, one of the four possible actions may performed:

- The element does not contain attributes, and this is what the counters in the attribute state predict. Increase the appropriate counter by 1. Nothing else needs to be done.
- The element does not contain attributes, and the counters in the attribute state predict the opposite. Then send the NACK reserved symbol, followed by the value of the distance counter to the decompressor. Increase the appropriate counter by 1 and set the distance counter to 0.
- The element contains attributes, and this is what the counters in the attribute state predict. Increase the appropriate counter by 1, set the distance counter to 0, and encode the block of attributes.
- The element contains attributes, and the counters in the attribute state predict the opposite. Then send the NACK reserved symbol, followed by the value of the distance counter to the decompressor. Increase the appropriate counter by 1, set the distance counter to 0, and encode the block of attributes.

The algorithm in the decompressor is modified as follows. Each time a data block starting with NEW TAG or KNOWN TAG is read, the subsequent movement along the deterministic edges can stop also in an attribute state that predicts the attributes. If this is the case, the data block represents the first of the attributes, and the remaining attributes are read. If the decompressor reads the NACK symbol, and ends in an attribute state after the movemenent along the specified number of transitions, the prediction of the attribute state is interpreted inversely: that is, if the state does not predict the attributes, they are expected in the subsequent data, and vice versa. Each time the decompressor visits an attribute state, its counters are incremented in an appropriate way.

6.3.3 Structural entropy

In this section, we show how the element models can be used for measuring the complexity and regularity of the XML documents.

Let M_e be the model of the element e. Define $C(M_e)$ to be the set of all accepting computations of M_e. Since the automaton M_e is acyclic, $C(M_e)$ is finite, and for each computation $c \in C(M_e)$, there exist finitely many transitions $t_1, t_2, \ldots, t_{n_c} \in M_e$ such that $c = t_1 t_2 \cdots t_{n_c}$. The probability of the accepting computation c is equal to

$$p(c) = \prod_{i=1}^{n_c} p_{t_i}$$

where p_{t_i} denotes the probability of the transition t_i in the calculation (for the definition of this probability, see Section 6.3.1).

If we know the probabilities of the accepting computations of M_e, we are able to evaluate the *structural entropy of element* e, which we define as follows:

$$SH_e = -\sum_{c \in C(M_e)} p(c) log_2 p(c)$$

Structural entropy characterizes the complexity the element. The more regular the element is, the smaller its structural entropy is.

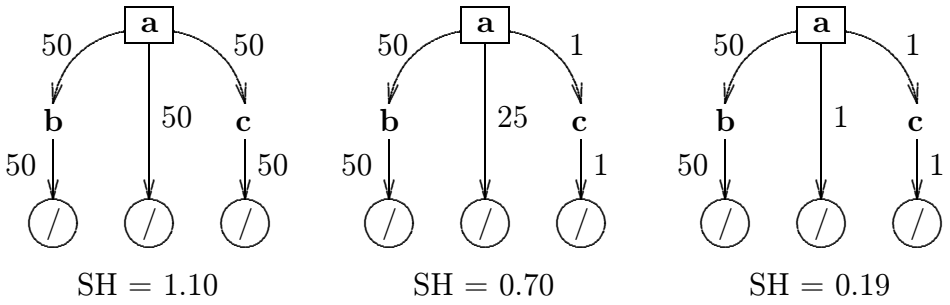


Figure 6.4: Models with different structural entropies

Example 6.2. To illustrate the meaning of the structural entropy, consider the following situation. Suppose we want to examine the complexity of the element `a`, which may be either empty or contains one nested element (`b` or `c`). Depending on the "behavior" of the element `a`, its complexity may vary. Three of the possible scenarios are shown in Figure 6.4.

In the first scenario, the probabilities that `a` contains elements `b` or `c`, or none of them, are equal. This situation is the worst for the adaptive modeling, since element `a` is very irregular and it is almost impossible to predict its structure based upon its model. As the result, the escape mechanism is used very frequently, causing more data to be compressed.

In the second scenario, `a` contains element `b` in most occasions. It is also empty quite often, while element `c` is rather improbable to occur. In this case, the structure of `a` is much more predictable. Still, the escape mechanism has to be used frequently.

In the third—and the most ideal—scenario, `a` contains element `b` in most occasions, and only very rarely it is empty or contains element `c`. In this case, the model gives the best prediction, because it is almost sure that `a` contains `b`. The escape mechanism is used only scarcely. □

Based upon the structural entropies of the individual elements, the complexity of the entire XML document can be estimated. Suppose that there are n different elements in the document d, and each element e_i has occurred l_i $(1 \leq i \leq n)$ times. The *structural entropy of document d* is defined as

$$SH(d) = \frac{\sum_{i=1}^{n} l_i * SH_{e_i}}{\sum_{i=1}^{n} l_i}$$

In other words, the structural entropy of the document is equal to the weighted average of the structural entropies of the elements.

6.3.4 Practical example

In this section, we demonstrate the adaptive modeling on a short example. Consider the XML document in Figure 6.5.[4]

```
<x><y a="..."><z>...</z></y><y a="..."><z>...</z></y></x>
```

Figure 6.5: Sample XML document

By using the adaptive modeling strategy, the compressor transforms the structure of the document as follows:

```
N x N y ! 1 N a ... E N z C ... / / K I2 K A1 ... E C ... /
```

Following notation is used. The characters `N` and `K` represent the reserved symbols NEW TAG and KNOWN TAG, respectively. The character `!` stands for the NACK symbol, the character `E` for the END OF DATA symbol, and the character `C` for the CHARACTERS symbol. The END TAG symbol is represented by the character `/`. Finally, `I2` and `A1` represent the identifiers assigned to the element `y`, and to the attribute `a`, respectively.

In Figure 6.6, the resulting element models are depicted. Because the models are very simple, it is easy to see that their structural entropies are equal to 0. In other words, the elements have a very regular structure.

To compare the performance of the adaptive modeling to that of the simple modeling, we transformed the sample document using the simple modeling:

```
N x N y A N a ... E N z ... / / K I2 A K I3 ... E K I4 ... ///
```

Here, the character `A` represents the ATTRIBUTES symbol, and the identifiers `I2`, `I3`, `I4` represent the repeated occurences of the tags `y`, `a` and `z`, respectively.

It can be seen that the adaptive modeling yields a slightly shorter representation. It uses 14 structural symbols and 2 identifiers, whereas in the case of the simple modeling, 16

[4]The whitespace characters and the indentation have been removed from the source to make it shorter.

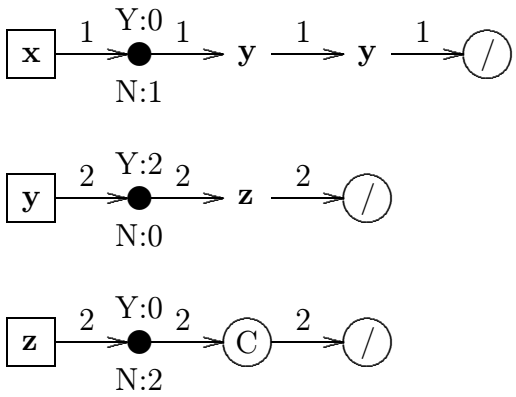


Figure 6.6: Models after processing the sample document

structural symbols and 3 identifiers have been used. In the adaptive modeling, one escape event was emitted, because the element y contained attributes on its first occurrence (and by default, the elements are supposed not to have any attributes when they occur for the first time).

In this particular case, the difference between the simple and the adaptive modeling is not very significant. However, it follows from our experiments (see Section 9.2.1) that documents with regular structure can be reduced up to 40% better using the adaptive modeling. On the other hand, documents with complex and irregular structure cause the escape mechanism to be applied very often, and the performance of the adaptive modeling may deteriorate.

Chapter 7

The Architecture of Exalt

Exalt is written in C++, with object-oriented design in mind. We decided to implement it as a library that would be easy to use and extend. The functionality of both the compressor and the decompressor is present in the library.

The system is component-based. For each component, a unified interface has been defined to make it easy to modify its functionality, or to replace it.

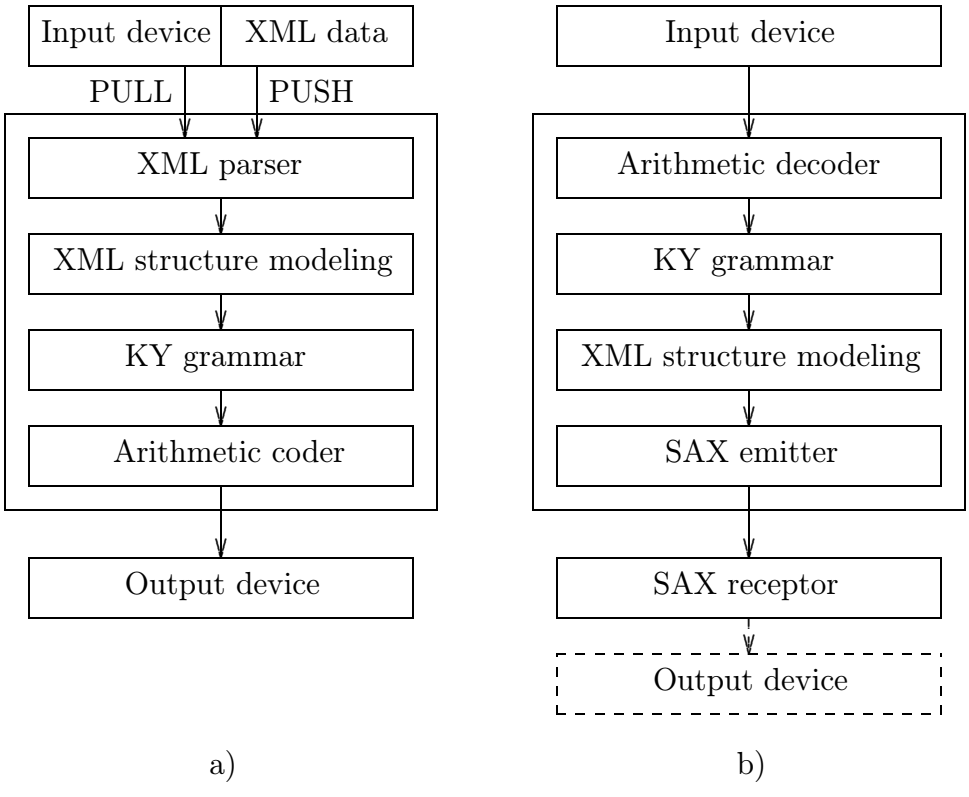


Figure 7.1: Architecture of Exalt: the compressor (a) and the decompressor (b)

The architecture of both the compressor and the decompressor is sketched in Figure 7.1. Besides the components mentioned in the figure, there is a number of less important helper components in our system (for example, a component for conversions between text encodings). For more detailed information about these components, please refer to the developer documentation (Appendix B) or to the generated API documentation on the supplied CD.

During the compression, a stream of data is produced. As can be seen in Figure 7.2, the structure of the stream is very simple. It starts with a short header block which contains the identification of the Exalt file format and information about the compressed data (most importantly, the modeling strategy used for the compression). After the header block, the compressed data follows.

