Università degli Studi di Palermo
Dipartimento di Matematica e Applicazioni
Dottorato di Ricerca in Matematica e Informatica
XXIII° Ciclo - S.S.D. Inf/01

Université Paris-Est
École doctorale MSTIC
Thése de doctorat en Informatique

Optimal Parsing for Dictionary Text Compression

Author: LANGIU Alessio
Ph.D. Thesis in Computer Science
April 3, 2012

PhD Commission:

Thesis Directors:	*Prof.* CROCHEMORE Maxime	*University of Paris-Est*
	Prof. RESTIVO Antonio	*University of Palermo*
Examiners:	*Prof.* ILIOPOULOS Costas	*King's College London*
	Prof. LECROQ Thierry	*University of Rouen*
	Prof. MIGNOSI Filippo	*University of L'Aquila*
Referees:	*Prof.* GROSSI Roberto	*University of Pisa*
	Prof. ILIOPOULOS Costas	*King's College London*
	Prof. LECROQ Thierry	*University of Rouen*

Università degli Studi di Palermo
Dipartimento di Matematica e Applicazioni
Dottorato di Ricerca in Matematica e Informatica
XXIII Ciclo - S.S.D. Inf/01

Université Paris-Est
École doctorale MSTIC
Thése de doctorat en Informatique

Optimal Parsing for Dictionary Text Compression

Author: LANGIU Alessio
Ph.D. Thesis in Computer Science
April 3, 2012

Ph.D. Candidate	LANGIU Alessio	Alessio Langiu
Thesis Director	*Prof.* RESTIVO Antonio	
Ph.D. Course Coord.	*Prof.* DI PIAZZA Luisa	

Summary

Dictionary-based compression algorithms include a parsing strategy to transform the input text into a sequence of dictionary phrases. Given a text, such process usually is not unique and, for compression purpose, it makes sense to find one of the possible parsing that minimize the final compression ratio. This is the parsing problem. An optimal parsing is a parsing strategy or a parsing algorithm that solve the parsing problem taking account of all the constraints of a compression algorithm or of a class of homogeneous compression algorithms. Compression algorithm constrains are, for instance, the dictionary itself, i.e. the dynamic set of available phrases, and how much a phrase weights on the compressed text, i.e. the number of bits of which the codeword representing such phrase is composed, also denoted as the encoding cost of a dictionary pointer.

In more than 30th years of history of dictionary based text compression, while plenty of algorithms, variants and extensions appeared and while dictionary approach to text compression became one of the most appreciated and utilized in almost all the storage and communication processes, only few optimal parsing algorithms were presented. Many compression algorithms still leaks optimality of their parsing or, at least, proof of optimality. This happens because there is not a general model of the parsing problem that includes all the dictionary based algorithms and because the existing optimal parsing algorithms work under too restrictive hypothesis.

This work focus on the parsing problem and presents both a general model for dictionary based text compression called Dictionary-Symbolwise Text Compression theory and a general parsing algorithm that is proved to be optimal under some realistic hypothesis. This algorithm is called

Dictionary-Symbolwise Flexible Parsing and it covers almost all of the known cases of dictionary based text compression algorithms together with the large class of their variants where the text is decomposed in a sequence of symbols and dictionary phrases.

In this work we further consider the case of a free mixture of a dictionary compressor and a symbolwise compressor. Our Dictionary-Symbolwise Flexible Parsing covers also this case. We have indeed an optimal parsing algorithm in the case of dictionary-symbolwise compression where the dictionary is prefix closed and the cost of encoding dictionary pointer is *variable*. The symbolwise compressor is any classical one that works in linear time, as many common variable-length encoders do. Our algorithm works under the assumption that a special graph that will be described in the following, is well defined. Even if this condition is not satisfied, it is possible to use the same method to obtain almost optimal parses. In detail, when the dictionary is LZ78-like, we show how to implement our algorithm in linear time. When the dictionary is LZ77-like our algorithm can be implemented in time $O(n \log n)$. Both have $O(n)$ space complexity.

Even if the main aim of this work is of theoretical nature, some experimental results will be introduced to underline some practical effects of the parsing optimality in terms of compression performance and to show how to improve the compression ratio by building extensions Dictionary-Symbolwise of known algorithms. Finally, some more detailed experiments are hosted in a devoted appendix.

Resume

Les algorithmes de compression de données basés sur les dictionnaires incluent une stratégie de parsing pour transformer le texte d'entrée en une séquence de phrases du dictionnaire. Etant donné un texte, un tel processus n'est généralement pas unique et, pour comprimer, il est logique de trouver, parmi les parsing possibles, celui qui minimise le plus le taux de compression finale.

C'est ce qu'on appelle le problème du parsing. Un parsing optimal est une stratégie de parsing ou un algorithme de parsing qui résout ce problème en tenant compte de toutes les contraintes d'un algorithme de compression ou d'une classe d'algorithmes de compression homogène.

Les contraintes de l'algorithme de compression sont, par exemple, le dictionnaire lui-même, c'est-à-dire l'ensemble dynamique de phrases disponibles, et combien une phrase pèse sur le texte comprimé, c'est-à-dire quelle est la longueur du mot de code qui représente la phrase, appelée aussi le coût du codage d'un pointeur de dictionnaire.

En plus de 30 ans d'histoire de la compression de texte par dictionnaire, une grande quantité d'algorithmes, de variantes et d'extensions sont apparus. Cependant, alors qu'une telle approche de la compression du texte est devenue l'une des plus appréciées et utilisées dans presque tous les processus de stockage et de communication, seuls quelques algorithmes de parsing optimaux ont été présentés.

Beaucoup d'algorithmes de compression manquent encore d'optimalité pour leur parsing, ou du moins de la preuve de l'optimalité. Cela se produit parce qu'il n'y a pas un modèle général pour le problème de parsing qui inclut tous les algorithmes par dictionnaire et parce que les parsing optimaux

existants travaillent sous des hypothèses trop restrictives.

Ce travail focalise sur le problème de parsing et présente à la fois un modèle général pour la compression des textes basée sur les dictionnaires appelé la théorie Dictionary-Symbolwise et un algorithme général de parsing qui a été prouvé être optimal sous certaines hypothèses réalistes. Cet algorithme est appelé Dictionary-Symbolwise Flexible Parsing et couvre pratiquement tous les cas des algorithmes de compression de texte basés sur dictionnaire ainsi que la grande classe de leurs variantes où le texte est décomposé en une séquence de symboles et de phrases du dictionnaire.

Dans ce travail, nous avons aussi considéré le cas d'un mélange libre d'un compresseur par dictionnaire et d'un compresseur symbolwise. Notre Dictionary-Symbolwise Flexible Parsing couvre également ce cas-ci. Nous avons bien un algorithme de parsing optimal dans le cas de compression Dictionary-Symbolwise où le dictionnaire est fermé par préfixe et le coût d'encodage des pointeurs du dictionnaire est variable. Le compresseur symbolwise est un compresseur symbolwise classique qui fonctionne en temps linéaire, comme le sont de nombreux codeurs communs à longueur variable.

Notre algorithme fonctionne sous l'hypothèse qu'un graphe spécial, qui sera décrit par la suite, soit bien défini. Même si cette condition n'est pas remplie, il est possible d'utiliser la même méthode pour obtenir des parsing presque optimaux. Dans le détail, lorsque le dictionnaire est comme LZ78, nous montrons comment mettre en œuvre notre algorithme en temps linéaire. Lorsque le dictionnaire est comme LZ77 notre algorithme peut être mis en œuvre en temps $O(n \log n)$ où n est le longueur du texte. Dans les deux cas, la complexité en espace est $O(n)$. Même si l'objectif principal de ce travail est de nature théorique, des résultats expérimentaux seront présentés pour souligner certains effets pratiques de l'optimalité du parsing sur les performances de compression et quelques résultats expérimentaux plus détaillés sont mis dans une annexe appropriée.

Contents

Introduction

Data compression concerns with transformations thought a more concise data representation. When such transformation is perfectly invertible we have a lossless data compression, otherwise, a lossy compression. Since data preservation is usually required for textual data, lossless data compression is often called text compression. On the opposite, usually working on visual data, such as the images or video, on sound data and on many other data domains, a certain degree of approximation is allowed to the compression - decompression process in favour of a stronger compression, i.e. a smaller compression ratio.

Roughly speaking, compression ratios greater than a certain threshold, given by the percentage of information contained in the data, are reachable by text compression techniques as they strip just redundancy in the text. Stronger compressions imply data approximation because part of their information is lost along the compression process. The quantity of information in a certain data or, more precisely, the average information inside the data provided by a certain source is called entropy. The entropy ratio is then a limit for text compression, i.e. it is a lower bound for the compression ratio.

Entropy, data complexity and data compression are therefore bidden all together. Indeed, fundamental and seminal methods for dictionary based compression, such as the Lempel' and Ziv's methods, were firstly introduced as text complexity measures.

Lempel' and Ziv's methods are still the basis of almost all recent dictionary compression algorithms. More in detail they are the LZ77 and the LZ78 compression methods, i.e. the Lempel and Ziv compression methods presented in 1977 and 1978 years. They are the firsts relevant dictionary methods that use dynamic dictionaries. Static dictionary compression was already known as it is a side effect of some code and transducer theories. Static dictionary compression was the topics of many works around '70, as the text substitution methods in Schuegraf' and Heaps's work (1974) or in the Wagner's work (1973).

Dictionary-based compression include, more or less explicitly, a parsing strategy that transform the input text into a sequence of dictionary phrases. Since that usually the parsing of a text is not unique, for compression purpose it makes sense to find one of the possible parsing that minimizes the final compression ratio. This is the parsing problem.

In the foundational methods (such as the work of Lempel and Ziv), the parsing problem was not immediately clear as it was confused with the dictionary building strategy. The overall compression algorithms strictly imposed the parsing of the text. As soon as many variant of such methods appeared along the sequent years, like the Storer' and Szymanski's variant (1982) or the Welch's variant (1984), the maintenance of the dynamic dictionary was clearly divided from the text parsing strategy and, in the meantime, the importance of coupling a kind of compression on the symbols different from the compression for the dictionary phrases taken place. This last feature was initially undervalued in the theoretical model of the compression processes.

The first parsing problem model by graphs is due to Schuegraf et al. (see [33]). They associated a graph with as much nodes as many characters constituting the text string and one edge for each dictionary phrase. In this model, the optimal parsing is obtained by using shortest path algorithms on the associated graph. But this approach was not recommended for practical purpose as it was considered too time consuming. Indeed, the graph can have quadratic size with respect to the text length.

A classic formalization of a general dictionary compression algorithm was proposed by Bell et al. in the late 1990, focusing on just three points. The dictionary definition, the dictionary phrases encoding method and the parsing strategy. This model does not acquire all the richness of many advanced dictionary based compression algorithms as it does not take account of the symbolwise compression.

Recently, in chronological order, [12], [25], [9] and [5] revised both a more general dictionary compression algorithms definition and the graph based parsing problem model, and they also presented optimal solutions. A similar optimal parsing result for the LZ77-like dictionary case, were independently obtained in [17] where the symbolwise feature is anyhow not considered.