Figure 7.2: Structure of the compressed file

In the following sections, the individual components of our system are discussed.

7.1 XML parser

The XML parser module, as the name suggests, deals with the parsing of the input XML documents. We have not written the complete parser by ourselves; instead, we have used the Expat XML parser [8], an open-source SAX parser whose native language is C, and have written C++ bindings to it. We decided to use Expat because of its speed, but it is possible to use virtually any SAX parser.

The parser processes the input XML data and emits corresponding SAX events. These events are passed to the XML structure modeling module (see section 7.2) for further analysis.

The parser can work in two main modes: in the PULL mode, and in the PUSH mode.

In the PULL mode, the parser is given access to some input device (for more information about inputs and outputs in Exalt, refer to Section 7.6). It then reads the XML document from the device and processes it. There is no way how to stop the parser unless complete document is processed or a parse error occurs.

In the PUSH mode, the parser doesn't read the data by itself. Instead, it waits until the data is supplied by other component of the system. The data is supplied to the parser block by block, and the parsing stops when the final block is indicated or a parse error occurs. The PUSH interface is useful for processing the data that is dynamically generated, for example.

7.2 XML structure modeling

This module acts as the "intelligence" of our compression scheme. It deals with the modeling of the structure of the XML document. The goal of this modeling is to reduce the amount of the data that has to be compressed. Since XML documents usually have a rather regular structure, which contains a lot of redundant information, it is possible to create a model of it which may be utilized in the process of compression. The modeling is based on the analysis of SAX events that are supplied by the XML parser.

We have implemented two modeling strategies: the simple modeling strategy and the adaptive strategy (they are described in Sections 6.2 and 6.3, respectively). While the principles of the individual modeling strategies are significantly different, the interface of the module remains uniform.

During the compression, the structure modeling module receives the SAX events from the XML parser, and after processing them, it sends the data that has to be compressed to the underlying KY grammar module (see Section 7.3). In this module, the grammar is updated to reflect the incoming data, and possible codebits are output by the arithmetic coder (see Section 7.4).

During the decompression, the structure modeling module receives data from the KY grammar module and recovers corresponding SAX events from it. These SAX events are emitted by means of the SAX emitter module, and handled either by the default SAX receptor (for information about the SAX emitter and SAX receptor modules, refer to Section 7.5), or by a user-supplied SAX receptor. The default SAX receptor simply reconstructs the original XML document from the incoming SAX events and outputs it, while the user-supplied SAX receptor may do anything the user needs.

7.3 KY grammar

Together with the arithmetic coding module (see Section 7.4), this module implements the grammar-based compression routines, as described in the Section 3.6.1.

The module implements the greedy irreducible grammar transform and the set of reduction rules that are used to keep the grammar irreducible after it is updated. A sequential algorithm is used to encode the grammar.

During the compression, the symbols supplied by the structure modeling module are appended to the grammar using the greedy irreducible grammar transform. If it is necessary, the reduction rules are applied to make the grammar irreducible. Depending on the state of the updated grammar, either a terminal symbol or a variable is encoded using the sequential algorithm and the arithmetic code, and the next symbols are processed. The sequential algorithm encodes the grammar during its construction, making on-line compression possible.

During the decompression, the symbols (either terminal symbols or variables) decoded by the arithmetic decoder are appended to the grammar, and the reduction rules are applied if the appended grammar is not irreducible. The data sequence represented by the

appended symbol (one character in case of a terminal symbol, and a string of two or more characters in case of a variable) is sent to the structure modeling module. This module buffers the incoming data and once sufficient information is gathered, the model is updated and an SAX event is possibly emitted by means of the SAX emitter module.

7.4 Arithmetic coding

In the grammar-based compression, an arithmetic code with a dynamic alphabet is used to encode the structure of the grammar. It was clear to us that to create a usable arithmetic coding routines would be a difficult task in itself, therefore we decided to use a third-party sources for arithmetic coding, as in the case of the XML parser module.

The implementation of arithmetic coding that we have chosen originates from Moffat and Witten [22]. It is written in C, and is available for academic purposes. It brings substantial improvements to the "standard" arithmetic coding routines that have been presented by Witten in 1987. Its most important features are the support for large alphabets, the use of fewer multiplicative operations, and the overall speed. Furthermore, it completely divorces the coder from the modeling and probability estimation.

In order to make the use of the arithmetic coder simple, we have devised a unified C++ interface to the arithmetic coding module, and have rewritten the Moffat and Witten's routines to C++ such that they conformed to this interface. Using the C++ inheritance mechanism, it is easy to switch to different implementations of arithmetic coding, if necessary.[1]

When using the sequential algorithm to encode the grammar, arithmetic coding is used to encode the symbols (terminal symbols and variables) that are appended to the grammar. Because the number of variables varies (usually it grows) during the construction of the grammar, a dynamic alphabet has to be used. Sequential algorithm deals with the growing alphabet this way: each time a new variable is introduced by the grammar transform, the KY grammar module instructs the arithmetic coding module to install this symbol into the symbol table. Thanks to that, it is always guaranteed that the arithmetic coder knows the encoded symbols, and no escaping is necessary.

During the decoding, arithmetic decoder recovers the symbols and gives them forth to the KY grammar module. In the sequential algorithm, the grammar is repeatedly adjusted by appending these symbols to it, with the reduction rules applied if they are necessary. If a new variable is introduced after the update of the grammar, the arithmetic coding module is instructed to install it to its symbol table. Thus, any eventual occurrence of the new variable in the subsequent data will not confuse the arithmetic decoder.

[1] Actually, the same holds to more or less any component of the system.

7.5 SAX emitter and SAX receptor

The original XML document is compressed on the basis of SAX events that are supplied by the XML parser. During the decompression, we find ourselves in a reverse situation: the decoder incrementally processes the compressed data and emits SAX events that it has recovered from it. Based upon this events, the original document is reconstructed.

To make the emitting and handling of SAX events simple, Exalt contains two specialized components: SAX emitter and SAX receptor. The only function of the SAX emitter is to deliver the SAX events to the SAX receptor. Whenever the decompressors recovers complete information about a certain SAX event, it passes it to the SAX emitter, which forwards it to the SAX receptor for the processing. It up to the SAX receptor, how the event is handled.

By default, Exalt uses a SAX receptor that simply reconstructs the original data from the SAX events, and outputs it. However, it is possible to register a user-defined SAX receptor and handle the events in a different way. Thanks to that, a transparent SAX parsing of the compressed data is possible.

7.6 Inputs and outputs

We designed Exalt to support various types of inputs and outputs. In order to make this possible, Exalt works with a higher level abstraction to input and output, which we call IO devices. At present, only files are supported, but it is easy to use any other "device", such as the network or some database, for example. We have defined a unified interface for the work with the IO devices, and by using the inheritance mechanism, it is easy to create new custom devices for a C++ programmer.

The life cycle of each IO device consists of three main phases: At first, the device must be *prepared*, then it is being *used* for some time, and at the end, it must be *finished*. To illustrate the work with the IO devices, consider a device that represents a file. Suppose that we want to write some data to that file. In the preparation phase, the file is opened for writing. After that, it is possible to write the data to the device. At the end, we close the file by finishing the device.

When the application supplies a device to the (de)compressor, it is supposed that it is already prepared. After the (de)compressor processes the data, it does not finish the device—it is up to the application to finish the work with it.

Chapter 8

Implementation issues

8.1 Grammar-related operations

In a grammar-based system, the most time consuming operations are those related to the building of the grammar and its updating. In this chapter, we describe our implementation of these operations.

We represent the grammar by a linked list of production rules. Similarly, each production rule is represented by a doubly linked list of terminal symbols and variables. The terminal symbols contain numeric values (representing character codes in most occasions), whereas the variables contain pointers to associated production rules. In each production rule, pointers to its first and last symbol are stored.

In our compression scheme, we have implemented the greedy irreducible grammar transform for construction of the grammar, and the sequential algorithm for its encoding. For more detailed information, refer to Section 3.6.1 (page 31).

8.1.1 Reduction Rules

In the greedy irreducible grammar transform, the grammar is constructed by appending symbols representing the longest possible prefix of the input data to the root production rule. In each step, exactly one symbol is appended. This symbol is either a variable representing a production rule of the grammar, or a terminal symbol representing the first character of the input data. After the symbol is appended, three Reduction Rules[1] may be performed to keep the grammar as compact as possible.

Reduction Rule 1 ensures that the production rules are used at least two times in the range of the grammar. If some of the production rules is used only once, it is removed from the grammar, and its content is inserted into the production rule that referenced it.

Reduction Rule 2 ensures that no digram (i.e. a pair of two adjacent symbols) appears in two non-overlapping positions in the range of the root production rule. If some digram

[1] In Section 3.6.1, two more Reduction Rules are defined. However, these are not required because of the nature of the greedy irreducible grammar transform.

appears two times in the root production rule, a new production rule representing this digram is created, and each occurrence of the digram in the root production rule is replaced by the variable referencing the new rule.

Reduction Rule 3 is similar to Reduction Rule 2, except that it ensures that no digram appears in two distinct production rules. If some digram appears in two distinct production rules, a new production rule representing this digram is created, and each occurrence of the digram in both production rules is replaced by the variable referencing the new rule.

Let α be the last symbol of the root production rule before β is appended to it. It follows from Theorem 3 (see page 37) that three possible scenarios may happen:

- The digram $\alpha\beta$ does not appear in two non-overlapping positions in the range of the grammar. Then no Reduction Rules can be applied.
- The digram $\alpha\beta$ appears in two non-overlapping positions in the range of the grammar, and new rule has been introduced in the previous update operation. Then apply Reduction Rule 2 if $\alpha\beta$ repeats itself in the root production rule, and Reduction Rule 3 otherwise.
- The digram $\alpha\beta$ appears in two non-overlapping positions in the range of the grammar, and no new rule has been introduced in the previous update operation. Then apply Reduction Rule 2 followed by Reduction Rule 1 if $\alpha\beta$ repeats itself in the root production rule, and Reduction Rule 3 followed by Reduction Rule 1 otherwise.

To make the detection of repeated digrams efficient, we store the digrams in a hash table. Specifically, for each symbol in the grammar (terminal symbol or variable), we keep track of the symbols that adjoin it somewhere in the range of the grammar. In the hash table, each symbol is assigned a list of adjoining symbols and pointers indicating their location. In our implementation, we use two hash tables, one for storing the adjoining symbols of terminal symbols and one for storing the adjoining symbols of variables.

During the processing of the input data, the set of digrams is updated by adding new digrams, or by removing the existing ones. If no Reduction Rule is applicable, one lookup in the hash table is necessary. Reduction Rule 1 requires up to three lookups in the hash table, and Reduction Rules 1 and 2 require up to four lookups. In the worst case, when Reduction Rule 2 or 3 is followed by Reduction Rule 1, seven lookup operations may be performed.