The study of free mixtures of two compressors is quite involved and it represents a new theoretical challenge. Free mixture has been implicitly or explicitly used for a long time in many fast and effective compressors such as the gzip compression utility (see [30, Sect. 3.23]), the PkZip Archiving Tool (see [30, Sect. 3.23]), the Rolz Compressor[1], and the MsZip cabinet archiving software (see [30, Sect. 3.7]), also known as CabArc. In order to glance at compression performances see the web page of Mahoney's challenge[2] about large text compression. In detail, there are two famous compression methods that can work together: the *dictionary* encoding and the *statistical* encoding, which are also called *parsing* (or macro) encoding and *symbolwise* encoding, respectively. The fact that these methods can work together is commonly accepted in practice even if the first theory of Dictionary-Symbolwise methods started in [12].

This work focus on the parsing problem and introduce a twofold result; a general model for dictionary based text compression called Dictionary-Symbolwise theory and a general parsing algorithm that is proved to be optimal under some realistic hypothesis. The Dictionary-Symbolwise model extend both the Bell dictionary compression formalization and the Schuegraf parsing model based on graphs to better fit to the wide class of common compression algorithms.

The parsing algorithm we present is called Dictionary-Symbolwise Flexible Parsing and it covers almost all the cases of the dictionary based text compression algorithms together with the large class of their variants where the text is parsed as a sequence of symbols and dictionary phrases. It exploits the prefix closed property of common dictionaries, i.e. both the LZ77 and LZ78-like dictionaries. It works for *dynamic* dictionaries and *variable*

[1]For an example see the RZM Order-1 ROLZ Compressor by Christian Martelock (2008) web site: `http://encode.ru/threads/1036`. Last verified on December 2011

[2]Matt Mahoney's Large Text Compression Benchmark web page: `http://mattmahoney.net/dc/text.html`. Last verified on December 2011

costs either for dictionary phrases and symbols. His main part concerns with the construction of a smallest subgraph that guarantees parsing optimality preservation and then a shortest path is found by using a classic single source shortest path approach.

The symbolwise encoding can be any classical one that works in linear time, as many common variable-length encoders do. Our algorithm works under the assumption that a special graph that will be described in the following, is well defined. Even if this condition is not satisfied it is possible to use the same method to obtain almost optimal parses. In detail, when the dictionary is LZ78-like, we show that our algorithm has $O(n)$ complexity, where n is the size of the text. When the dictionary is LZ77-like our algorithm can be implemented in time $O(n \log n)$. Both above solutions have $O(n)$ space complexity.

The advantages of using Dictionary-Symbolwise methods are both theoretical and practical. The theoretical advantage with respect to the pure dictionary compression is described in Section 2.5.

Even if the main aim of this work is of theoretical nature, some experimental results will be introduced to underline some practical effects of the parsing optimality in compression performance and some more detailed experiments are hosted in a devoted appendix.

Chapter 1

Background

This chapter concerns with some well known concepts from the field of the Information Theory, that are fundamental to deal with data compression. Information Theory literature is quite large by now. We remand to [30], [31] and [32] books for a comprehensive look on background notions and standard techniques of data compression. We report here just few prerequisites to make readers comfortable with notation and concepts we will use in the rest of this thesis.

1.1 Self-Information and Entropy

A foundational concept for Information Theory is the Shannon's self-information definition. It is a quantitative measure of information. Let A be a probabilistic event, i.e. A is the set of outcomes of some random experiment. If $P(A)$ is the probability that the event A will occur, then the self-information associated with A is given by: $i(A) = -log_2 P(A)$ bits.

If we have a set of independent events A_i, which are sets of outcomes of some experiment $\mathcal{S}$, which sample space is $S = \cup A_i$, then the average self-information associated with the random experiment $\mathcal{S}$ is given by $H(\mathcal{S}) = \sum P(A_i) i(A_i) = -\sum P(A_i)\ log_2 P(A_i)$ bits. This quantity is called the *entropy* associated with the experiment.

Now, if the experiment is a source $\mathcal{S}$ that emits a string S of symbols over the alphabet $\Sigma = \{1, ..., m\}$, i.e. $S = s_1 s_2 s_3 ...$ with $s_i \in \Sigma$, then the sample space is the set of all the strings the source can produce, i.e. the set of all the possible sequences of alphabet symbols of any length. The *entropy* of the source $\mathcal{S}$ is given by

$$H(\mathcal{S}) = \lim_{n\to\infty} \frac{1}{n} G_n$$

with

$$G_n = -\sum_{i_1=1}^{m} \dots \sum_{i_n=1}^{m} P(s_1 = i_1, ..., s_n = i_n) log P(s_1 = i_1, ..., s_n = i_n).$$

If each symbol in the string is independent and identically distributed (*iid*), then we have that

$$G_n = -n \sum_{i=1}^{m} P(i) log P(i) \quad \text{and} \quad H(\mathcal{S}) = -\sum_{i=1}^{m} P(i) log P(i).$$

When the symbol probabilities are not independent from each other, the distribution follow an intrinsic model of probability of the source. In this case, the above two entropy equations are not equal and we distinguish them calling the latter *first order entropy*.

The probability distribution over the symbols of a source is not usually *a priori* known and the best we can do is to infer the distribution looking inside some sample string. Obviously, the underlay assumption is that the source is a *ergodic* source, i.e. its output at any time has the same statistical properties.

The Markov process is the common way to model the source distribution when symbols are not independent each other. In this case we have that each new outcome depends on all the previous one. A *discrete time Markov chain* is a special type of Markov model for those experiments where each observation depends on just the k previous one, i.e.

$$P(s_n | s_{n-1}, s_{n-2}, ...) = P(s_n | s_{n-1}, s_{n-2}, ..., s_{n-k})$$

where the set $\{s_{n-1}, s_{n-2}, ..., s_{n-k}\}$ is the state of the k-order Markov process. The entropy of a Markov process is defined as the average value of the entropy at each state, i.e.

$$H(\mathcal{M}_k) = -\sum_{s_{n-k}} P(s_{n-k}) \sum_{s_{n-k+1}} P(s_{n-k+1} | s_{n-k}) \sum_{s_{n-2}} P(s_{n-2} | s_{n-3}, ..., s_{n-k}) ...$$

$$... \sum_{s_{n-1}} P(s_{n-1}|s_{n-2},...,s_{n-k}) \sum_{s_n} P(s_n|s_{n-1},...,s_{n-k}) log P(s_n|s_{n-1},...,s_{n-k})$$

where $s_i \in \Sigma$. In the data compression field is common to refer to the state $\{s_{n-1},...,s_{n-k}\}$ of previous symbols by using the string $s_{n-k}...s_{n-1}$ called the context of length k of s_n.

Empirical Entropy

The k-order empirical entropy (see [16]) is the measure of information of a text T based on the number of repetitions in T of any substring w of length k. Let be

$$H_k(T) = -\frac{1}{n} \sum_{w \in \Sigma^k} n_w \left[\sum_{\sigma \in \Sigma} \frac{n_{w\sigma}}{n_w} log \left(\frac{n_{w\sigma}}{n_w} \right) \right]$$

where $n = |T|$, Σ is the alphabet, $w \in \Sigma^k$ is a string over Σ of length k, $w\sigma$ is the string w followed by the symbol σ and n_w is the number of occurrences of w in T.

This quantity does not refer to a source or to a probabilistic model, but it only depends from the text T. The empirical entropy is used to measure the performance of compression algorithms as a function of the string structure, without any assumption on the input source.

1.2 Entropy Encoding

Entropy encoding, statistical codes or simbolwise codes, as they are also called, are those compression methods that use the expectation value to reduce the symbol representation. There are static model as well as adaptive or dynamic models. They are usually coupled with a probabilistic model that is in charge of providing symbol probability to the encoder. Common models use symbol frequencies or the symbol context to predict the next symbol.

The simplest statistical encoder is the 0th order arithmetic encoding. It considers all the symbols as if they are independent each other. The adaptive version use to estimate symbol probability with the frequency of occurrence of any symbol in the already seen text.

Huffman coding keep count of the symbol frequencies while read the text or by preprocessing it, and then assign shorter codewords of a prefix-free codes to the most occurring symbols accordingly with the Huffman tree.

Arithmetic coding

The basic idea of arithmetic coding is to represent the entire input with an interval of real numbers between 0 and 1. The initial interval is [0, 1) and then it is divided in slots accordingly to the symbol probability. Once that a symbol is encoded, the corresponding slot of the interval is divided again accordingly with the adapted symbol distribution. While the active slots becomes finer and finer, its internal points bit representation grows. As soon as the extremal points of the slot have an equal upper part in their bit representation, these bits are outputted and the slot is scaled to be maintained under the finite precision of the representation of real values inside the machine. As any point of a slot represents an infinite set of infinite strings, all having the same prefix, one of them is chosen when the input string terminate to be outputted. The termination ambiguity is usually handled by using a special terminal symbol.

The output in length of arithmetic codes can be accurately estimated by using the Markov process entropy or the empirical entropy. Moreover, it is proved that their compression ratio tends asymptotically to the entropy of the source. Obviously, better results are obtained when higher order model are used, because the model gets closer to the real source model and the compression ratio tends more quickly to the source entropy. On the other side, higher order models need more time and space to be handled. Furthermore, since that all the models of order equal or greater than the source model are asymptotically the same and since that they perform about the same even in practical cases, it is a crucial point for a data compressor to estimate the order of the source and balancing the practical implementation constrains.

1.3 Encoding of Numbers and Commas

Encoding is a fundamental stage of many compression algorithms which consists of uniquely representing a sequence of integers as a binary sequence. In the simplest case the encoder makes use of a code, that is a mapping of the positive integers onto binary strings (codewords), in order to replace each value in input with its corresponding codeword. Codewords can be of variable-lengths as long as the resulting code is uniquely decodable, e.g. the prefix-free codes. Prefix-free property requires that no codeword can be equal to a prefix of another codeword. Several codes have been proposed that achieve small average codeword-length whenever the frequencies of the input integers are monotonically distributed, such that smaller values occur more frequently than larger values.

The unary encoding of an integer n is simply a sequence of n 1s followed by a 0. Unary encoding is rarely used as stand-alone tool and is often component of more complex codes. It achieves optimality when integer frequencies decrease exponentially as $p(i+1) \leq p(i)/2$.

The Elias codes are a recursively defined family of encoders. Each member is defined using the previous one starting from unary encoding as base element. The representation of an integer x consists of a first part, where the bit-length of the codeword is specified. Then, the standard binary representation of x, without the most significant bit, follows. The first useful Elias encoder is the well-known γ-code, which stores the prefix-part in unary. Elias δ-code differs from γ-code also with a γ-code for first-part of the codewords, rather than using unary code.

1.4 Dictionary Methods

Dictionary compression methods are based on the substitution of phrases in the text with references to dictionary entries. A dictionary is an ordered collection of phrases and a reference to a dictionary phrase is usually called dictionary pointer. The idea is that if encoder and decoder share the same dictionary and, for most of the dictionary phrases, the size of the representation in output of a dictionary pointer is less that the size of the phase itself, then a shorter representation of the input text is obtained replacing phrases with pointers. In order to proceed to phrase substitution, the text has to be divided into a sequence of dictionary phrases. Such decomposition is called *parsing* and is not usually unique. For compression purpose it makes sense to find one of the possible parsing that minimizes the final compression ratio. This is the parsing problem.

The foundational methods in dictionary compression class are Lempel' and Ziv's LZ77 and LZ78 algorithms that will be extensively considered along this thesis. Lempel' and Ziv's methods are the basis of almost all the dictionary compression algorithms. They are the firsts relevant dictionary methods that use dynamic dictionaries.

The LZ77 method consider the already seen text as the dictionary, i.e. it uses a dynamic dictionary that is the set of all the substrings of the up to the current position. The dictionary pointers refers to the occurrence of the phrase in the text by using the couple (*length*, *offset*) corresponding to an occurrence of the phrase. As phrase are usually repeated more then once in the text and since that pointer with smaller offset are usually smaller, occurrence close to the current position are preferred. Notice that this dictionary is both prefix and suffix closed. The parsing strategy use the greedy approach to find the longest phrase in the dictionary equal to a prefix of the rest of the text.

The LZ78 dictionary is a subset of the LZ77 one. It is prefix-closed but it is not suffix-closed. Each dictionary phrases is equal to another dictionary phrase with a symbol appended at the end. Exploiting this property, dictionary is implemented as a ordered collection of couple (*dictionary pointer*, *symbol*), where the dictionary pointer refers to a previous dictionary phrase and the dictionary contains the empty string. As long as the input text is analyzed, the longest match between the dictionary and the text is selected to form a new dictionary phrase. Indeed, a new couple is formed by this selected dictionary phrase and the symbol in the text that follow the occurrence of this phrase. This new dictionary phrase is added to the dynamic dictionary and it is chosen also to be part of the parsing of the text.

More detail about these method will be reported in next chapters.

Chapter 2

Dictionary-Simbolwise Text Compression

Many dictionary based compression algorithms and their practical variants use to parse the text as a sequence of both dictionary phrases and symbols. Different encoding are used for those two kinds of parse segments. Indeed, many variants of the classic Lempel and Ziv algorithms allows to parse the text as a free mixture of dictionary phrases and symbols. This twofold nature of the parsing segments was not caught in classic formulation of the dictionary based compression theory. In this chapter we recall the classical dictionary compression algorithm formulation and the classic model of the parsing problem before to present the more general framework for Dictionary-Symbolwise compression that better fit to almost all the dictionary based algorithms.

2.1 Dictionary Compression

In [4] it is possible to find a survey on Dictionary methods and of Symbolwise methods and a description of the deep relationship among them (see also [3, 11, 30, 31]).

Definition 2.1. A dictionary compression algorithm, as noticed in [4], can be fully described by:

1. The dictionary description, i.e. a static collection of phrases or a

complete algorithmic description on how the dynamic dictionary is built and updated.

2. The encoding of dictionary pointers in the compressed data.

3. The parsing method, i.e. the algorithm that splits the uncompressed data in dictionary phrases.

We notice that any of the above three points can depend on each other, i.e. they can be mutually interdependent.

As reader can notice, above three points are general enough to describe both static and dynamic dictionary and both static and variable costs for the dictionary phrase representation in the output data. We want now to focus on its third point where the parsing is defined as just a sequence of dictionary pointers. The drawback of this constrain is to bring to an overuse of formalism as it is not easy to describe the role played by symbols. Let us show this effect by examples. The characterization of the classic LZ77 and LZ78 algorithms according to the above Definition 2.1 are stated in what follows.

LZ77 characterization

Given a text $T \in \Sigma^*$ and processing it left to right, at time i the text up to the ith character has been read.

1. Let be $D_i = \{wa$, such that $w \in Fact(T[i - P : i])$ and $|wa| \leq Q\}$, where P is the maximum offset for text factors, Q is the maximum length for dictionary phrases and $a \in \Sigma$. $T[i - P : i]$ is called the *search buffer* and $T[i : i + Q]$ is called the *look-ahead buffer*.

2. The dictionary phrase $wa = T[i-p : i-p+q]a$ is represented by the vector (p, q, a) where p is the backward offset in the search buffer at which the phrase wa appears. The threefold vector (p, q, a) is coded by three fixed length sequence of bits where p has length $log_2(P)$, q has length $log_2(Q)$ and a is represented with 8 bits by using the ascii code for symbols.

3. The parsing follows a simple rule. At any time i, if the i is the position in the text at which the already chosen parsing ends up, the match between the longest prefix of the *look-ahead buffer* $T[i : i+Q]$ and a dictionary phrase $wa \in D_i$ is chosen to be the last phrase of the parsing of the text up to the position $i + |wa|$ (i.e. up to the position $i + q$, for wa represented by (p, q, a)). Otherwise, the already chosen parsing overpass position i in the text and nothing has to be addicted to the current parsing. For instance if the parsing of the text up to the position i is the ordered set of dictionary phrases $\{(p_1, q_1, a_1), (p_2, q_2, a_2), ..., (p_j, q_j, a_j)\}$, then the parsing up to the position $i+q$ is $\{(p_1, q_1, a_1), (p_2, q_2, a_2), ..., (p_j, q_j, a_j), (p_{j+1}, q_{j+1}, a_{j+1}) = (p, q, a)\}$.

LZ78 characterization

Given a text $T \in \Sigma^*$ and processing it left to right, at time i the text up to the ith character has been read. The algorithm maintain a table of phrases T_i initialized with the empty word at row 0, i.e. $T_0 = [\epsilon, ...]$.

1. The dictionary D_i is defined as $D_i = \{wa$ such that $w \in T_i$ and $a \in \Sigma\}$. Let $j \geq i$ be the position over the text at which the already chosen parsing ends up. If $j > i$, then the dictionary doesn't change, i.e $T_i = T_{i+1}$ and therefore $D_i = D_{i+1}$. Otherwise, for $j = i$, a dictionary phrase $wa \in D_i$ is chosen to be part of the parsing and to be included in the set of phrases T_{i+1}. Let wa_1 be this phrase. wa_1 is inserted in the table T at the first empty row, therefore T_{i+1} becomes $T_i \cup wa_1$ and $D_{i+1} = D_i \cup wa_1\Sigma$. D_i is prefix closed at any time by construction. When the table T_i becomes full, some deletion strategy has to take place in order to allow dictionary adaptation to the input text. Typically, some long entry are removed, preserving the prefix closed property.

2. The dictionary phrase $wa \in D_i$ is represented by the couple (x, a) where x is the row number of w in T_i. The couple (x, a) is encoded by using a fixed length encodings for the integer x followed by the ascii value of a.

3. At any time i, if the i is the position in the text at which the already chosen parsing ends up, the match between the longest prefix of the uncompressed text $T[i : n]$ and a dictionary phrase $wa \in D_i$ is chosen to be the last phrase of the parsing of the text up to the position $i + |wa|$. Otherwise, the already chosen parsing overpass position i in the text and nothing has to be addicted to the current parsing. For instance if the parsing of the text up to the position i is the ordered set of dictionary phrases $\{(x_1, a_1), (x_2, a_2), ..., (x_j, a_j)\}$, then the parsing up to the position $i + |wa|$ is $\{(x_1, a_1), (x_2, a_2), ..., (x_j, a_j), (x_{j+1}, a_{j+1}) = (x, a)\}$.

2.2 Dictionary-Symbolwise Compression

We propose a new definition for the class of dictionary based compression algorithms that takes account of the presence of single characters beside to dictionary phrases. For this reason we chose to call them dictionary-symbolwise algorithms. The following definition is an extension of the above Definition 2.1 due to Bell et al. (see [4]) and it refines what was presented in [9, 12, 25].

Definition 2.2. A dictionary-symbolwise compression algorithm is specified by:

1. The dictionary description.
2. The encoding of dictionary pointers.
3. The symbolwise encoding method.
4. The encoding of the flag information.
5. The parsing method.

A *dictionary-symbolwise* algorithm is a compression algorithm that uses both dictionary and symbolwise compression methods. Such compressors may parse the text as a free mixture of dictionary phrases and literal characters, which are substituted by the corresponding pointers or literal codes, respectively. Therefore, the description of a dictionary-symbolwise algorithm also includes the so called *flag information*, that is the technique used to distinguish the actual compression method (dictionary or symbolwise) used for each segment or factor of the parsed text. Often, as in the case of LZSS (see [36]), an extra bit is added either to each pointer or encoded character to distinguish between them. Encoded information flag can require less space than one bit according to the encoding used.

For instance, a dictionary-symbolwise compression algorithm with a fixed dictionary $D = \{ab, cbb, ca, bcb, abc\}$ and the static symbolwise codeword assignment $[a = 1, b = 2, c = 3]$ could compress the text *abccacbbabbcbcbb* as $F_d1F_s3F_d3F_d2F_d1F_d4F_d2$, where F_d is the flag information for dictionary pointers and F_s is the flag information for the symbolwise code.

More formally, a parsing of a text T in a dictionary-symbolwise algorithm is a pair $(parse, Fl)$ where *parse* is a sequence $(u_1, \cdots, u_s)$ of words such that $T = u_1 \cdots u_s$ and where Fl is a boolean function that, for $i = 1, \ldots, s$ indicates whether the word u_i has to be encoded as a dictionary pointer or as a symbol. See Table 2.1 for an example of dictionary-symbolwise compression.

LZ77 characterization

Given a text $T \in \Sigma^*$ and processing it left to right, at time i the text up to the ith character has been read.

1. Let be $D_i = \{w$, such that $w \in Fact(T[i-P : i])$ and $|w| < Q\}$, where P is the maximum offset for text factors, Q is the maximum length for dictionary phrases. Let $T[i - P : i]$ be called the *search buffer* and $T[i : i + Q]$ be called the *look-ahead buffer*.

Input	*ab*	*c*	*ca*	*cbb*	*ab*	*bcb*	*cbb*
Output	F_d1	F_s3	F_d3	F_d2	F_d1	F_d4	F_d2

Table 2.1: Example of compression for the text *abccacbbabbcbcbb* by a simple Dictionary-Symbolwise algorithm that use $D = \{ab, cbb, ca, bcb, abc\}$ as static dictionary, the identity as dictionary encoding and the mapping $[a = 1, b = 2, c = 3]$ as symbolwise encoding.

2. The dictionary phrase $w = T[i - p : i - p + q]$ is represented by the vector (p, q) where p is the backward offset in the search buffer at which the phrase w appears. The vector (p, q), also called dictionary pointer, is coded by two fixed length sequence of bits where p has length $log_2(P)$ and q has length $log_2(Q)$.

3. Any symbol $a \in \Sigma$ is represented with 8 bits by using the ascii code for symbols.

4. The flag information is not explicitly encoded as it is completely predictable. Indeed, after a dictionary pointer there is a symbol and after a symbol there is a dictionary pointer.

5. The parsing impose a strictly alternation between dictionary pointers and symbols. At any time i, if the i is the position in the text at which the already chosen parsing ends up, the match between the longest prefix of the *look-ahead buffer* $T[i : i + Q]$ and a dictionary phrase $w \in D_i$ is chosen to be outputted followed by the mismatch symbol. For instance, if w is the longest match between the dictionary and the look-ahead buffer, with w represented by the couple (p, q), then $F_d\ p\ q\ F_s\ T_{i+|w|}$ are concatenated to the parsing. Otherwise, the already chosen parsing overpass position i in the text and nothing has to be addicted to the current parsing.