Example 8.1. Suppose that the string **abracadabra** has been processed using the greedy irreducible grammar transform. The resulting grammar representing this string looks as follows:

$$\begin{aligned} s_0 &\rightarrow s_1\mathbf{cad}s_1 \\ s_1 &\rightarrow \mathbf{abra} \end{aligned}$$

In Table 8.1, the representation of the digrams in the hash tables after processing the string is displayed. □

Terminal digrams		Variable digrams	
Symbol	Adjoining symbols	Symbol	Adjoining symbols
a	**d**, **b**	s_1	**c**
b	**r**		
c	**a**		
d	s_1		
r	**a**		

Table 8.1: Digrams after processing the string **abracadabra**

8.1.2 Searching for the longest prefix

In each step of the greedy irreducible grammar transform, a symbol representing the longest possible prefix of the input data is appended to the root production rule of the grammar. If there are production rules in the grammar that represent a prefix of the input data, the one that represents the longest string is appended to the root production rule. Otherwise, the first character of the input data is appended.

It follows from the above that in each step, we have to test the production rules of the grammar whether they represent a prefix of the input data. This may be a serious problem for large grammars consisting of thousands of production rules. Fortunately, it shows that there are often production rules in the grammar that represent prefixes of many (ten, twenty, ...) other production rules. In our implementation, we make use of this fact. For example, suppose that the grammar contains the following two production rules:

$$\begin{aligned} s_i &\rightarrow s_j\mathbf{11}\ldots \\ s_j &\rightarrow \mathbf{00} \end{aligned}$$

Clearly, s_j represents a prefix of s_i. Suppose that s_j was tested before s_i, and did not represent a prefix of the input data. Since s_j is a prefix of s_i, it is obvious that s_i cannot match the input data, too. Therefore, it would make no sense to test s_i. In the case that s_i was long, we might have saved substantial time by ignoring it. In our implementation, each rule is assigned a list of production rules that it is a prefix of. If the production rule does not represent a prefix of the input data, the production rules in the corresponding list are ignored in the current round of tests.

Similar situation may happen if the production rule s_i were tested before s_j. Since s_j represents a prefix of s_i, it is matched against the input data first. This is the same as if s_j were tested alone, not as a prefix of another rule. Thus, we can remember the results for s_j and omit s_j in the subsequent tests.

The data is supplied in blocks to the compressor. Therefore, it may happen that there are production rules in the grammar that represent strings longer than the data currently available in the buffer. If these production rules match the content of the buffer, we have to wait for the next data to see whether they match completely. To make this

possible, a position within these production rules after processing the data buffer has to be remembered.

After the tests, the longest completely matching production rule is appended to the root production rule, if such production rule exists; otherwise, the first symbol of the input data is appended.

8.1.3 Sequential algorithm

In our grammar-based compression scheme, we have implemented the sequential algorithm (see page 41) for the encoding of the grammar. In the sequential algorithm, the grammar is encoded incrementally using the arithmetic code with dynamic alphabet. Initially, the alphabet contains all possible values of terminal symbols (in our case, it contains characters with codes from 0 to 255).

The sequential algorithm works as follows. After a symbol (terminal symbol or variable) is appended to the root production rule of the grammar, it is arithmetically encoded and its frequency count is updated. If a new production rule has been introduced after appending the symbol and applying Reduction Rules 1-5, arithmetic coder is instructed to install the corresponding variable into its symbol table. Therefore, the alphabet used by the arithmetic code grows as new production rules are introduced.

The decompressor operates in a reverse fashion: it decodes the symbol, adjusts its frequency counts, appends it to the root production rule of the grammar, and applies Reduction Rules 1-5. In the case that a new production rule has been introduced, the corresponding variable is added into the symbol table used by the arithmetic decoder.

In a real implementation, it is not always necessary to install new symbols into the symbol table used by the arithmetic (de)coder whenever a new production rule is introduced. This is possible due to the following reason. Each time Reduction Rule 1 is applied, some of the production rules is removed from the grammar—and the symbol in the symbol table that corresponds to this production rule becomes useless. If we assign these "orphan" symbols to the newly created rules, there is no need to install new identifiers into the symbol table.

Some policy for the assignments of the orphan symbols has to be adopted to make it possible for the decompressor to uniquely identify the production rules. In our system, we use a queue of orphan symbols in both the compressor and the decompressor. Each time a new orphan symbol is introduced, it is enqueued to the queue. After a new rule is created, the queue of the orphan symbol is checked first. If the queue is not empty, a symbol is dequeued and assigned to the production rule. Only if the queue is empty, new symbol has to be created and installed into the symbol table used by the arithmetic code.

8.2 Modeling of XML structure

The modeling of XML structure represents an important component of our compressor. We have implemented two modeling strategies: the simple modeling and the adaptive

modeling (see Sections 6.2 and 6.3). In the subsequent text, we unveil some details on the implementation.

In both the simple and the adaptive modeling, we use hash tables for storing the information about the elements and the attributes. In the simple modeling, one hash table is used for both the elements and the attributes, whereas in the adaptive modeling, two separate hash tables are used. For each element (or attribute), its unique numeric identifier is stored in the hash table, and can be accessed by the name of the element (or attribute). In the adaptive modeling, each record in the element hash table contains also a pointer to a corresponding model (see Section 8.2.1 below).

During the compression, the modeling is based on the SAX events that are supplied by the XML parser. Depending on the type of the events and on the type of the modleing method, various actions may be performed. In most occasions, the content of the event in transformed into a mixture of structural symbols and data in some way, and output using a specified output device.

During the decompression, the situation is a little bit more complicated, since the underlying grammar-based decompressor supplies the data by short blocks. The modeling component therefore has to buffer the incoming data (we use a queue), and to process it only when a sufficient information is gathered.

While the simple modeling operates in a rather simple manner (it only encodes the incoming SAX events using a predefined set of structural symbols, and encodes the repeated occurrences of elements and attributes by indexes to the hash table), the adaptive modeling is much more complicated.

8.2.1 Adaptive modeling

During the adaptive modeling, element models are being built to describe the structure of the elements. Based upon the information supplied by these models, the compressor and the decompressor try to predict the structure of the elements.

We represent the element models by a list of their states. In each state, its type, the list of outgoing transitions, and a pointer to the most probable (or deterministic) transition that leads out of the state are stored. The element states contain also a pointer to the "nested" element model. The transitions contain a frequency counter, and a pointer to their end state.

Initially, the element models consist of one initial state and one attribute state. During the processing of the data, new states are added to the list of states, and new transitions are added to the states. In each model, a pointer to the current state is stored.

A specific situation occurs when an element state (representing a nested element) is encountered in the model. In that case, the initial state of the nested model is entered. When the nested element ends (i.e. an accepting state is visited in its model), we return back to the element state of the parent model.

Since the elements are nested in XML documents, we store the models in a stack to reflect this structure. Each time a new element is encountered, its model is pushed onto the stack. Similarly, when the element ends, the model is removed from the stack.

Chapter 9

Experimental evaluation

In this chapter, we evaluate the performance of Exalt on several types of XML documents. A comparison to some of the other XML-conscious compressors (XMill and XMLPPM), as well as to the general-purpose compressors (Gzip, Bzip2, and Sequitur) is presented. We also examine the performance of the simple and adaptive modeling strategies in terms of compression and decompression times, and compression ratios achieved.

An interesting general-purpose compressor involved in our tests is Sequitur (see Section 3.6.2). Because it is based on a syntactical compression technique that is in many ways similar to the grammar-based codes, which we have used in our compression scheme, it surely deserves a special attention. Thanks to its ability to identify the hierarchical structure of the data, Sequitur performs extremely well on structured and semi-structured inputs (such as sources of computer programs). It was an interesting test for us to see how well it stacked up when applied to XML.

The experiments were performed on a Linux, 1.5GHz Pentium 4 machine with 256MB of main memory. In the tests, the compressors were run under their default settings.

9.1 Our corpus

In our tests, we used the XML documents that came with the XMLPPM compressor, and added several others to them to make the corpus more rich. In Table 9.1, the documents in the corpus are listed, and divided into three categories depending on their type.

Textual documents have a rather simple structure (small amount of elements and attributes) and relatively long character data content. We used four Shakespearean plays[1] and two computer tutorials (`python` and `emacs`) for the tests. While the Shakespearean plays are fairly structured (they contain several types of elements), the tutorials consist of only one element with long character data content, and allow us to evaluate the performance of grammar-based codes on textual data.

Regular documents consist mainly of structured and regular mark-up, and short character data content. Typically, the values of structural entropy are small (far below 0.10).

[1] `http://www.ibiblio.org/xml/examples/shakespeare/`

Often, the documents have a repetitive structure (`periodic`, `stats1`, `weblog`).

Textual documents		
`antony`	246 KB	"The Tragedy of Antony and Cleopatra" marked up as XML
`errors`	134 KB	"The Comedy of Errors" marked up as XML
`hamlet`	273 KB	"The Tragedy of Hamlet, Prince of Denmark" marked up as XML
`much_ado`	197 KB	"Much Ado about Nothing" marked up as XML
`python`	124 KB	Python tutorial (LaTeX source enclosed with one XML tag)
`emacs`	43 KB	Emacs editor tutorial (text enclosed with one XML tag)
Regular documents		
`periodic`	114 KB	Periodic table of the elements in XML
`stats1`	654 KB	One year statistics of baseball players (1)
`stats2`	599 KB	One year statistics of baseball players (2)
`tpc`	281 KB	TPC-D benchmark database[2] transformed to XML
`weblog`	869 KB	Apache Web server log transformed to XML
Irregular documents		
`pcc1`	175 KB	Formal proofs transformed to XML (1)
`pcc2`	247 KB	Formal proofs transformed to XML (2)
`qual2003`	414 KB	MeSH[3] qualifier records
`sprot`	10 KB	SwissProt[4] representation of protein structure
`tal1`	717 KB	Safety-annotated assembly language converted to XML (1)
`tal2`	498 KB	Safety-annotated assembly language converted to XML (2)
`tal3`	246 KB	Safety-annotated assembly language converted to XML (3)
`treebank`	6 KB	Parsed English sentences from Wall Street Journal
`w3c`	197 KB	XML language specification
File	Size	Description

Table 9.1: Our XML corpus

Irregular documents have complex and irregular structure. The documents in this category can be characterized by fairly large structural entropy values (0.50 and more). Some of the documents contain quite a lot of character data (for example, `w3c` and `qual2003`), while the others—often computer-generated and not very legible for humans—do not contain any character data at all (for example, `pcc1` and `tal1`).

[4] `http://www.nlm.nih.gov/mesh/meshhome.html`

[4] `http://www.expasy.ch/sprot`

[4] `http://www.tpc.org`

9.2 Performance results

9.2.1 Compression effectiveness

In this section, we evaluate the compression results achieved by our XML compressor, and compare its performance to the general-purpose compressors and the XML-conscious compressors.

Tables 9.2, 9.3, and 9.5 summarize the compression results of the individual compressors on textual, regular, and irregular data, respectively. The compression results are reported in bits per original XML character. For each document, the best compression is in **bold**, and the worst compression is in *italic*. At the bottom of each table, the average bit rate is listed. The results for both the simple modeling (Exalt-S) and the adaptive modeling (Exalt-A) are presented.

9.2.1.1 Textual documents

On textual data, XMLPPM achieves the best overall results, competing only with Bzip. On average, Exalt makes it to the third position, beating the remaining compressors.

On Shakespearean plays, Exalt performs consistently about 5-10% better than XMill, and about 15% better than Gzip. This is a rather optimistic result for us, since it indicates that the grammar-based compression might be a rival to the dictionary-based techniques on textual data. From this point of view, we were particularly interested in the results on the documents `emacs` and `python`. These documents consist of only one element with long character data content, and have been included in the corpus to test the performance of the compressors on unstructured text data. It follows from the results that the syntactical compressors (Exalt and Sequitur) compress the documents roughly the same as the dictionary-based compressors represented by Gzip and XMill. However, in a comparison to Bzip and XMLPPM, both the syntactical and the dictionary-based compressors lag about 25% behind.

File	Gzip	Bzip	Sequitur	XMill	XMLPPM	Exalt-S	Exalt-A
`hamlet`	*2.267*	1.647	2.000	2.153	**1.590**	1.932	1.977
`errors`	*2.141*	1.630	2.020	2.058	**1.566**	1.970	2.018
`much_ado`	*2.099*	1.534	1.889	1.961	**1.481**	1.808	1.875
`antony`	*2.160*	1.549	1.900	2.039	**1.489**	1.879	1.898
`emacs`	2.755	2.432	*2.967*	2.759	**2.262**	2.934	2.920
`python`	2.658	2.256	*2.767*	2.659	**2.121**	2.709	2.702
Average	*2.347*	1.841	2.257	2.272	**1.752**	2.205	2.232

Table 9.2: Performance on textual data

Although the Shakespearean plays contain a fair amount of XML mark-up, the adaptive modeling yields about 2% worse results than the simple modeling. The reason is that the

elements in these documents form a rather irregular structure, which causes the escape mechanism to be applied quite often in the adaptive modeling. Therefore, the compression deteriorates a little.

During our experiments on textual data, it became obvious that XMLPPM and Bzip compressors would be extremely difficult to beat for the remaining compressors involved in the tests. This was confirmed by our experiments with structured XML data.

9.2.1.2 Regular documents

While the compression rates ranges from 1.5 to 2.8 bits per character in the case of textual documents, the compressors perform considerably better on the regular documents, often achieving compression rates below 0.5 bits per character.

Again, XMLPPM and Bzip perform the best. XMill yields considerably better results than on textual data, and approaches the compression rates of Bzip and XMLPPM. Using the simple modeling, Exalt performs about 16% worse than XMill on average, which was a sort of let-down for us. And although the adaptive modeling improves on the simple modeling about 7% on average, the overall results were far away behind our expectations.

File	Gzip	Bzip	Sequitur	XMill	XMLPPM	Exalt-S	Exalt-A
`stats2`	*0.750*	0.338	0.498	0.403	**0.288**	0.484	0.384
`tpc`	*1.475*	1.100	1.347	1.144	**1.092**	1.322	1.313
`stats1`	*0.798*	0.368	0.535	0.425	**0.314**	0.516	0.422
`periodic`	*0.603*	0.404	0.510	0.421	**0.370**	0.472	0.487
`weblog`	*0.337*	**0.173**	0.262	0.217	0.190	0.251	0.251
Average	*0.793*	0.477	0.630	0.522	**0.451**	0.609	0.571

Table 9.3: Performance on regular data

In three occasions (`stats1`, `stats2`, and `tpc`), the adaptive modeling yielded better results than the simple modeling. On the remaining documents, it performed the same, or even worse (`periodic`). To find out why is this possible, we examined the data generated during the simple and the adaptive modeling. We observed that by using the adaptive modeling instead of the simple modeling, the amount of the data that has to be compressed can be reduced by 10-40%, depending on the structural properties of the document. However, it shows that—no matter how big the reduction is—the grammar-based coding in itself is powerful enough to eliminate the redundant information, too.

As an experiment, we compressed the output of the simple and the adaptive modeling using Gzip and Bzip. The results are summarized in Table 9.4.

Gzip compressed the transformed data much better than raw XML data. The simple modeling improved the average compression rate of Gzip by 10%, and the adaptive modeling even by 20%. Still, Gzip performed about 10-15% worse in a comparison to our grammar-based compressor. In the case of Bzip, the improvement was only negligible: 3% for both the simple and adaptive modeling. As opposed to Gzip (and similarly to our

File	Exalt-S	Exalt-A	Gzip-S	Gzip-A	Bzip-S	Bzip-A
`stats2`	0.484	0.384	*0.600*	0.468	**0.322**	**0.322**
`tpc`	1.322	1.313	*1.429*	1.388	1.090	**1.088**
`stats1`	0.516	0.422	*0.634*	0.492	0.352	**0.350**
`periodic`	0.472	0.487	*0.569*	0.564	**0.384**	0.389
`weblog`	0.251	0.251	*0.302*	0.293	0.174	**0.172**
Average	0.609	0.571	*0.707*	0.641	**0.464**	**0.464**

Table 9.4: Comparison of the simple and the adaptive modeling

compressor), Bzip succeeds in discovering the structural redundancy within the data, and the adaptive transformation does not bring any substantial compression gain.

9.2.1.3 Irregular documents

On irregular documents, XMLPPM performed the best on average. However, it was beaten by Bzip and XMill in several occasions. The compression rates tend to be surprisingly small, often below 0.3 bits per character. We believe that this is caused by the fact that most of the documents (except for `qual2003` and `w3c`) contain only a short character data content, if any.

File	Gzip	Bzip	Sequitur	XMill	XMLPPM	Exalt-S	Exalt-A
`pcc1`	*0.361*	0.206	0.306	0.214	**0.186**	0.301	0.317
`treebank`	1.782	1.524	*2.057*	1.249	**1.171**	1.735	1.932
`pcc2`	*0.311*	0.166	0.290	**0.164**	0.168	0.274	0.269
`tal2`	*0.321*	0.149	0.246	0.183	**0.147**	0.252	0.271
`tal3`	*0.328*	0.195	0.306	0.200	**0.175**	0.312	0.322
`qual2003`	*0.650*	**0.378**	0.556	0.437	0.393	0.534	0.544
`tal1`	*0.312*	**0.121**	0.211	0.164	0.139	0.213	0.238
`w3c`	2.139	1.721	2.174	*2.273*	**1.592**	2.190	2.226
Average	*0.776*	0.558	0.768	0.610	**0.497**	0.726	0.765

Table 9.5: Performance on irregular data

The adaptive modeling performed about 5% worse than the simple modeling on the irregular documents, which was a rather expected behavior. Because it is difficult to predict the structure of the documents in the adaptive modeling, the escape mechanism is used frequently. And because the escaping carries a penalty in coding effectiveness, the overall compression performance worsens.

Probably the most complex document in our corpus is the XML language specification (`w3c`), which contains a very irregular structure and long character data content. Also, additional XML mark-up (such as comments, CDATA sections and entity declarations) is

used heavily. When applied to this document, only two compressors (Bzip2 and XMLPPM) managed to achieve compression rates below 2 bits per character, while the remaining compressors had considerable difficulties. Surprisingly enough, XMill performed the worst, leaving Gzip and our compressor (using the simple modeling) about 5% behind.

9.2.2 Compression time

Besides the compression ratios, we measured also the compression times for the documents in our corpus. In Table 9.6, the results are summarized.

File	Gzip	Bzip2	Sequitur	XMill	XMLPPM	Exalt-S	Exalt-A
hamlet	**0.070**	0.240	0.900	0.080	1.960	13.380	*13.730*
errors	**0.030**	0.110	0.440	**0.030**	0.820	3.490	*3.650*
antony	0.220	**0.050**	0.810	**0.050**	1.680	*15.780*	10.220
much_ado	**0.050**	0.180	0.590	**0.050**	1.230	5.980	*7.160*
emacs	**0.010**	0.050	0.130	**0.010**	0.280	*1.090*	0.980
python	**0.030**	0.090	0.370	0.040	1.110	*5.710*	5.520
stats2	**0.040**	0.870	2.300	0.070	1.320	3.830	*10.070*
tpc	**0.030**	0.270	1.080	0.050	1.380	5.700	*6.090*
stats1	**0.070**	0.950	2.910	0.080	1.500	4.640	*4.790*
periodic	**0.010**	0.090	*0.460*	**0.010**	0.260	0.360	0.350
weblog	**0.060**	2.810	3.460	0.070	2.220	*15.960*	13.330
pcc1	**0.010**	0.190	0.590	0.030	0.350	0.580	*1.150*
treebank	**0.001**	**0.001**	0.020	**0.001**	0.030	0.030	*0.060*
pcc2	**0.020**	0.310	0.770	0.040	0.490	0.850	*2.880*
tal2	**0.040**	0.820	1.600	0.050	1.130	1.410	*3.230*
tal3	**0.010**	0.230	0.840	0.040	0.530	0.550	*0.930*
qual2003	**0.020**	0.430	1.540	**0.020**	1.270	*3.860*	3.760
tal1	**0.050**	1.500	2.460	0.080	1.620	2.200	*6.070*
w3c	**0.040**	0.140	0.570	**0.040**	1.670	*11.510*	11.100
Average	**0.043**	0.491	1.490	0.044	1.097	5.100	*5.530*

Table 9.6: Compression times (in seconds)

It follows from the table that Exalt is the slowest of the compressors. On average, it compresses about 5 times slower that XMLPPM and Sequitur, and about 10 times slower than Bzip. While the loss is not so significant on structured documents (`periodic`, `pcc`, `tal`), the compression times increase substantially on long documents and on documents with long character data content.

The adaptive modeling causes our compressor to run slower in a comparison to the simple modeling. However, the difference is not as dramatic as expected. While the adaptive modeling is slower by a factor of 3 on documents with intricate structure (`stats2`,

`pcc2`, `tal1`), it is faster in several occasions (`antony`, `weblog`). We believe that the reason is that the adaptive modeling filters out a lot of redundant information from the data, and therefore the underlying compressor can operate faster. However, if the structure of the document is too complex, the adaptive modeling runs slower, and these benefits are lost.

It is interesting to compare the performance of our compressor to that of Sequitur. Although both compressors are based on a similar grammar-inferring technique, Sequitur evidently runs faster. There are two main reasons for this. First, Sequitur operates in a much simpler manner. It does not search for production rules that represent the longest prefix of the input data (which is a rather time consuming operation in our compressor). Instead, the data is processed character by character. Moreover, the conditions on the grammar are not as restrictive as in the case of the grammar-based coding, and can be expressed in a form of two simple constraints. Second, we know that our implementation of grammar-based coding surely is not the most optimum, and that there are several components in our compressor whose functionality can be improved. However, our goal was more to test the potentials of the grammar-based coding, and to establish an illustrative framework for further research, than to present an ideal implementation.

Chapter 10

Conclusions and further work

In this thesis, we investigated the possibilities of XML data compression based on grammar-based coding and probabilistic modeling of the XML structure. We have implemented a prototype XML compressor called Exalt and tested its performance on a variety of XML documents. The compressor is not a "black box" type of application. Rather than this, it was intended to provide a framework for experiments and further research in the field of syntactical compression of XML. Thanks to the object-oriented nature of our compressor, it is easy to extend or to improve its functionality.