This new formalization allows to describe dictionary algorithms in a more natural way. Moreover, it allows to easily describe those variants where just a single point of the algorithm is different. For instance, let us focus on LZSS, the LZ77 based algorithm due to Storer and Szymanski of the '82 (see [36]). The main idea of this algorithm is to relax the parsing constrain about dictionary pointers and symbols alternation, allowing their us as needed.

LZSS characterization

Given a text $T \in \Sigma^*$ and processing it left to right, at time i the text up to the ith character has been read.

1. Let be $D_i = \{w$, such that $w \in Fact(T[i-P:i])$ and $|w| < Q\}$, where P is the maximum offset for text factors, Q is the maximum length for dictionary phrases. Let $T[i-P:i]$ be called the *search buffer* and $T[i:i+Q]$ be called the *look-ahead buffer*.

2. The dictionary phrase $w = T[i-p:i-p+q]$ is represented by the vector (p,q) where p is the backward offset in the search buffer at which the phrase w appears. The vector (p,q), also called dictionary pointer, is coded by two fixed length sequence of bits where p has length $log_2(P)$ and q has length $log_2(Q)$.

3. Any symbol $a \in \Sigma$ is represented with 8 bits by using the ascii code for symbols.

4. The flag information is explicitly encoded by using 1 bit conventional value. For instance, $F_d = 0$ and $F_s = 1$.

5. At any time i, if the i is the position in the text at which the already chosen parsing ends up, the match between the longest prefix of the *look-ahead buffer* $T[i:i+Q]$ and a dictionary phrase $w \in D_i$ is chosen to be outputted. For instance, if w is the longest match between the dictionary and the look-ahead buffer, with w represented by the couple (p,q), then $F_d\ p\ q$ } are concatenated to the parsing. If there is no match between dictionary and look-ahead buffer, then a single symbol is emitted as $F_s\ T[i+1]$. Otherwise, the already chosen parsing overpass position i in the text and nothing has to be added to the current parsing.

Dictionry-Symbolwise Schemes

Let now focus on the parsing point. Some dictionary compression algorithm, as LZ77 and LZ78, imposes strictly a parsing strategy while some other don't. For any dictionary compression algorithm we can build a class of variants taking fixed the first four points and changing the parsing strategy or, if it is needed, arranging a bit the first four points to allow using of another parsing. Usually the variants of the same class maintains decoder compliant, i.e. their can be decoded by the same decoding algorithm.

Algorithms of the same class can be compared looking for the the optimal parsing, i.e. the parsing that minimize compression ratio. We call *scheme* such class of algorithms.

Definition 2.3. Let a dictionary-symbolwise *scheme* be a nonempty set of dictionary-symbolwise algorithms having in common the same first four specifics, i.e. they differ from each other by the parsing methods only.

A scheme does not need to contain *all* the algorithms having the same first four specifics. Let us notice that any of the specifics from 1 to 5 above can depend on all the others, i.e. they can be mutually interdependent. The word *scheme* has been used by other authors with other meaning, e.g. scheme is sometimes used as synonymous of algorithm or method. In this thesis scheme always refers to the above Definition 2.3.

Remark 1. For any dictionary-symbolwise scheme S and for any parsing method P, a dictionary-symbolwise compression algorithm $\mathcal{A}_{S,P}$ is completely described by the first four specifics of any of the algorithms belonging to S together with the description of the parsing method P.

Let us here briefly analyze some LZ-like compression algorithms. The LZ78 algorithm is, following the above definitions, a dictionary-symbolwise algorithm. It is easy to naturally arrange its original description to a dictionary-symbolwise complaint definition. Indeed, its dictionary building description, its dictionary pointer encoding, its symbolwise encoding, its parsing strategy and the null encoding of the flag information are, all together, a complete dictionary-symbolwise algorithm definition. The flag information in this case is not necessary, because there is not ambiguity about the nature of the encoding to use for any of the parse segments of the text as the parsing strategy impose a rigid alternation between dictionary pointers and symbols. Similar arguments apply for LZ77. Later on we refer to these or similar dictionary-symbolwise algorithms that have null flag information as “pure” dictionary algorithms and to scheme having only “pure” dictionary algorithms in it as “pure” scheme.

LZW (see [30, Section 3.12]) naturally fits Definition 2.1 of dictionary algorithms and, conversely, LZSS naturally fits Definition 2.2 on dictionary-

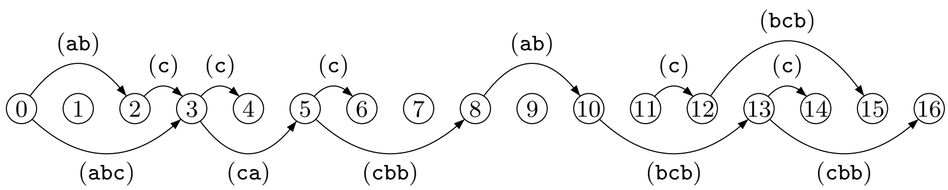

Figure 2.1: Graph $G_{D,T}$ for the text $T = \texttt{abccacbbabbcbcbb}$ and for the static dictionary $D = \{\texttt{ab}, \texttt{cbb}, \texttt{ca}, \texttt{bcb}, \texttt{abc}, \texttt{c}\}$. The dictionary phrase associated with an edge is reported near the edge label within parentheses.

symbolwise algorithms as well as the LZMA algorithm (see [30, Section 3.24]).

Let us notice that sometimes the flag information may be implicitly represented. For instance, in the Deflate compression method, characters and part of the dictionary pointers (i.e. the length part of the couples (length,distance) that represent the dictionary pointers) are firstly mapped into a single codeword space (together with few control characters), and then encoded via Huffman codes belonging to just a single Huffman tree. This mapping hides the flag information that has to be considered implicitly represented, but still existing. It is easy to show how in this case the flag information is involved in the compression process. Indeed the frequency of any character related code is equal to the frequency of the character on the character space, times the frequency of the flag information for the character encoding. The same argument applies to the length-codeword frequencies. In this way, the compressed stream is a sequence of character codewords and dictionary pointer codewords bringing implicitly the flag informations.

2.3 The Graph Based Model

Extending the approach introduced for static dictionaries in [33] to the dynamic dictionary case, similarly to what it is already done in [12, 25, 9, 5], we show how to associate a directed weighted graph $G_{\mathcal{A},T} = (V, E, L)$ with any dictionary compression algorithm $\mathcal{A}$, any text $T = a_1a_2a_3 \cdots a_n$ and any cost function $C : E \to \mathbb{R}^+$ in the following way.

The set of vertices is $V = \{0, 1, \ldots, n\}$, where vertex i corresponds to a_i, i.e. the i-th character in the text T, for $1 \leq i \leq n$ and vertex 0 corresponds to the position at the beginning of the text, before any characters. The empty word ε is associated with vertex 0, that is also called the *origin* of the graph. The set of directed edges is

$$E = \{(p, q) \subset (V \times V) \mid p < q \text{ and } \exists\ w_{p,q} = T[p+1 : q] \in D_p\}$$

where $T[p+1 : q] = a_{p+1}a_{p+2} \cdots a_q$ and D_p is the dictionary relative to the p-th processing step, i.e. the step in which the algorithm either has processed the input text up to character a_p, for $p > 0$, or it has begun, for $p = 0$. For each edge (p, q) in E, we say that (p, q) is *associated with* the dictionary phrase $w_{p,q} = T[p+1 : q] \in D_p$. In the case of a static dictionary, D_i is constant along all the algorithm steps, i.e. $D_i = D_j, \forall i, j = 0 \cdots n$. Let L be the set of edge labels $L_{p,q}$ for every edge $(p, q) \in E$, where $L_{p,q}$ is defined as the cost (weight) of the edge (p, q) when the dictionary D_p is in use, i.e. $L_{p,q} = C((p, q))$.

Let us consider for instance the case where the cost function C associates the length in bit of the encoded dictionary pointer of the dictionary phrase $w_{p,q}$ to the edge (p, q), i.e. $C((p, q)) = \mathit{length}(\mathit{encode}(\mathit{pointer}(w_{p,q})))$, with $w_{p,q} \in D_p$. In this case the weight of a path $\mathcal{P}$ from the origin to the node $n = |T|$ on the graph $G_{\mathcal{A},T}$ corresponds to the size of the output obtained by using the parsing induced by $\mathcal{P}$. The path of minimal weight on such graph corresponds to the parsing that achieves the best compression. The relation between path and parsing will be investigated in Section 2.4.

If the cost function is a total function, then $L_{p,q}$ is defined for each edge of the graph.

Remark 2. Let us say that $G_{\mathcal{A},T}$ is *well defined* iff $L_{p,q}$ is defined for each edge (p, q) of the graph $G_{\mathcal{A},T}$.

For instance, the use of common variable-length codes for dictionary pointers, as Elias or Fibonacci codes or static Huffman codes, leads to a well defined graph. Sometimes the cost function is a partial function, i.e. $L_{p,q}$ is not defined for some p and q, and $G_{\mathcal{A},T}$ in such cases is not well defined. For instance, encoding the dictionary pointers via statistical codes, like Huffman codes or arithmetic codes, leads to partial cost functions. Indeed the encoding of pointers and, accordingly, the length of the encoded dictionary pointers may depend on how many times a code is used (i.e. in variable length codes, the codeword lengths depend either on how frequently they are used in the past for adaptive codes or on how frequently their are used in the overall compression process for offline codes like the semi static Huffman codes). In these cases the cost function depends on the parsing (it depends on the parsing chosen up to a certain position of the text or on the parsing of the whole text, respectively). Moreover, the cost function may be undefined for edges that represent phrases never used by the parsing. The latter case is still an open problem, i.e. it is not known how to find an optimal parsing strategy when the encoding costs depend on the parsing itself.

Remark 3. We call $G_{\mathcal{A},T}$ the "Schuegraf's graph" in honour of the first author of [33] where a simpler version was considered in the case of static-dictionary compression method.

We can naturally extend the definition of the graph associated with an algorithm to the dictionary-symbolwise case. Given a text $T = a_1 \dots a_n$, a dictionary-symbolwise algorithm $\mathcal{A}$, and a cost function C defined on edges, the graph $G_{\mathcal{A},T} = (V, E, L)$ is defined as follows. The vertices set is $V = \{0 \cdots n\}$, with $n = |T|$. The set of directed edges $E = E_d \bigcup E_s$, where

$$E_d = \{(p,q) \subset (V \times V) \mid p < q-1, \text{ and } \exists w = T[p+1:q] \in D_p\}$$

is the set of dictionary edges and

$$E_s = \{(q-1,q) \mid 0 < q \leq n\}$$

is the set of symbolwise edges. L is the set of edge labels $L_{p,q}$ for every edge $(p,q) \in E$, where the label $L_{p,q} = C((p,q))$. Let us notice that the cost function C hereby used has to include the cost of the flag information to each edge, i.e. $C((p,q))$ is equal to the cost of the encoding of F_d (F_s, resp.) plus the cost of the encoded dictionary phrase $w \in D_p$ (symbolwise a_q, resp.) associated with the edge (p,q) where $(p,q) \in E_d$ (E_s, resp.). Moreover, since E_d does not contain edges of length one by definition, $G_{\mathcal{A},T} = (V, E, L)$ is not a multigraph. Since this graph approach can be extended to multigraph, with a overhead of formalism, one can relax the $p < q-1$ constrain in the definition of E_d to $p \leq q-1$. All the results we will state in this thesis, naturally extend to the multigraph case.