The grammar-based coding, introduced recently by Kieffer and Yang [16, 15], represents a rather novel and interesting topic in the field of lossless data compression, and is still a subject of intense research. Our compressor is the first practical implementation of this promising compression technique that we are aware of.

The modeling of XML structure plays an important role in the process of XML data compression. We have implemented two modeling strategies which we call the simple modeling and the adaptive modeling, respectively. While the first strategy is quite onefold, the latter improves on it and is much more sophisticated. Moreover, the adaptive modeling gives us resources for measuring the structural complexity of XML documents.

Although our compressor is fully functional, there are several areas where there is a room for improvements or enhancements.

The grammar-based coding—as implemented in our compressor—uses the sequential algorithm for the encoding of the grammar. Kieffer and Yang have proposed other algorithms in [16, 15], which may yield better results but are more difficult to implement. Without any doubt, it would be interesting to examine the behavior of these algorithms, and to compare their influence on the properties the resulting grammar-based code.

We have proposed two modeling strategies for the compression. We expect that the performance of the compressor can be further improved by using more sophisticated modeling.

Last but not least, many of the algorithms can be optimized. For example, the most time consuming operations are those related to the building of the grammar. By optimizing these operations and the data structures involved, one may substantially improve the overall performance of the compressor.

Appendix A

User documentation

A.1 Building and installing

Exalt is suggested to be used on UNIX type of platforms (such as LINUX). It requires the Expat XML parser, version 1.95.5 or above[1], to be installed in your system. If Expat is not present, or if you are using an older version, please install the current version, which is available for example at [8].

To make the building and the installing simple, Exalt uses the GNU Autotools. The complete process of installation can be described in the following steps:

1. Copy the file `exalt-0.1.0.tar.gz` from the supplied CD to some local directory (for example to your home directory):

   ```
   $ cp exalt-0.1.0.tar.gz /home/me
   ```

2. Untar the archive by executing:

   ```
   $ cd /home/me
   $ tar zxf exalt-0.1.0.tar.gz
   ```

3. Change to the newly created directory containing the source code:

   ```
   $ cd exalt-0.1.0
   ```

4. Configure the package for your system:

   ```
   $ ./configure
   ```

5. Compile the package:

   ```
   $ make
   ```

[1] Versions of Expat prior to 1.95.5 contain a bug that causes incorrect SAX handlers to be called in some occasions. Also, some valid UTF-8 encoded documents are rejected.

6. If you have right permissions, install the library, the binaries, and the reference and API documentation:

   ```
   $ make install
   ```

The `configure` script attempts to guess correct values for various system-dependent variables used during compilation. It checks for required software and configures the Makefiles and header files to reflect the settings of your system.

There are various command-line options you can pass to the `configure` script to modify its default behavior. The most important options are listed below:[2]

- `--prefix`

  By default, the Exalt library gets installed in `/usr/local/lib`, the binaries in `/usr/local/bin`, and the header files in `/usr/local/include`. To change this behavior, use the option `--prefix=dir`. The library, the binaries, and the header files will be then installed in `dir/lib`, `dir/bin`, and `dir/include`, respectively.

- `--enable-docdir`

  By default, the documentation gets installed in `/usr/share/doc/exalt`. To install the documentation in a different directory, use the option `--enable-docdir=dir`. The documentation will be then installed in the directory `dir`.

- `--enable-expatdir`

  If the Expat parser is installed in some nonstandard location, you can use the option `--enable-expatdir=dir`. The Expat library and header files will be then expected to reside in `dir/lib` and `dir/include`, respectively.

Note. If the Exalt package has been installed in some nonstandard location (for example `/home/me/exalt`), it is important to ensure that the `LD_LIBRARY_PATH` environment variable contains the entry pointing to the library files (for example `/home/me/exalt/lib`), and the `PATH` environment variable contains the entry pointing to the location of binary files (for example `/home/me/exalt/bin`).

A.2 Using the command-line application

The distribution comes with a simple and easy to use command-line application that demonstrates the functionality of the library. It is named `exalt` and allows you to compress and decompress XML files in a convenient way.

After installation, you can test the application by typing:

[2] You can find out all the options available by running `configure` with just the `--help` option.

```
$ exalt -h
```

This command executes the application and displays the use information. (If nothing happened, please check your `PATH` environment variable.)

The arguments of `exalt` are file names and options in any order. The possible options are listed below:

- `-s .suf` (or `--suffix .suf`)

  Use the suffix `.suf` on compressed files. Default suffix is `.e`.

- `-d` (or `--decompress`)

  Decompress file(s).

- `-f` (or `--force`)

  Overwrite files, do not stop on errors.

- `-c` (or `--stdout`)

  Write on standard input.

- `-a` (or `--adaptive`)

  Use the adaptive modeling strategy for compression.

- `-x` (or `--erase`)

  Erase source files.

- `-e enc` (or `--encoding enc`)

  Set the decompressed data encoding to `enc`. Currently supported encoding is UTF-8.

- `-l` (or `--list-encodings`)

  List the recognized encodings. These encodings are not necessarily supported by `exalt`. They are present for future enhancements of the Exalt library.

- `-v` (or `--verbose`)

  Be verbose.

- `-m` (or `--print-models`)

  Display the element models. This option makes sense only if the adaptive modeling strategy is turned on. (*Beware: the models may be huge!*)

- `-g` (or `--print-grammar`)

  Display the generated grammar. (*Beware: the grammar may be huge!*)

- `-V` (or `--version`)

  Display the version number.

- `-L` (or `--license`)

  Display the version number and the software license.

- `-h` (or `--help`)

  Display the usage information.

The default action is to compress. If no file names are given, or if a file name is '`-`', `exalt` compresses or decompresses from standard input to standard output.

For example, if you want to compress `file.xml` in the verbose mode using the adaptive model, and if you wish also to display the generated grammar, use the following command:

```
$ exalt -a file.xml -v -g
```

If everything went all right, a file named `file.xml.e` is created. The original file `file.xml` can be restored by executing the command below:

```
$ exalt -d file.xml.e
```

It is also easy to use `exalt` as a filter:

```
$ cat file.xml.e | exalt -d -c | more
```

A.2.1 How the grammar is displayed

If you pass the `-g` option (or `--print-grammar`) to `exalt`, it will display the content of the grammar inferred from the input data. To demonstrate how the grammar is displayed, we compress the XML data `<a>abracadabra</a>`. The resulting grammar will be displayed as follows:

```
R0 (length: 21, used: 1x) -> '0x01 a 0x00 0x04 R1 c a d R1
                               0x00 0x03 0x05 0x0d 0x0a 0x00'
R1 (length: 4, used: 2x)  -> 'a b r a'
```

The grammar is given by the production rules R_0 (which represents the root production rule) and R_1. The production rule R_1 represents 4 characters of the input sequence, and has been used two times. The root production rule has been used once (in fact, it can't be used more times) and represents the whole input sequence which is 21 characters long. The right

sides of the production rules are displayed in apostrophes, with the symbols delimited by spaces. If it is possible, the terminal symbols are displayed as ASCII characters, otherwise they are displayed as hexadecimal numbers. Variables are displayed in the form of "`RX`" where `RX` is a production rule of the grammar.

A.2.2 How the models are displayed

When the adaptive compression strategy is turned on, or when decompressing data previously compressed using the adaptive strategy, `exalt` can display a textual information about the element models inferred from the data. This can be achieved by the `-m` option (or `--print-models`). A typical record describing one element model looks as follows:

```
Model for element "para"

Number of references:        3
Has attributes (Yes/No):     0/3
Structural entropy:          0.636514

0 (S): 2[1], *3[2]
2 (C): *1[1]
3 (E: image): *1[2]
1 (/):
```

In the first row, the name of the element is displayed. The second and third rows indicate the total number of occurrences of the element in the document, and the number of times it had/had not attributes. The fourth row evaluates the structural entropy of the element (for the definition, see Section 6.3.3). In the following rows, the structure of the element model is displayed. Each row represents one node of the element model. The structure of the (rather cryptic) notation is very simple:

First, the unique number of the node is displayed, followed by the node type. There are four possible types of nodes: the start-of-element node ("`S`"), the end-of-element node ("`/`"), the character data node ("`C`"), and the reference to another model ("`E`"). The rest of the row contains a comma-separated list of edges leading out from the node. For each edge, the number of its end node, as well as its usage count is displayed. The most often used edges are indicated by the symbol "`*`".

As an example, we describe the entry: `3 (E: image): *1[1]`. It represents a node with number 3, which is a reference to the model of element "image". There is one edge leading out from this node. This edge leads to the end-of-element node and has been used two times.

A.3 Using the library

In the following text we will demonstrate how to use the functionality of the Exalt library in a C++ program.

A.3.1 Sample application

We present a sample application that uses the Exalt library. It takes a name of an XML file on the command line, and compresses that file on standard output (be sure to redirect standard output to some file or to `/dev/null` to avoid terminal confusion). The functionality of the Exalt library is made available via the `ExaltCodec` class, so the only thing we have to do is to create an instance of this class, and to call an appropriate method of it. The methods used most often are `encode()` and `decode()`. In their basic variants, they both take two arguments: the name of the input file and the name of the output file. If the name of the input file is NULL, standard input is used. Similarly, if the name of the output file is NULL, the standard output is used.

When (de)compressing, a variety of errors can occur (the input data is not well-formed XML, the file does not exist, etc.). To report these errors, Exalt uses the mechanism of C++ exceptions. Each exception is derived from `ExaltException`, thus handling this exception will handle all the other exceptions. For more detailed description of the exceptions used by Exalt, please refer to the API documentation.

So let's have a look at the example code:

```
#include <exaltcodec.h>

int main(int argc, char **argv)
{
  ExaltCodec codec;

  if (argc < 2)
    return 1;

  try {
    codec.encode(argv[1], 0);
  }
  catch (ExaltException) {
    cerr << "Failed to compress " << argv[1];
    return 1;
  }

  return 0;
}
```

The example includes the header file `exaltcodec.h`. It is required to include this header file in order to use the library. No other header files are necessary.

We will save the example source as `example.cpp`. To compile, you have to tell the compiler where to find the library and the headers. If Exalt has been installed in the directory `/home/me/exalt`, then you should pass the following options to the compiler: `-I/home/me/exalt/include -L/home/me/exalt/lib`. Next to that, the linker should be instructed to link against the Exalt library. This can be achieved by the option `-lexalt`.

Note. If Exalt has been installed in the standard location (the default suggested by `configure`), you probably do not have to specify the options mentioned above (except `-lexalt`).

```
$ c++ -o example example.cpp -I/home/me/mystuff/include
 -L/home/me/mystuff/lib -lexalt
```

If everything went all right, a sample application has been built. We will test it on some XML data:

```
$ ./example sample.xml > tmpfile
```

If the file `sample.xml` exists in the current directory, and the XML data is well-formed, the compressed data is written to the file `tmpfile`. If you compare the sizes of `sample.xml` and `tmpfile`, the latter one should be smaller.

A.3.2 Using the PUSH interface

When compressing, the Exalt library can work in two main modes: in the PULL mode and in the PUSH mode.

The PULL interface means that the input data is read from the input stream by the coder. This is useful mainly in the occasions when you are compressing some files. The sample application presented in the previous section demonstrates the use of the PUSH interface.