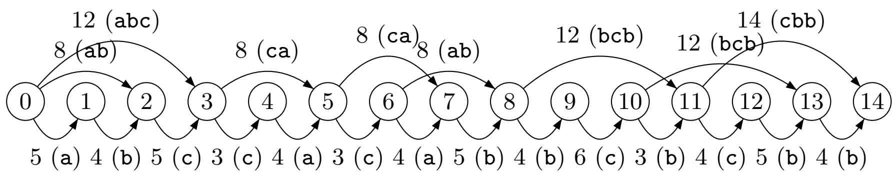

Figure 2.2: Graph $G_{\mathcal{A},T}$ for the text $T = \texttt{abccacabbcbcbb}$, for the dictionary-symbolwise algorithm $\mathcal{A}$ with static dictionary $D = \{\texttt{ab}, \texttt{abc}, \texttt{bcb}, \texttt{ca}, \texttt{cbb}\}$ and cost function C as defined in the graph. The dictionary phrase or the symbol associated with an edge is reported near the edge label within parenthesis.

2.4 On Parsing Optimality

In this section we assume that the reader is well acquainted with LZ-like dictionary encoding and with some simple statistical encodings such as the Huffman encoding.

Definition 2.4. Fixed a dictionary description, a cost function C and a text T, a *dictionary (dictionary-symbolwise) algorithm* is *optimal* within a set of algorithms if the cost of the encoded text is minimal with respect to all others algorithms in the same set. The parsing of an *optimal* algorithm is called *optimal* within the same set.

When the bit length of the encoded dictionary pointers is used as cost function, the previous definition of optimality is equivalent to the classical well known definition of bit-optimality for dictionary algorithm. Notice that the above definition of optimality strictly depends on the text T and on a set of algorithms. A parsing can be optimal for a certain text but not for an another one. Obviously, we are mainly interested on parsings that are optimal either for *all* texts over an alphabet or for classes of texts. Whenever it is not explicitly written, from now on when we talk about optimal parsing we mean optimal parsing for *all* texts. About the set of algorithm it makes sense to find sets as large as possible.

Classically, there is a bijective correspondence between parsings and paths in $G_{\mathcal{A},T}$ from vertex 0 to vertex n, where optimal parses correspond to minimal paths and vice-versa. We say that a parse (path, resp.) *induces* a path (parse, resp.) to denote this correspondence. This correspondence was firstly stated in [33] only in the case of sets of algorithms sharing the same *static* dictionary and where the encoding of pointers has constant cost.

For example the path along vertices $(0, 3, 4, 5, 6, 8, 11, 12, 13, 14)$ is the shortest path for the graph in Fig. 2.2. Authors of [12] were the first to formally extend the Shortest Path approach to dynamically changing dictionaries and variable costs.

Definition 2.5. A scheme $\mathcal{S}$ has the *Schuegraf property* if, for any text T and for any pair of algorithms $\mathcal{A}, \mathcal{A}' \in \mathcal{S}$, the graph $G_{\mathcal{A},T} = G_{\mathcal{A}',T}$ with $G_{\mathcal{A},T}$ well defined.

This property of schemes is called *property of Schuegraf* in honor of the first of the authors in [33]. In this case we define $G_{\mathcal{S},T} = G_{\mathcal{A},T}$ as the graph of (any algorithm of) the scheme. The proof of the following proposition is straightforward.

Proposition 2.4.1. *There is a bijective correspondence between optimal parsings and shortest paths in $G_{\mathcal{S},T}$ from vertex 0 to vertex n.*

Definition 2.6. Let us consider an algorithm $\mathcal{A}$ and a text T and suppose that graph $G_{\mathcal{A},T}$ is well defined. We say that $\mathcal{A}$ is *graph* optimal (with respect to T) if its parsing induces a shortest path in $G_{\mathcal{A},T}$ from the origin (i.e. vertex 0) to vertex n, with $n = |T|$. In this case we say that its parsing is *graph* optimal.

Let $\mathcal{A}$ be an algorithm such that for any text T the graph $G_{\mathcal{A},T}$ is well defined. We want to associate a scheme $\mathcal{SC}_A$ with it in the following way. Let S be the set of all algorithms $\mathcal{A}$ such that for any text T $G_{\mathcal{A},T}$ exists (i.e. it is well defined). Let $\mathcal{B}$ and $\mathcal{C}$ two algorithms in S. We say that $\mathcal{B}$ and $\mathcal{C}$ are equivalent or $\mathcal{B} \equiv \mathcal{C}$ if, for any text T, $G_{\mathcal{B},T} = G_{\mathcal{C},T}$.

We define the scheme $\mathcal{SC}_A$ to be the equivalence class that has $\mathcal{A}$ as a representative. It is easy to prove that $\mathcal{SC}_A$ has the Schuegraf property.

We can connect the definition of *graph* optimal parsing with the previous definition of $\mathcal{SC}_A$ to obtain the next proposition, which proof is a easy consequence of the Proposition 2.4.1 and the Schuegraf property of $\mathcal{SC}_A$. Roughly speaking, the graph optimality within the scheme $\mathcal{SC}_A$ implies scheme (or global) optimality.

Proposition 2.4.2. *Let us consider an algorithm $\mathcal{A}$ such that for any text T the graph $G_{\mathcal{A},T}$ is well defined. Suppose further that for a text T the parsing of $\mathcal{A}$ is graph optimal. Then the parsing of $\mathcal{A}$ of the text T is (globally) optimal within the scheme $\mathcal{SC}_A$.*

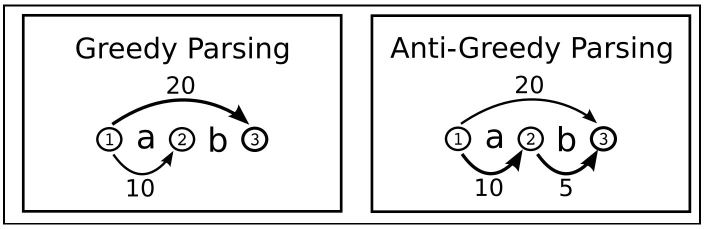

Figure 2.3: Locally but not globally optimal parsing

We have simple examples (see Figure 2.3), where a parsing of a text is graph optimal and the corresponding algorithm belongs to a scheme that has not the Schuegraf property and it is not optimal within the same scheme.

For instance, let us define a dictionary scheme where the dictionary is composed by $\langle a, ab \rangle$ if the parsing of processed text has reached an even position (starting from position 0) with costs 10 and 20 respectively. The dictionary is $\langle a, b \rangle$ if the parsing of processed text has reached an odd position with costs 5 each. Notice that now the dictionary phrase “a” has a different cost than before. The dictionary and the costs are changing as a function of the reached position, depending if this position is even or odd, and, in turn, it depends on the parsing. Therefore this scheme has not the Schuegraf property because there is not an unique Shuegraf graph $G_{A,T}$ for all the algorithms in the scheme. Indeed, given a text T and $\mathcal{A}$, $\mathcal{A}'$ in the scheme, with $G_{A,T}$, $G_{A',T}$ well defined, $G_{A,T}$ is different from $G_{A',T}$ as $\mathcal{A}$ has a different parse from $\mathcal{A}'$.

Let us now consider the text $T = ab$. As first parsing let us choose the greedy parsing, that at any reached position chooses the longest match between text and dictionary. The graph $G_{\mathcal{A},T}$for this greedy algorithm has three nodes, $0, 1, 2$, and only two edges, both outgoing 0, one to node 1 and cost 10 and another to node 2 and cost 20. The greedy parsing reaches the end of the text with this second arc which has global cost 20 and then it is graph optimal. As second parsing we choose the anti-greedy that at any reached position chooses the shortest match between text and dictionary. The graph $G_{\mathcal{A}',T}$ for this anti-greedy algorithm has three nodes, $0, 1, 2$, and three edges, two outgoing 0, one to node 1 and cost 10 and another to node 2 and cost 20 and a third outgoing 1 to node 2 and cost 5. The parsing of the anti-greedy algorithm is (a)(b) with cost 15. Therefore both the greedy and the anti-greedy parsing are graph optimal but the greedy one is not (globally) optimal.

2.5 Dictionary-Symbolwise Can Have Better Ratio

So, why should we use dictionary-symbolwise compressors?

From a practical point of view, coupling a fast symbolwise compressor to a dictionary compressor gives one more degrees of freedom to parsing, increasing compression ratio without slowing up the entire compression process. Or, at the other extreme, a dictionary compressor coupled with a powerful symbolwise compressor can speed up the decompression process without decreasing the compression ratio. This approach that mix together dictionary compression and symbolwise compression methods is already widely used in practical compression software solutions, even if it scientific basis were not clearly defined and it was treated just as a practical trick to enhance compression ratio and to take under control and improve the decompression speed. Several viable algorithms and most of the commercial data compression programs, such as *gzip*, *zip* or *cabarc*, are, following our definition, dictionary-symbolwise. Still from a practical point of view, some experimental results are showed and discussed in next section.

In this section instead we study some theoretical reasons for using dictionary-symbolwise compression algorithms.

First of all, it is not difficult to give some "artificial" and trivial example where coupling a dictionary and a symbolwise compressor give rise to a better optimal solution. Indeed let us consider the static dictionary $D = \{a, b, ba, bb, abb\}$ together a cost function C that could represents the number of bits of a possible code: $\{C(a) = 8, C(b) = 12, C(ba) = 16, C(bb) = 16, C(abb) = 4\}$.

A greedy parsing of the text $babb$ is $(ba)(bb)$ and the cost of this parsing is 32. An optimal parsing for this dictionary is $(b)(abb)$ that has cost 16. This example shows, as also the one of Figure 2.3, that a greedy parsing is not always an optimal parsing in dictionary compressors.

Let us consider further the following static symbolwise compressor that associate with the letter a a code of cost 8 and associate with the letter b a code of cost 4 that could represents the number of bits of this code. The cost of coding $babb$ following this symbolwise compressor is 20.

If we connect them in a dictionary-symbolwise compressor then an optimal parsing is $S(b)D(abb)$ where the flag information is represented by the letter S for symbolwise of next parse phrase or D that stands for dictionary. The cost of the trivially encoded flag information is one bit for each letter or phrase. Therefore the cost of this parsing is 10.

In this subsection, however, we will prove something more profound than artificial examples such the one above. Indeed, from a theoretical point of view Ferragina et al. (cf. [17]) proved that the compression ratio of the classic greedy-parsing of a LZ77 pure dictionary compressor may be far from the bit-optimal pure dictionary compressor by a multiplicative factor $\Omega(\log n/ \log\log n)$, which is indeed unbounded asymptotically. The family of strings that is used in [17] to prove this result, is a variation of a family that was used in [24].

We show in next two subsections a similar result between the bit-optimal dictionary compressor and a dictionary-symbolwise compressor. Therefore a bit optimal dictionary-symbolwise compressor can use, in some pathological situation, the symbolwise compressor to avoid them and be provably better than a simple bit optimal dictionary compressor.

LZ77 Case

Let us define these two compressors. The first is a LZ77 based compressor that allows overlaps with unbounded windows as dictionary and with a Huffman encoding on the lengths and an optimal parser. The encoding of pointers can be any of the classical intelligent encoding. We just impose an Huffman coding on the lengths.

We further denote by OPT-LZH(s) the bit length of the output of this compressor on the string s.

The same LZ77 is used as dictionary compressor in the dictionary-symbolwise compressor. Clearly we do not include the parser in the dictionary-symbolwise compressor, but, analogously as above, we suppose we have an optimal parser for the dictionary-symbolwise compressor, no matter about the description. The flag information $\{D, S\}$ is coded by a run-length encoder. The cost of a run is subdivided over all symbolwise arcs of the run, i.e. if there is a sequence of n consecutive symbolwise arcs in the optimal parsing then the cost of these n flag information S (for Symbolwise) will be in total $O(\log n)$ and the cost of each single flag information in this run will be $O(\frac{\log n}{n})$.

It remains to define a symbolwise compression method.

In the next result we could have used a PPM* compressor but, for simplicity, we use a longest match symbolwise. That is, the symbolwise at position k of the text searches for the closest longest block of consecutive letters in the text up to position $k-1$ that is equal to a suffix ending in position k. This compressor predicts the $k+1$-th character of the text to be the character that follows the block. It writes a symbol "y" (that is supposed not to be in the text) if this is the case. For otherwise it uses an escape character n (that is supposed not to be in the text) and then write down the correct character plainly. A temporary output alphabet has therefore two characters more than the characters in the text. This temporary output will be subsequently encoded by a run-length encoder (see [15]).

This is not a very smart symbolwise compressor but it fits our purposes, and it is simple to analyze.

We further denote by OPT-DS(s) the bit length of the output of this dictionary-symbolwise compressor on the string s.

Theorem 2.5.1. *There exists a constant $c > 0$ such that for every $n' > 1$ there exists a string s of length $|s| \geq n'$ satisfying*

$$\frac{\textit{OPT-LZH}(s)}{|s|} \geq c \frac{\log |s|}{\log \log |s|} \textit{OPT-DS}(s).$$

Proof. For every n' let us pick a binary word w of length $2n, n \geq n'$, $w = a_1 a_2 \cdots a_{3n}$ that has the following properties.

1. For any $i, 1 = 1, 2 \cdots n$ compressor OPT-LZH(s) cannot compress the word $a_i a_{i+1} \cdots a_{2i+n-1}$ of length $n+i$ with a compression ratio greater than 2.

2. every factor (i.e. every block of consecutive letters) of w having length $3 \log 3n$ of w is unique, i.e. it appears in at most one position inside w.

Even if it could be hard to explicitly show such a word, it is relatively easy to show that such a word exists. Indeed, following the very beginning of the Kolmogorov's theory, the vast majority of words are not compressible. A simple analogous counting argument can be used to prove that property 1) is satisfied by the vast majority of strings of length $2n$, where, for vast majority we mean that the percentage of strings not satisfying 1) decreases exponentially in n. Here, to be safer, we allowed a compression "two to one".all the n considered factors.

A less known result (see [2, 37, 13, 19, 14, 8]) says that for random strings and for any $\epsilon > 0$ the percentage of strings of length n having each factor of length $2 \log n + \epsilon$ *unique* grows exponentially to 1 (i.e. the percentage of strings not having this property decreases exponentially). Here we took as ϵ the number 1. Therefore such a string $a_1 \cdots a_{3n}$ having both properties surely exists for some $n \geq n'$.

Let us now define the word s over the alphabet $\{0, 1, c\}$ in the following way.

$$s = a_1 a_2 \cdots a_{n+1} c^{2^n} a_2 a_3 \cdots a_{n+3} c^{2^n} \cdots a_i a_{i+1} \cdots a_{2i+n-1} c^{2^n} \cdots a_{n+1} a_{n+2}.$$

Let us now evaluate OPT-LZH(s). By property 1) each binary word that is to the left or to the right of a block of 2^n c's cannot be compressed in less than $\frac{1}{2}n$ bits in a "stand-alone" manner. If one such a string is compressed by a pointer to a previous string then the offset of this pointer will be greater than 2^n and, so, its cost in bit is $O(n)$. We defined the string s in such a manner that all "meaningful" offsets are different, so that even a Huffman encoding on offsets (that we do not use, because we use Huffman codes only for lengths) cannot help. Therefore there exists a constant c' such that OPT-LZH(s) $\geq c'n^2$.

Let us now evaluate OPT-DS(s). We plan to show a parse that will give a string of cost P-DS(s) $\leq \hat{c}n \log n$ as output. Since OPT-DS(s) $\leq$ P-DS(s) then also OPT-DS(s) $\leq \hat{c}n \log n$.

The blocks of 2^n c's have all the same length. We parse them with the dictionary compressor as $(c)(c^{2^n} - 1)$. The dictionary compressor is not used in other positions in the parse P of the string s. The Huffman encoding on lengths of the dictionary compressor would pay n bits for the table and a constant number of bits for each occurrence of a block of 2^n c's. Hence the overall cost in the parse P of all blocks of letters c is $O(n)$. And this includes the flag information that consists into two bits n times.

Parse P uses the symbolwise compressor to parse all the binary strings. The first one $a_1 a_2 \cdots a_{n+1}$ costs $O(n)$ bits. Starting from the second $a_2 a_3 \cdots \cdots a_{n+3}$ till the last one, the symbolwise will pay $O(\log n)$ bits for the first $3 \log 3n$ letters and then, by property 1), there is a long run of y that will cover the whole string up to the last two letters. This run will be coded by the run-length code of the symbolwise. The overall cost is $O(\log n)$ and this includes the flag information that is a long run of S coded by the run-length of the flag information. The cost of the symbolwise compressor included the flag information over the whole string is then $O(n \log n)$, that dominates the cost of the dictionary-symbolwise parse P.

The length of the string s is $O(n2^n + n^2)$ and therefore $\log |s| = n + o(n)$ and the thesis follows. □

Remark 4. In the theorem above it is possible to improve the constants in the statement. This can be done simply using for instance a word $a_1 \cdots a_{n^2}$ instead of $a_1 \cdots a_{3n}$. It is possible to optimize this value, even if, from a conceptual point of view, it is not important.

We want to underline that the Huffman coding on the lengths is essential in this statement. At the moment we were not able to find a sequence of strings s where the dictionary-symbolwise compressor is provably better than the optimal dictionary version without using Huffman codes. It is an open question whether this is possible.

We finally notice that if the dictionary is coupled with a ROLZ technique then the optimal solution of the pure dictionary compressor reaches the same level of the dictionary symbolwise compressor. This is not surprising because the ROLZ technique is sensible to context and do not "pay" for changing the source of the text.

LZ78 Case

Matias and Sahinalp in [28] already shown that Flexible Parsing is optimal with respect to all the prefix-closed dictionary algorithms, included LZ78, where optimality stand for phrase optimality. Flexible Parsing is also optimal in the suffix-close dictionary algorithms class. Phrase optimality is equal to bit optimality under the fixed codeword length assumption, so we say just optimality. From now on we assume FP or its extension as optimal parsing and the bit length of the compressed text as coding cost function.

In this subsection we prove that there exists a family of strings such that the ratio between the compressed version of the strings obtained by using an optimal LZ78 parsing (with constant cost encoding of pointers) and the compressed version of the strings obtained by using an optimal dictionary-symbolwise parsing is unbounded. The dictionary, in the dictionary-symbolwise compressor is still the LZ78 dictionary, while the symbolwise is a simple Last Longest Match Predictor that will be described later. We want to notice here that similar results were proved in [28] between flexible parsing and the classical LZ78 and in [17] between a compressor that uses optimal parsing over a LZ77 dictionary and the standard LZ77 compressor (see also [24]).

Last but not least we notice that in this example, analogously as done in [28], we use an unbounded alphabet just to make the example clearer. An analogous result can be obtained with a binary alphabet with a more complex example.

Let us define a Dictionary-Symbolwise compressor that uses LZ78 as dictionary method, the Last Longest Match Predictor as symbolwise method, Run Length Encoder to represent the flag information and one optimal parsing method. Let us call it OptDS-LZ78. We could have used a PPM* as symbolwise compressor but Last Longest Match Predictor (LLM) fits our purposes and it is simple to analyze. LLM Predictor is just a simple symbolwise compression method that uses the last longest seen match to predict next char.

The symbolwise searches, for any position k of the text, the closest longest block of consecutive letters up to position $k-1$ that is equal to a suffix ending in position k. This compressor predicts the $(k+1)$-th character of the text to be the character that follows the block. It writes a symbol 'y´ (that is supposed not to be in the text) if this is the case. Otherwise it uses an escape character 'n' (that is supposed not to be in the text) and then writes down the correct character plainly. A temporary output alphabet has therefore two characters more than the characters in the text. This temporary output will be subsequently encoded by a run-length encoder. This method is like the Yes?No version of Symbol Ranking by P. Fenwick (see [15]).

It costs $\log n$ to represent a substring of n chars that appear after the match. For each position i in the uncompressed text if m_i is the length of the longest match in the already seen text it produces n that cost $O(\log n)$ bits as output, i.e. $C(T[i+1..i+n]) = n$ and $Cost(n) = O(\log n)$ where

$$\forall m, j \; m_i = Max_m(T[i-m..i] = T[j-m..j] \text{ with } j < i \text{ and } T[i-m-1] \neq T[j-m-1])$$

Let us consider a string S

$$S = \textstyle\sum_{z=1}^{k} 1 + ... + z = [1 + 12 + 123 + ... + 1..z + ... + 1..k]$$

that is the concatenation of all the prefixes of $1..k$ in increasing order. Let consider the string T' that is the concatenation of the first $\sqrt{k}$ suffixes of $2..k$, i.e. $T' = 2..k \cdot 3..k \cdot \ldots \cdot \sqrt{k}..k$ and a string $T = S \cdot T'$. We use S to build a dictionary formed by just the string $1..k$ and its prefixes and no more. We assume that both the dictionary and the symbolwise methods work the same up to the end of the S string, so they produce an output that is very similar in terms of space. It is not difficult to prove that an optimal LZ78 compressor would produce on T a parse having cost at least $O(k + k \log k) = O(k \log k)$ while the optimal dictionary-symbolwise compressor (under the constant cost assumption on encoding pointers) has a cost that is $O(k + \sqrt{k} \log k) = O(k)$.

Proof. (Sketch) An optimal constant cost LZ78 compressor must uses k phrases to code S. Then each phrase used to code the subword $2 \ldots k$ of T' has length at most 2 and therefore the number of phrases that it must use to code $2 \ldots k$ is at least $(k-1)/2 \geq \frac{1}{2}k/2$. Analogously, each phrase used to code the subword $3 \ldots k$ of T' has length at most 3 and therefore the number of phrases that it must use to code $3 \ldots k$ is at least $(k-2)/3 \geq \frac{1}{3}k/2$. We keep on going up to conclude that number of phrases that it must use to code $\sqrt{k} \ldots k$ is at least $(k - \sqrt{k} + 1)/\sqrt{k} \geq \frac{1}{\sqrt{(k)}}k/2$. Adding all these numbers we get that the total number of phrases is smaller than or equal to $O(k + \log \sqrt{k} \times k/2) = O(k \log k)$.