The PUSH interface means that the application supplies the data to the coder by itself. This mode can be used for compression of the data that is dynamically generated. The PUSH mode has a different semantics from that of the PULL mode. In order to use the PUSH interface, you have to use following two methods of the `ExaltCodec` class: `initializePushCoder()` and `encodePush()`.

The `initializePushCoder()` method *must* be called before any calls to `encodePush()`. It initializes the coder in the PUSH mode. In its basic variant, the method requires a name of an output file as a parameter.

The `encodePush()` method encodes given block of XML data. The method takes three parameters: the pointer to the data, the length of the data, and the flag indicating the last block of data.

Note. Any attempt to use the PUSH coder in the PULL mode (or vice versa) is considered to be an error. If you attempt to use the PUSH coder in the PULL mode, `ExaltCoderIsPushException` exception is raised. Similarly, the use the PULL coder in the PUSH mode causes the `ExaltCoderIsPullException` exception to be raised.

Below you can see a fragment of code that demonstrates the PUSH functionality of the library:

```
...

ExaltCodec codec;
int length;
bool finished = false;

codec.initializePushCoder(fileName);

while (!finished) {
  data = generateData(&length, &finished);
  codec.encodePush(data, length, finished);
}

...
```

A.3.3 Using the SAX interface

Exalt can act (with some limitations) as an ordinary SAX parser on the compressed XML data. It can read the stream of compressed data and emit SAX events to the application. The SAX interface is similar to that of the Expat XML parser.

To use the SAX event facilities, you have to inherit the `SAXReceptor` class and reimplement appropriate event handling methods (for details, please refer to the API documentation). The second step is to use a special variant of the `decode()` method of the `ExaltCodec` class, which takes a pointer to an instance of `SAXReceptor` instead of the name of the output file. The optional parameter of this method is a generic pointer to the user data structure. This pointer is passed to the handlers of the receptor.

The handlers defined by the class `SAXReceptor` are listed in Table A.5. By default, the handlers do nothing unless you overload them.

For more details on the meaning of the individual handlers, as well as for the description of their parameters, please refer to the API documentation.

Note. If you are familiar with SAX parsing, you probably noticed that the list of handlers is not complete. For example, there is no way to handle the external entity references,

Handler	Description
`startElement()`	Start element event
`endElement()`	End element event
`characterData()`	Character data event
`processingInstruction()`	Processing instruction event
`comment()`	Comment event
`startCDATASection()`	Start CDATA event
`endCDATASection()`	End CDATA event
`xmlDecl()`	XML declaration event
`startDoctypeDecl()`	Start document type declaration event
`endDoctypeDecl()`	End document type declaration event
`entityDecl()`	Entity declaration event
`notationDecl()`	Notation declaration event
`defaultHandler()`	Default data (unrecognized by the parser)

Table A.5: Available SAX handlers

or the element or attribute declarations in the DTD. This is caused by the design of the Exalt library. To diminish this drawback, Exalt tries to report as most of this information as possible via the `defaultHandler()` handler.

The following example demonstrates the use of the `startElement()` handler:

```
#include <exaltcodec.h>

class MySAXReceptor : public SAXReceptor
{
public:
  void startElement(void *userData, const XmlChar *name,
                    const XmlChar **attr)
  {
    cout << "Element " << name << endl;

    if (attr)
      for (int i = 0; attr[i]; i += 2)
        cout << "Attribute " << attr[i]
             << " has value " << attr[i+1] << endl;
  }
};

int main(int argc, char **argv)
{
```

```
  ExaltCodec codec;
  MySAXReceptor receptor;

  if (argc < 2)
    return 1;

  try {
    codec.decode(argv[1], &receptor);
  }
  catch (ExaltException) {
    cerr << "Failed to decompress " << argv[1];
    return 1;
  }

  return 0;
}
```

A.3.4 Changing the default options

There are various options that affect the behavior of the library. In most occasions, there is no need to change the default settings, because the library works quite fine without any user/programmer assistance.

The library uses a static class `ExaltOptions` for setting and reading the options. This class contains methods `setOption()` and `getOption()` for setting the option values, and for reading the option values, respectively. The possible options and their values are listed below:

- `ExaltOptions::Verbose`

  Determines whether the library should be verbose. In the verbose mode, some textual information is displayed on the standard error output. Possible values:

  - `ExaltOptions::Yes` – Be verbose.
  - `ExaltOptions::No` – Don't be verbose (*default*).

- `ExaltOptions::Model`

  Determines what modeling strategy is used for the compression. Possible values:

  - `ExaltOptions::SimpleModel` – Use the simple modeling strategy for compression.
  - `ExaltOptions::AdaptiveModel` – Use the adaptive modeling strategy for compression (*default*).

- `ExaltOptions::PrintGrammar`

  Determines whether to display the grammar generated from the input data. (*Beware: The grammar may be huge!*) Possible values:

  - `ExaltOptions::Yes` – Display the generated grammar.
  - `ExaltOptions::No` – Don't display the generated grammar (*default*).

- `ExaltOptions::PrintModels`

  Determines whether to display the element models generated from the input data. The models are displayed only when using the adaptive modeling strategy. (**Beware: The models may be huge!**) Possible values:

  - `ExaltOptions::Yes` – Display the element models.
  - `ExaltOptions::No` – Don't display the element models (*default*).

- `ExaltOptions::Encoding`

  Determines the encoding of the decompressed data. Possible values:

  - The MIB[3] of the encoding (see the API documentation for details). The default encoding is either `Encodings::UTF_8` or `Encodings::UTF_16` (depends on the configuration of the Expat parser).

The options can be set by means of the static method `setOption()` of the class `ExaltOptions`. To turn the verbose mode on, for example, one should call:

```
ExaltOptions::setOption(ExaltOptions::Verbose,
   ExaltOptions::Yes)
```

You can also read the current values of the options with the static method `getOption()`. For example, the call

```
ExaltOptions::setOption(ExaltOptions::Verbose)
```

will return the current value of the "verbose" option.

A.3.5 Input and output devices

In the preceding text, the work with files was only discussed. The data was read from some file and written to another. However, the library allows you to use any "device" you want, such as the network, some database, etc. In order to make this possible, the library works with so called IO devices. From the library's point of view, file is nothing but a special type (and the most common one) of IO devices.

[3]The MIBs are unique values identifying coded character sets, and are defined in the IANA Registry [13]

There exists an abstract class `IODevice` that defines the functionality (see the API documentation) that every device has to implement. Using this class and the C++ inheritance mechanism, it is simple to create new custom devices.

How to use the new device? It is quite straightforward, since the `encode()`, `decode()` and `initializePushCoder()` methods of the `ExaltCodec` class exist also in the variants that accept pointers to the input devices as their parameters. Below you can see an example:

```
...

codec.encode(inputDevice, outputDevice);

...
```

Appendix B

Developer documentation

Exalt is written as a C++ library. We intended it to be a component-based system, which would be easy to enhance and to extend. Using the C++ inheritance mechanism, it is possible to replace or modify virtually any component of the system.

In two occasions, we made use of third-party sources or programs. For XML parsing tasks, we rely on the Expat XML parser [8], and for arithmetic coding, we use the routines originating from Moffat and Witten [22]. Expat has been chosen because of its speed and simple and clean interface, while the selected arithmetic coding routines allow us to use large dynamic alphabets (which are required by the grammar-based codes) without being unnecessarily slow.

B.1 Overview of the most important classes

In this section, we describe the most important classes present in our system. For each class, its purpose and role within the scope of the system are discussed. In most occasions, we also briefly describe the methods of the classes. For each class discussed, the related sources are listed.

Complete information on the classes used in our system can be found in the API documentation (see the supplied CD).

B.1.1 Collection classes

We make extensive use of various collection classes. A collection class is a class that can contain a number of items in a certain data structure, and perform operations on them.

Each collection class is inherited from the `Collection` template class. In this class, some minimal functionality—common to all of the collection classes—is defined.

The collection classes are implemented as templates to make it possible to store any data type. The items in the collection are nothing but pointers to the contained data. It is possible to specify whether the data referenced by these pointers should be automatically deleted or not when the items are removed from the collection.

There are four collection classes in our system: `List`, `Stack`, `Queue`, and `HashTable`.

List `list.h`

The `List` class represents a doubly-linked list. Items may be appended (`append()`) or prepended (`prepend()`) to the list, or inserted at the current position of the list (`insert()`). When removing the items, it is possible to remove the item at the current position in the list (`remove()`), or the item with specified value (`remove()` with the value as a parameter).

It is easy to traverse the list by using the methods `first()`, `last()`, `next()`, `prev()`, and `current()`.

Stack `stack.h`

The `Stack` template class provides a stack. The items may be pushed onto the stack (`push()`), or removed from the top of it (`pop()`). The method `top()` makes it possible to access the item on the top of the stack without removing it.

Queue `queue.h`

The `Queue` class represents a queue of items. The items may be enqueued to the queue (`enqueue()`), or dequeued from it (`dequeue()`). The method `first()` makes it possible to access the first item of the queue without dequeuing it.

HashTable `hashtable.h`

The `HashTable` template class represents a hash table. The template allows to specify the type of the keys, the type of the values, the size of the hash array and the type of container for storing the values in the hash array. The keys can be numbers, characters and strings.

`HashTable` makes it possible to insert the data with given key (`insert()`), and to lookup or remove the data based upon the value of the key (`find()`, `remove()`).

B.1.2 Input/output classes

The inputs and outputs are handled in a specific way in Exalt, because we wanted to support various types of input and output devices, not just files. In order to make this possible, we use a higher level abstraction to inputs and outputs, which we call IO devices.

The `IODevice` class represents an abstract predecessor to all IO devices and defines a unified interface that each device has to implement.

Each IO device has to be prepared (`prepare()`) before it is possible to write to it (`writeData()`, `putChar()`) or to read from it (`readData()`, `getChar()`). To finish the work with the device, `finish()` has to be called.

There are various methods for checking the state of the device: `isPrepared()`, `eof()`, `bytesRead()`, `bytesWritten()`, `errorOccurred()`.

FileDevice `filedevice.h`, `filedevice.cpp`

The `FileDevice` class represents an IO device for working with files. We have used the functionality of the standard `fstream` class.

The `prepare()` method takes two parameters: the name of a file, and the mode. The mode allows to specify whether the file should be opened for writing or reading. If the name `NULL`, standard input/output is used depending on the mode. The method `finish()` closes the file.

FunnelDevice `funneldevice.h`, `funneldevice.cpp`

The `FunnelDevice` class represents an IO device that acts as one-directional "funnel" or pipe. It makes the communication between two objects possible: one object writes the data to the device, and another one receives it.

In order to be able to receive the data supplied by the `FunnelDevice` in some class, we have to inherit the `UserOfFunnelDevice` class and implement the method `receiveData()`. The method `prepare()` of the device takes the pointer to a receiver object as a parameter.

B.1.3 XML compression classes

To make the use of the compressor and the decompressor simple, there are two classes in the system that make the functionality of both the compressor and the decompressor available via single function calls. Both classes—`XmlCodec` and `ExaltCodec`—are derived from the same base class (`XmlCodecBase`). Therefore, their interfaces differ only a little. However, `XmlCodec` represents a more low-level layer than `ExaltCodec` which is intended to be used by the programmers who want to use Exalt in their own programs. Actually, `ExaltCodec` uses the functionality of `XmlCodec` internally.