Let us now prove that an optimal dictionary-symbolwise compressor has a cost that is $O(k)$ by showing that there exists at least one parse that has cost $O(k)$.

The parse that we analyze parses S with the LZ78 dictionary and spend for this part of the string $O(k)$. Then it uses the LLM Predictor to compress the subword $2 \ldots k$ of T'. Firstly it outputs a symbol 'n' followed by the symbol 2 because it is unable to predict the symbol 2 and then it outputs $k - 2$ symbols 'y' that, in turn, are coded by the run length encoder with a cost that is $O(\log k)$. The whole cost of subword $2 \ldots k$ is then $O(\log k)$. Then the LLM Predictor compresses sequentially the subword $i \ldots k$ of T' , with $3 \leq i \leq \sqrt{k}$ and any time it spends at most $O(\log k)$. The total cost of this parse is then $O(k + \sqrt{k} \log k) = O(k)$. □

Chapter 3

History of the Parsing Problem

In this chapter we survey some of the milestone results about the parsing problem, starting for those concerning static dictionaries through the dynamic case. In the last section we present a small new contribution that complete the picture of fixed costs case. It is the generalization of the greedy parsing of Cohn for static dictionary to the dynamic dictionary case.

3.1 Static Dictionaries and Uniform Costs

The problem of optimal parsing in the case of static dictionaries and uniform costs is equal to find the minimal number of substrings that covers a given text. In '73, the Wagner's paper (see [38]) shows that a dynamic programming solution can be found in $O(n^2)$, where a text T of length $|T| = n$ is provided at ones.

in '74 Schuegraf (see [33]) showed that the $O(n^2)$ complexity is also achieved by a graph based approach to the problem (see 2.3 for the details of this approach).

In '77 and '78 years the foundational dynamic dictionary based compression methods LZ77 and LZ78 have been introduced. They both use an online linear time greedy approach to the parsing problem without proof of optimality in favour of the algorithm simplicity and execution speed.

Those compression methods use both an uniform cost model for the dictionary pointers. The online greedy approach is realized by choosing the longest dictionary phrase that matches with the forwarding text to extend the previous parsing, moving on the text left to right, until the whole text is covered.

In '82, LZSS compression method, based on the LZ77 one, was presented (see [36]). It improves the compression rate and the execution time without changing the original parsing approach. The main difference is that a symbol is used only when there is no match between dictionary and text. But when there is any match, the longest one is always chosen.

In '84, LZW variant of LZ78 was introduced by Welch (see [39]). This is, to our best knowledge, the first compression methods that use a dynamic dictionary and variable costs of pointers.

In '85, Hartman et al. (see [20]) showed that for the prefix-closed LZ-like dictionaries an offline optimal parsing solution has $O(n^{\frac{3}{2}})$ complexity.

In '96 the greedy parsing was ultimately proved by Cohn et al. (see [6]) to be optimal for fixed suffix-closed dictionary under the uniform cost model. They also proved that the right to left greedy parsing is optimal for prefix-closed dictionaries. Unfortunately the dynamic dictionary case, like the LZ-like one, was not fully solved.

In [36] the greedy parsing was showed to be optimal and linear for LZ77-like dictionaries under a constant cost for encoding dictionary pointers assumption. The optimality of greedy parsing for static suffix closed dictionaries and constant cost dictionary pointers was proved in [6] and it was later used also in [23].

3.1.1 Suffix-closed Dictionary Optimal Parsing

We want here extend the elegant proof of Cohn et al. (see [6]) to the case of dynamic dictionaries under the assumption that the dictionary in position $i+1$ contains all strict suffixes of all phrases of the dictionary in position i. In this way the proof of Cohn at al. hold also for LZ77-like algorithms for sliding windows or unbounded window. This is an original small contribution.

The classic Cohn's theorem states that if D is a static suffix-closed dictionary, then the greedy parsing is optimal on all the texts. The proof exploits the suffix property of the dictionary, that is that if a phrase w is in D, then all the suffixes of w are in the dictionary too, i.e. $\mathit{suff}(w) \subset D$, where $\mathit{suff}(w)$ is the set of all the suffixes of w.

Let us focus on the effect of the dictionary suffix-closed property on the Schuegraf graph $G_{A,T}$. Given a text T and an algorithm A which use the dictionary D, we have that if D is suffix-closed, then for any edge (i, j) of $G_{A,T}$ associated to the phrase $w \in D$, with $|w| = j - i$ and $w = T[i : j]$, all the edges (k, j), $i < k < j$ belong to $G_{A,T}$.

To generalize the suffix-close property of dictionaries to the dynamic dictionary framework, let us consider a dictionary phrase $w \in D_i$, with $|w| = n$. The set of all the suffixes of w is $\mathit{suff}(w) = \{w_j\}$ with $0 \leq j \leq n$ where for $j < n$ $w_j = w[j : n-1]$ is the jth suffix of w of length $n - j$ and $w_n = \epsilon$ is the empty word for $j = n$. If the dynamic dictionary D have the suffix-close property, then $\forall i, w$ if $w \in D_i$, then $w_j \in D_{i+j}$, for any suffix w_j of w. On the graph model, if D is a dynamic suffix-closed dictionary, for any i, j if the edge (i, j) is in $G_{A,T}$, then $(k, j) \in G_{A,T}$ for any $k \in [i..j)$.

Exploiting the dynamics of the dictionary D, previous statements may be simplified by highlighting the dependence by consecutive dictionaries in the following way.

Proposition 3.1.1. *If the dynamic dictionary D have the suffix-close property, then $\forall i, w$ if $w \in D_i$, then $w_1 \in D_{i+1}$. On the graph model, if the dynamic dictionary D have the suffix-close property, then $\forall i, j$ if the edge (i, j) is in $G_{A,T}$, then $(i + 1, j) \in G_{A,T}$.*

Notice that LZ77-like dictionaries satisfy the above property, either in the unbounded version and sliding window variants.

We want here to generalize this result of the classic Cohn's theorem to the dynamic dictionary case.

3.2 Flexible Parsing

In [28] Matias and Sahinalp gave a linear-time optimal parsing algorithm in the case of dictionary compression where the dictionary is prefix closed *and* the cost of encoding dictionary pointer is constant, i.e. all the codewords have equal length. In this thesis we eliminate the latter constraint and we further extend this result to the dictionary-symbolwise case. Matias and Sahinalp called their parsing algorithm *Flexible Parsing*. Hence, we called our parsing algorithm *Dictionary-Symbolwise Flexible Parsing*.

The basic idea of *one-step-lookahead parsing*, that is the basis of flexible parsing, was firstly used to our best knowledge in [20] in the case of dictionary compression where the dictionary is static and prefix closed and the cost of encoding dictionary pointer is constant. A first intuition, not fully exploited, that this idea could be successfully used in the case of dynamic dictionaries, was given in [21] and also in [23], where it was called maximum two-phrase-length (MTPL) parsing. It is also called a semi-greedy parsing.

3.3 The Optimal Parsing Problem

Optimal with respect to what? Obviously in data compression we are mainly interested to achieve the best compression ratio, that correspond to minimizing the size of the compressed data. This notion of optimality is sometimes called bit-optimality. But our question has a deeper sense. When can we say that a parsing strategy is an optimal parsing algorithm? When is it optimal with respect to the input data? When is it optimal with respect to the compression algorithms in which it is involved?

We have chosen this last option.

Therefore, given a dictionary based compression algorithm, we will define a compression scheme as the set of all the algorithms that use the same dictionary description and the same encodings. They differ for just the parsing method. We also will define an equivalence relation between algorithms where all the similar algorithms belong to the same equivalence class. The optimal parsing will be the parsing that minimize the compression ration within a compression scheme and, for extension, the compression algorithm that include the optimal parsing will be an optimal compression algorithm.

Chapter 4

Dictionary-Symbolwise Flexible Parsing

In this chapter we consider the case of dictionary-symbolwise algorithms where the parsing is a free mixture of dictionary phrases and symbols. We present the *dictionary-symbolwise flexible parsing* that is a dictionary-symbolwise optimal parsing algorithm for prefix-closed dictionaries and *variable* costs. This is a generalization of the Matias' and Sahinalp's *flexible parsing* algorithm (see [28]) to variable costs.

The algorithm is quite different from the original Flexible Parsing but it has some analogies with it. Indeed, in the case of LZ78-like dictionaries, it makes use of one of the main data structures used for the original flexible parsing in order to be implemented in linear time.

In next sections we will show some properties of the graph $G_{\mathcal{A},T}$ when the dictionary of $\mathcal{A}$ is prefix-closed and the encoding of the dictionary pointers leads to a *nondecreasing* cost function. We will call *c-supermaximal* some significant edges of $G_{\mathcal{A},T}$ and we will use the *c-supermaximal* edges to build the graph $G'_{\mathcal{A},T}$ that is a subgraph of $G_{\mathcal{A},T}$. Then, we will show that any minimal path form the origin of $G'_{\mathcal{A},T}$ is a minimal path in $G_{\mathcal{A},T}$. We will introduce the *dictionary-symblwise flexible parsing* algorithm that build the graph $G'_{\mathcal{A},T}$ and then find a minimal weight path on it in order to parse the text. We will prove that this parsing is optimal within any scheme having the Schuegraf Property. We will show that $G'_{\mathcal{A},T}$, and the *dictionary-symblwise* *flexible parsing* consequently, has linear space and time complexity w.r.t. the text size in the LZ78-like dictionary cases. In the case of LZ77-like dictionaries, $G'_{\mathcal{A},T}$ has linear space and $O(n \ log \ n)$ time complexities. In the last two sections we will report some implementation details.

4.1 The *c-supermaximal* Edges

We suppose that a text T of length n and a dictionary-symbolwise algorithm $\mathcal{A}$ are given. We assume here that the dictionary is prefix closed at any moment.

Concerning the costs of the dictionary pointer encodings, we recall that costs are variable, costs assume positive values and they must include the cost of flag information. Concerning the symbolwise encodings, the costs of symbols must be positive, including the flag information cost. They can vary depending on the position of the character in the text and on the symbol itself. Furthermore, we assume that the graph $G_{\mathcal{A},T}$ is well defined under our assumption.

We denote by d the function that represents the distance of the vertices of $G_{\mathcal{A},T}$ from the origin of the graph. Such a distance $d(i)$ is classically defined as the minimal cost of all possible weighted paths from the origin to the vertex i, where $d(0) = 0$. This distance obviously depends on the cost function. We say that cost function C is *prefix-nondecreasing* at any moment if for any $u, v \in D_p$ phrases associated with edges $(p, i), (p, q)$, with $p < i < q$ (that implies that u is prefix of v), one has that $C((p, i)) \leq C((p, q))$.