XmlCodec `xmlcodec.h`, `xmlcodec.cpp`

The `XmlCodec` class encapsulates the functionality of both the compressor and the decompressor. It makes it possible to encode or to decode XML data by single function calls.

The data may be compressed either in the PULL mode (`encode()`), or in the PUSH mode (`encodePush()`). The method `encode()` takes the pointers to an input device and an output device as its parameters. In the PUSH mode, only output device is required, and it has to be specified by the method `initializePushCoder()` before the data can be compressed. The method `encodePush()` takes three parameters: the pointer to a buffer of XML data, the length of the buffer, and a flag indicating the last block of data.

To decompress the data, one should call the method `decode()`. This method takes three paramaters: the pointer to an input device, the pointer to a SAX receptor, and (optionally) the pointer to a user data structure. The pointer to this data structure will be passed to the event handlers of the SAX receptor.

ExaltCodec `exaltcodec.h`, `exaltcodec.cpp`

The `ExaltCodec` class is intended to be used by the programmers who want use the functionality of our system in their own programs. Internally, `ExaltCodec` makes use of the `XmlCodec` class, and calls its methods.

Besides the standard methods that accept only IO devices as the parameters (`encode()`, `initializePushCoder()`, and `decode()`), `ExaltCodec` makes it is possible to use their modifications that accept also file names as parameters.

B.1.4 XML processing classes

There are three types of classes that deal with the processing of the XML data: `XmlParser`, `SAXEmitter`, and `SAXReceptor`. Since all of them operate with the SAX events in some way, they are derived from the `SAXBase` abstract class. In this class, the interface for SAX event handling is defined. The list of the available handlers (see the table below) is similar to that of Expat XML parser.

Handler	Description
`startElement()`	Start element event
`endElement()`	End element event
`characterData()`	Character data event
`processingInstruction()`	Processing instruction event
`comment()`	Comment event
`startCDATASection()`	Start CDATA event
`endCDATASection()`	End CDATA event
`xmlDecl()`	XML declaration event
`startDoctypeDecl()`	Start document type declaration event
`endDoctypeDecl()`	End document type declaration event
`entityDecl()`	Entity declaration event
`notationDecl()`	Notation declaration event
`defaultHandler()`	Default data (unrecognized by the parser)
`reportError()`	Report error messages
`unknownEncoding()`	Unknown character encoding

XmlParser `xmlparser.h`, `xmlparser.cpp`

The `XmlParser` class represents a SAX parser. In our compressor, we have used the Expat XML parser and have written C++ bindings to it. The parser defines several callback methods (`startElement()`, `characterData()`, etc.) and registers them so that Expat calls them when corresponding events occur. In most occasions, the callback methods simply hand over the supplied data to the modeling components of the system. An exception to this is the method `unknownEncoding()` which gets called when Expat is unable to recognize the encoding of the XML document. In this method, the `TextCodec` class is used to create (is it is possible) the character encoding conversion table for Expat parser.

The parser can operate in two basic modes. In the PULL mode (`parse()`), the parser reads the data from a specified IO device (`setInputDevice()`) by itself. Once the parsing starts, there is no way how to stop the parser unless an arror occurs or complete document is processed.

In the PUSH mode, the application itself supplies the data to the parser (`parsePush()`). In the PUSH mode, the data is supplied by blocks; the parsing stops when the last block is specified in the `parsePush()` method or an error occurs

During the parsing of the document, the parser hands over the data supplied by Expat parser to the modeling component of the system. To specify this component, the method `setXmlModel()` should be used.

SAXEmitter

`saxemitter.h`, `saxemitter.cpp`

During the decompression, the `SAXEmitter` class is used for emitting the decoded SAX events. The class itself only accepts the event data and forwards them to registered instance of class `SAXReceptor`. To register the SAX receptor, the method `setSAXReceptor()` should be called before de decompression starts.

SAXReceptor

`saxreceptor.h`

The `SAXReceptor` class can be used in conjunction with the class `SAXEmitter` to receive the emitted SAX events during the decompression. `SAXReceptor` is an abstract class (it only defines the SAX event interface), therefore it cannot be used directly. In out system, we inherit the `OutputSAXReceptor` class from it, which reconstructs XML document from the received SAX events, and outputs it using specified IO device.

The programmers are allowed to implement and use their own SAX receptors.

B.1.5 XML modeling classes

The XML structure modeling component of our system is represented by two classes: `XmlSimpleModel` and `XmlAdaptiveModel`. The first class implements the simple modeling (see Section 6.2), while the latter implements the adaptive modeling (see Section 6.3). Both classes are derived from the `XmlModelBase` abstract class, which defines a unified interfaces to all modeling classes. Thanks to that, it is possible to implement new classes that deal with the XML structure modeling.

Every XML modeling class has to implement four methods which are declared in `XmlCodecBase`. The most important methods are `manageEvent()` and `receiveData()`.

The method `manageEvent()` takes a structure describing one SAX event as the parameter. During the compression, this method is being called by the XML parser (`XmlParser`) whenever any SAX event is emitted. In `manageEvent()`, the SAX-based modeling of the XML structure is performed.

The method `receiveData()` is inherited from the `UserOfFunnelDevice` class and allows the modeling classes to receive data from the underlying grammar-based decoder. The

method takes two parameters: the pointer to the data, and the length of the data. Each time the grammar-based decoder recovers new data, it is made available via this method. Therefore, this method is important during the decompression, since it reconstructs SAX events from the decoded data.

During the compression, the modeling classes generate data that is compressed by the grammar-based coder. The underlying grammar should be specified by calling the method `setGrammar()` before the compression starts.

During the decompression, SAX events are being reconstructed from the data supplied by the grammar-based decoder. These events are emitted by means of a SAX emitter which should be specified before the decompression starts (`setSAXEmitter()`).

XmlSimpleModel

`xmlsimplemodel.h`, `xmlsimplemodel.cpp`

The `XmlSimpleModel` class implements the simple modeling, as described in Section 6.2. During the compression, the SAX events supplied by the XML parser are transformed such that the structure of the document is encoded using structural symbols. The elements and attributes are stored in a hash table and encoded using numeric identifiers. The current position within the document is represented by a stack of the nested elements.

During the decompression, the structure is recovered based upon the decoded structural symbols. Because the data is supplied by short blocks by the grammar-based decoder, it has to be buffered. Depending on the current decoding context, the instance of the class may find itself in various states. Based on this state, and on the data read, corresponding SAX events are constructed. Because many SAX events contain structured information (for example, the `xmlDecl` event contains the version of XML being used, the character encoding, and the "standalone" flag; the `startElement` event contains the name of the element and a list of its attributes and their values; ...), the resolved pieces of information have to be enqueued during the decompression. Only when complete information is decoded, the SAX event can be emitted.

XmlAdaptiveModel

`xmladaptivemodel.h`, `xmladaptivemodel.cpp`

The `XmlAdaptiveModel` class improves on the `XmlSimpleModel` and implements the adaptive modeling, as described in Section 6.3. During the compression and the decompression, the structure of the element is modeled. The structure is encoded by structural symbols, as in the case of `XmlSimpleModel`, but only when it is necessary. Moreover, the range of structural symbols is reduced, because the meaning of them is context-dependent. This causes many problems to arise during the decompression, since special care has to be taken of correct interpretation of the symbols.

In `XmlAdaptiveModel`, two separate hash tables are used to store the names of elements and attributes.

During the processing of the document (compressed or uncompressed), models of individual elements are being built. One model is represented and maintained by the `ElementModeler` class. The models are constructed incrementally, by adding new states

and transitions to them if necessary. The element models are stored in a stack, to reflect the hierarchy of the nested elements.

ElementModeler `elementmodel.h`, `elementmodel.cpp`

The `ElementModeler` class represents one element model. The model consists of a nodes and transitions. Unique numeric identifiers and frequency counts are added to the transitions. Each node is assigned a list containing the transitions that lead out of it. Furthermore, the pointer to the most frequently used transition that leads out of the node, is stored in each node.

There are four possible types of nodes: `StartNode`, `EndNode`, `CharactersNode`, and `ElementNode`. The element nodes represent the nested elements, and contain pointers to the corresponding instances of `ElementModeler`. Initially, each model consists only of isolated start node and end node.

In each element model, the current position is stored. When an element node is encountered, the node is pushed onto a stack of nodes, so that the position is remembered. When the nested element ends, the position is popped from the stack and the processing can resume.

The element models are constructed and stored in the `XmlAdaptiveModel` class. There are several methods for building of the models and their updating.

The method `moveToDesiredNode()` attempts to move (using just one transition) to the node with desired type from the current node. If such a node does not exist, it is created (end nodes are coalesced to one node to save memory). Depending on the return value of the method, various actions may be performed by `XmlAdaptiveModel`. For example, if the node did not exist, or if the transiton was not the most frequent one, the NACK structural symbol is output.

The method `moweForward()` moves along the most frequent transitions. The method `followEdge()` behaves in a similar fashion, except that is moves along edges with specified identifiers.

In each element model, the number of occurrences of the corresponding element with and without attributes is stored. Based upon these counts, the occurrence of attributes is predicted. The method `setAttributes()` is used to specify that the element contains attributes, or not. The method `hasAttributes()` returns `true` if the attributes are expected to occur, and `false` otherwise.

There are various other methods available: `getStartNode()` (get the pointer to the start node), `getEndNode()` (get the pointer to the end node), `getCurrentNode()` (get the pointer to the current node), `getElementName()` (get the name of the element modeled), `increaseRefCount()` (increase the reference count of the element model), `getRefCount()` (get the reference count), `computeStructuralEntropy()` (compute structural entropy of the element model), `getStructuralEntropy()` (get the value of structural entropy), `print()` (print a textual representation of the model).

B.1.6 Grammar-related classes

The majority of the grammar-related operations is implemented by the `KYGrammar` class. In the class itself, many small and one-purpose data structures are used: `Rule` (one production rule), `RuleSet` (the set of production rules), `TerminalDigrams` (hash table of digrams starting with a terminal symbol), `VariableDigrams` (hash table of digrams starting with a variable), etc. For more detailed information, refer to the file `kydefs.h` or to the API documentation.

KYGrammar

`kydefs.h`, `kygrammar.h`, `kygrammar.cpp`

The `KYGrammar` class implements the greedy irreducible grammar transform for the construction of the grammar, and the sequential algorithm for its encoding. Three Reduction Rules are also implemented in the class.

During the compression or the decompression, the data is supplied to the grammar by calling the method `append()`, which appends one symbol to the input data queue. The data in the queue is processed by means of the method `eatData()`: the longest possible prefix of the data is searched, and one symbol (terminal symbol or variable) is appended to the root rule (`appendToRootRule()`). After that, Reduction Rules 1-3 are applied if necessary (`reductionRule1()`, `reductionRule2()`, `reductionRule3()`). During the compression, the appended symbol is arithmetically encoded (the necessary frequency statistics of the symbols is maintained by an instance of class `Context`). During the decompression, the symbol is delivered to the modeling component (`XmlSimpleModel` or `XmlAdaptiveModel`) via a `FunnelDevice`.

To ensure that all the data in the queue is processed by the grammar at the end of the compression or the decompression, the method `flush()` should be called.

In case that the grammar becomes too large (8MB is the upper bound on its size), it is purged (`purge()`) and new grammar is constructed.

B.1.7 Arithmetic coding classes

Arithmetic coding represents an important component of a grammar-based compressor. In our implementation, we have used arithmetic coding routines originating from Moffat and Witten [22]. Because the sources were in C, we have rewritten them into object-oriented C++. The arithmetic coding and the statistics module were separated in the original implementation, and so they are in our compressor.

Context

`context.h`, `context.cpp`

The purpose of the `Context` is to maintain the alphabet used during arithmetic coding, and to store the cumulative frequencies of the symbols. The alphabet can be dynamic. To store the cumulative frequencies, Fenwick implicit tree data structure is used (for details, refer to [22]). In this representation, n words are required to represent an n-symbol alphabet, and the frequency counts are calculated in $\Theta(\log n)$ time for each symbol in the alphabet.

In the constructor of the class, the size of the initial alphabet has to be specified. New symbols can be installed by means of the method `installSymbol()`.

To arithmetically encode one symbol, the method `encode()` can be used. In this method the probability of the symbol is estimated, and the underlying arithmetic coder is instructed to encode it. Conversely, to decode one symbol, the method `decode()` can be used. In this method, the underlying arithmetic decoder is instructed to decode the target probability, and the symbol is recovered based upon this probability.

At the end of the encoding, the method `encodeEndOfMessage()` should be used to encode the end-of-message symbol. This symbol informs the arithmetic decoder that complete message has been decoded.

ArithCodec `arithdefs.h`, `arithcodec.h`, `arithcodec.cpp`

The `ArithCodec` class implements the functionality of both the arithmetic coder and the decoder. It allows to arithmetically encode the symbols of the source message based upon the probability estimates supplied by the `Context`, or to decode the symbols from the encoded data.

During the coding or the decoding, the data is written to (or is read from) some IO device. To specify this IO device, the methods `setOutputDevice()` and `setInputDevice` are present in the class.

Before the encoding starts, the coder should be initialized by a call to the method `startOutputtingBits()`. After the encoding is over, the method `doneOutputtingBits()` has to be called to shut down the coder. Similar methods—`startInputtingBits()` and `doneInputtingBits()`—should be called also before and after the decoding.

To encode and decode one symbol, the class contains the methods `arithmeticEncode()` and `arithmeticDecode()`, respectively. These are called from within the `Context` class.

At compile time (see the file `defs.h`), it can be specified whether the arithmetic coding routines should use multiplicative operations, or fixed precision arithmetic (shifts and adds). The low precision arithmetic causes the coder to run faster, but the average length of the generated codeword may be slightly longer. By default, low precision arithmetic is used.

B.1.8 Exception classes

In our system, we make use of the C++ exception mechanism. Many operations (such as those related to inputs and outputs) may fail during the compression or the decompression, and for various reasons. By raising appropirate exceptions in these situations, the exception states can be handled conveniently.

Each exception is inherited from the class `ExaltException`. There are several types of exceptions:

- IO exceptions (`ExaltIOException`)
- XML parsing exceptions (`ExaltParserException`)

- Compression and decompression exceptions (`ExaltCompressionException`)
- Character encoding exceptions (`ExaltEncodingException`)
- Fatal error exceptions (`ExaltFatalErrorException`)

Each of these groups contains other exceptions that describe the exception states more accurately. For example, when the format of the input file is not recognized by the decompressor, the exception `ExaltUnknownFileFormatException`, which is derived from `ExaltCompressionException`, is raised.

B.1.9 Other classes

Besides the "core" classes, we make use of several other classes, which perform mostly support tasks. In the following text, the most important of these classes are described.

Fibonacci

`fibonacci.h`, `fibonacci.cpp`

The `Fibonacci` static class implements several methods for encoding nonnegative integers using order-2 Fibonacci codes, and their decoding. The integers may be encoded either into an integer variable (`encode()`, `decode()`), or into character buffer (`encodeToBuffer()`, `decodeFromBuffer()`).

The `Fibonacci` class is used by the modeling component of the compressor for encoding the identifiers of elements and attributes. In adaptive modeling, Fibonacci codes are also used for encoding the values of the distance counter.

TextCodec

`textcodec.h`, `textcodec.cpp`

The `TextCodec` class is used for character encoding conversions during the compression and the decompression. The functionality of the class is required when the underlying Expat parser is unable to recognize the encoding in the declaration of the input XML document. In such a situation, the parser hands over the name of the encoding to the text codec, and expects the text codec to fill in the encoding structure which describes the mapping from the document character encoding to UTF-8.

The `TextCodec` operates with so called MIBs, which are numeric identifiers uniquely assigned to the character encoding by IANA organization [13]. The method `getMIB()` can be used to get the MIB of the character encoding based upon its name. To test whether the text codec knows or is able to convert the encoding, it is possible to use the methods `knowsMIB()` or `isAbleToConvert()`.

There are several character encoding conversion methods contained in the `TextCodec` class: `fillInMapArray()` (fill in the Expat encoding structure), `convert()` (convert one character to specified encoding), `output()` (convert one character or string to specified encoding and output it using given IO device).

The current functionality of `TextCodec` is very limited, but is is easy to enhance it by adding the support for new character encodings. At present, `TextCodec` supports the following character encodings: `UTF-8`, `UTF-16`, `US-ASCII`, `ISO-8859-1`, and `ISO-8859-2` (only for input).

Options

`options.h`, `options.cpp`

The `Options` static class allows to specify options that affect the behavior of the compressor and the decompressor. The key methods of the class are `setOption()` for setting the value of a certain option, and `getOption()` for accesing the value of a certain option.

There are five possible types of options: `Verbose` (verbose mode), `Model` (the simple modeling/the adaptive modeling), `PrintGrammar` (display the generated grammar), `PrintModels` (print the element models), and `Encoding` (the output character encoding[1]).

[1] Currently unsupported

Bibliography

[1] N. Bradley. *XML Kompletní průvodce*. Grada Publishing, 2000. In Czech.

[2] M. Burrows and D. J. Wheeler. A Block Sorting Lossless Data Compression Algorithm. Technical report, Digital Equipment Corporation, Palo Alto, California, 1994.

[3] M. Cannataro, G. Carelli, A. Pugliese, and D. Saccà. Semantic Lossy Compression of XML Data. In *Knowledge Representation Meets Databases*, 2001.

[4] J. Cheney. Compressing XML with Multiplexed Hierarchical PPM Models. In *Proceedings of IEEE Data Compression Conference*, pages 163–72, 2001.

[5] J. G. Cleary and W. J. Teahan. Unbounded Length Contexts for PPM. *Computer Journal*, 40:67–75, 1997.

[6] J. G. Cleary and I. H. Witten. Data Compression Using Adaptive Coding and Partial String Matching. *IEEE Transactions on Communication*, 32:396–402, 1984.

[7] G. V. Cormack and R. N. S. Horspool. Data Compression Using Dynamic Markov modelling. *The Computer Journal*, 30:541–550, 1984.

[8] Expat XML Parser. URL: `http://expat.sourceforge.net`.

[9] H. Fernau. Learning XML Grammars. *Machine Learning and Data Mining in Pattern Recognition MLDM'01*, 2123:73–87, 2001.

[10] M. Girardot and N. Sundaresan. Millau: An Encoding Format for Efficient Representation and Exchange of XML Documents over the WWW. In *Proceedings of the 9th international World Wide Web conference on Computer networks*, pages 747–765, 2000.

[11] E. R. Harold and W. S. Means. *XML v kostce*. Computer Press, 2002. In Czech.

[12] D. S. Hirschberg and D. A. Lelewer. Context Modeling for Text Compression. In *Image and Text Compression*, pages 113–145, 1992.

[13] IANA Character Set Names Registry. URL: `http://www.iana.org/assignments/character-sets`.

[14] Intelligent Compression Technologies (ICT). XML-Xpress. URL: `http://www.ictcompress.com/products_xmlxpress.html`.

[15] J. C. Kieffer and E.-H. Yang. Efficient Universal Lossless Data Compression Algorithms Based on a Greedy Sequential Grammar Transform – Part One: Without Context Models. *IEEE Transactions on Information Theory*, 46:755–777, 2000.

[16] J. C. Kieffer and E.-H. Yang. Grammar Based Codes: A New Class of Universal Lossless Source Codes. *IEEE Transactions on Information Theory*, 46:737–754, 2000.

[17] J. Kosek. *XML pro každého*. Grada Publishing, Prague, 2000. In Czech.

[18] D. A. Lelewer and D. S. Hirschberg. Data Compression. *ACM Computing Surveys*, 19(3), 1987.

[19] M. Levene and P. Wood. XML structure compression. Technical Report BBKCS-02-05, Birkbeck College, University of London, 2002.

[20] H. Liefke and D. Suciu. XMill: an Efficient Compressor for XML Data. In *Proceedings of 2000 ACM SIGMOD Conference*, pages 153–164, 2000.

[21] B. Melichar and J. Pokorný. Data Compression. (Revised version of the report DC-92-04). Survey Report DC-94-07, Department of Computers, Czech Technical University, 1994.

[22] A. Moffat and I. H. Witten. Arithmetic Coding Revisited. *ACM Transactions on Information Systems*, 16:256–294, 1998.

[23] T. M. Mover. Enumerative Source Encoding. *IEEE Transactions on Information Theory*, 19:73–77, 1973.

[24] C. G. Nevill-Manning and I. H. Witten. Compression and Explanation Using Hierarchical Grammars. *Computer Journal*, 40:103–116, 1997.

[25] C. G. Nevill-Manning and I. H. Witten. Identifying Hierarchical Structure in Sequences: A Linear-time Algorithm. *Journal of Artificial Intelligence Research*, 7:67–82, 1997.

[26] SAX: A Simple API for XML. URL: `http://www.saxproject.org`.

[27] C. E. Shannon. A Mathematical Theory of Communication. *The Bell System Technical Journal*, 27:379–423, 623–656, 1948.

[28] P. Tolani and J. R. Haritsa. XGrind: A Query-friendly XML Compressor. In *Proceedings of 18th IEEE International Conference on Data Engineering*, pages 225–234, 2002.

[29] World Wide Web Consorcium. Document Object Model (DOM) Level 1 Specification (Second Edition). URL: http://www.w3.org/TR/2001/REC-DOM-Level-1/.

[30] World Wide Web Consorcium. Extensive Markup Language (XML) 1.0. URL: http://www.w3.org/TR/2000/REC-xml-20001006.

[31] World Wide Web Consorcium. WAP Binary XML Content Format. URL: http://www.w3.org/TR/wbxml/.

[32] World Wide Web Consorcium. XML Path Language (XPath) 1.0. URL: http://www.w3.org/TR/xpath/.

[33] World Wide Web Consorcium. XML Schema Part 0: Primer. URL: http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/.

[34] XML Solutions. XMLZip. URL: http://www.xmls.com.