Lemma 4.1.1. *Let $\mathcal{A}$ be a dictionary-symbolwise algorithm such that for any text T the graph $G_{\mathcal{A},T}$ is well defined. If the dictionary is always prefix-closed and if the cost function is always prefix-nondecreasing then the function d is nondecreasing monotone.*

Proof. It is sufficient to prove that for any i, $0 \leq i < n$ one has that $d(i) \leq d(i + 1)$. Let $j \leq i$ be a vertex such that $(j, i + 1)$ is an edge of the graph and $d(i + 1) = d(j) + C((j, i + 1))$. If j is equal to i then $d(i + 1) = d(i) + C((i, i + 1))$ and the thesis follows. If j is smaller than i then, since the dictionary D_j is prefix closed, (j, i) is still an edge in D_j and

$d(i) \leq d(j)+C((j,i)) \leq d(j)+C((j,i+1)) = d(i+1)$ and the thesis follows. The last inequality in previous equation comes from the prefix-nondecreasing property of the cost function. $\square$

Let us call vertex j a *predecessor* of vertex $i \iff \exists (j,i) \in E$ such that $d(i) = d(j) + C((j,i))$. Let us define $pre(i)$ to be the smallest of the predecessors of vertex $i, 0 < i \leq n$, that is $pre(i) = \min\{j \mid d(i) = d(j) + C((j,i))\}$. In other words $pre(i)$ is the smallest vertex j that contributes to the definition of $d(i)$. Clearly $pre(i)$ has distance smaller than $d(i)$. We notice that a vertex can be a predecessor either via a dictionary edge or via a symbol edge. It is also possible to extend previous definition to pointers having a cost smaller than or equal to a fixed c as follows.

Definition 4.1. For any cost c we define $pre_c(i) = \min\{j \mid d(i) = d(j) + C((j,i))$ and $C((j,i)) \leq c\}$. If none of the predecessor j of i is such that $C((j,i)) \leq c$ then $pre_c(i)$ is undefined.

If all the costs of pointers are smaller than or equal to c then for any i one has that $pre_c(i)$ is equal to $pre(i)$.

Analogously to the notation of [27], we want to define two boolean operations *Weighted-Extend* and *Weighted-Exist*.

Definition 4.2. (Weighted-Extend) Given an edge (i,j) in $G_{\mathcal{A},T}$and a cost value c, the operation *Weighted-Extend*$((i,j),c)$ finds out whether the edge $(i,j+1)$ is in $G_{\mathcal{A},T}$having cost smaller than or equal to c.

More formally, let (i,j) in $G_{\mathcal{A},T}$be such that $w = T[i+1:j] \in D_i$. Operation *Weighted-Extend*$((i,j),c) =$ *"yes"* $\iff wa_{j+1} = T[i+1:j+1] \in D_i$ with $j < n$ such that $(i,j+1)$ is in $G_{\mathcal{A},T}$ *and* $C((i,j+1)) \leq c$, where C is the cost function associated with the algorithm $\mathcal{A}$. Otherwise *Weighted-Extend*$((i,j),c) =$ *"no"*.

Let us notice that *Weighted-Extend* always fails to extend any edge ending at node n.

Definition 4.3. (Weighted-Exist) Given $0 \leq i < j \leq n$ and a cost value c, the operation *Weighted-Exist*(i,j,c) finds out whether or not the phrase $w = T[i+1:j]$ is in D_i such that the corresponding edge (i,j) in $G_{\mathcal{A},T}$and the cost of (i,j) is smaller than or equal to c.

Let us notice that doing successfully the operation *Weighted-Extend* on $((i,j),c)$ means that $wa_{j+1} \in D_i$ is the weighted extension of w and the encoding of $(i, j+1)$ has cost less or equal to c. Similarly, doing a *Weighted-Exist* operation on (i,j,c) means that an edge (i,j) exists in $G_{\mathcal{A},T}$having cost less or equal to c.

Definition 4.4. (c-supermaximal) Let E_c be the subset of all edges of the graph having cost smaller than or equal to c. Let us define, for any cost c, the set $M_c \subseteq E_c$ be the set of *c-supermaximal* edges, where $(i,j) \in M_c \iff (i,j) \in E_c$ and $\forall p, q \in V$, with $p < i$ and $j < q$, the arcs $(p,j),(i,q)$ are not in E_c. For any $(i,j) \in M_c$ let us call i a c-starting point and j a c-ending point.

Proposition 4.1.2. *Suppose that* (i,j) *and* (i',j') *are in* M_c. *One has that* $i < i'$ *if and only if* $j < j'$.

Proof. Suppose that $i < i'$ and that $j \geq j'$. Since the dictionary D_i is prefix closed we have that (i,j') is still in D_i and therefore it is an edge of $G_{\mathcal{A},T}$. By the prefix-nondecreasing property of function C we have that $C((i,j')) \leq C((i,j)) = c$, i.e. $(i,j') \in E_c$. This contradicts the fact that (i',j') is in M_c and this proves that if $i < i'$ then $j < j'$. Conversely suppose that $j < j'$ and that $i \geq i'$. If $i > i'$ by previous part of the proof we must have that $j > j'$ that is a contradiction. Therefore $i = i'$. Hence (i,j) and (i,j') both belongs to M_c and they have both cost smaller than or equal to c. This contradicts the fact that (i,j) is in M_c and this proves that if $j < j'$ then $i < i'$. $\square$

By previous proposition, if $(i,j) \in M_c$ we can think j as a function of i and conversely. Therefore it is possible to represent M_c by using an array $M_c[\ ]$ such that if (i,j) is in M_c then $M_c[j] = i$ otherwise $M_c[j] = Nil$. Moreover the non-*Nil* values of this array are strictly increasing. The positions j having value different from *Nil* are the ending positions.

We want to describe a simple algorithm that outputs all *c-supermaximal* edges scanning the text left-to-right. We call it *Find Supermaximal*(c). It uses the operations *Weighted-Extend* and *Weighted-Exist*. The algorithm starts with $i = 0, j = 0$ and $w = \epsilon$, the empty word. The word w is indeed implicitly defined by the arc (i, j) when $i < j$ or it is the empty word when $i = j$. Therefore w will not appear explicitly in the algorithm. Since the values of i and j are only increased by one and i is always less or equal than j, the word w can be seen as a sliding window of variable size that scan the text left-to-right. w is moved along the text either by extensions or by contractions to its suffixes.

At each algorithm's step, j is firstly increased by one. This extends w concatenating it to $T[j]$. The algorithm executes then a series of *Weighted-Exist* increasing i by one, i.e. it contracts many times w. This series of *Weighted-Exist* ends when w is the empty word or an edge $(i, j) \in E_c$ is found such that (i, j) is not contained in any already found *c-supermaximal* edge (see 4.1.4). Indeed, since the increment on j at line 3, if such edge (i, j) exists then we have that $\forall\ (p, q) \in M_c$ with $p < i$, $(i, j) \in E_c$. Moreover, if such edge (i, j) exists, i is a *c-starting* point and a series of *Weighted-Extend* is executed looking for the corresponding *c-ending* point. After each *Weighted-Extend* positive answer, j is incremented by one. Once that *Weighted-Extend* outputs "no", i.e. once that (i, j) cannot be weighted-extended any more, (i, j) is a *c-supermaximal* and it is inserted into M_c to be outputted later. The algorithm's step ends when a *c-supermaximal* is found or when w is equal to the empty word. The algorithm runs as long as there are unseen characters, i.e. until j reaches n.

The algorithm is stated more formally in Table 4.1.

Proposition 4.1.3. *Given a cost value c, the* Find Supermaximal *algorithm correctly computes M_c.*

Proof. First of all let us prove that if $(\hat{\imath}, \hat{\jmath})$ is inserted by the algorithm in M_c then $(\hat{\imath}, \hat{\jmath})$ is *c-supermaximal.*

If $(\hat{\imath}, \hat{\jmath})$ is inserted into M_c at line 11, then an edge $(\hat{\imath}, j')$ at line 4 was previously proved to exist and to have cost $C((\hat{\imath}, j')) \leq c$. It caused the termination of the loop at lines $4 - 6$. For the line 7 we know that $\hat{\imath} < j'$ and by the loop $8 - 10$ we know that all the edges $(\hat{\imath}, q)$ with $j' \leq q \leq \hat{\jmath}$ exist and they all are such that $C((\hat{\imath}, q)) \leq c$. Therefore $(\hat{\imath}, \hat{\jmath})$ costs at most c and then the first part of the definition is verified. Since the *Weighted-Extend*$((\hat{\imath}, \hat{\jmath}), c)$ = "no" at line 8, that was the exit condition of that loop,

```
FIND SUPERMAXIMAL (c)
01.  i ← 0, j ← 0, M_c ← ∅
02.  WHILE j < n DO
03.      j ← j + 1
04.      WHILE i < j AND Weighted-Exist(i, j, c) = "no" DO
05.          i ← i + 1
06.      ENDWHILE
07.      IF i < j THEN
08.          WHILE Weighted-Extend ((i, j), c) = "yes" DO
09.              j ← j + 1
10.          ENDWHILE
11.          INSERT ((i, j), M_c)
12.      ENDIF
13.  ENDWHILE
14.  RETURN M_C
```

Table 4.1: The pseudocode of the Find Supermaximal algorithm. The function INSERT simply insert the edge (i, j) in the dynamical set M_c.

then $(\hat{\imath}, \hat{\jmath} + 1) \notin E_c$. Since D_i is prefix closed and the function cost C is prefix-nondecreasing $\forall q \in V$ with $\hat{\jmath} < q$ the arc $(\hat{\imath}, q)$ is not in E_c for otherwise $(\hat{\imath}, \hat{\jmath} + 1)$ would be in E_c.

It remains to prove that $\forall p \in V$ with $p < \hat{\imath}$ the arc $(p, \hat{\jmath})$ is not in E_c.

Suppose by contradiction that there exists one such arc $(p, \hat{\jmath})$ in E_c. Since the variables i, j never decrease along algorithm steps, the variable i reach the value p before that $(\hat{\imath}, \hat{\jmath})$ is inserted in M_c. Let j_p be the value of j when i reached the value p. Since the variable i is increased only inside the loop at lines $4 - 6$, we have that $p \leq j_p$. If $p = j_p$ the algorithm terminates the current step by the conditions at lines 4 and 7 and it enter the next step with $j = j_p + 1$ due to line 3. Therefore j will reaches the value $j_p + 1$ for $p = j_p$ otherwise j will be equal to j_p. In both cases, since $i < j$, the condition at line 7 is satisfied and the loop $8 - 10$ is reached. Since D_p is prefix closed and the function cost is prefix-nondecreasing then $\forall q$ such that $j \leq q < \hat{\jmath}$, *Weighted-Extend*$((p, q))$ = "yes". Then, the loop $8 - 10$ increases the j up to at least the value $\hat{j}$, i.e. the algorithm reaches the line 11 with $\hat{j} \leq j$. At this point, an edge (p, j) is inserted in M_c and the algorithm moves on the next step. Since the increment of the variable j at line 3, we have that in the rest of the algorithm only edges where j is greater than $\hat{j}$ may be considered and then $(\hat{\imath}, \hat{j})$ will not be inserted. That is a contradiction. Therefore, if $(\hat{\imath}, \hat{j})$ is inserted by the algorithm in M_c then $(\hat{\imath}, \hat{j})$ is *c-supermaximal*.

We have now to prove that if $(\hat{\imath}, \hat{j})$ is *c-supermaximal* then it is inserted by the algorithm in M_c.

Suppose that variable i never assumes the value $\hat{\imath}$. The algorithm ends when variable j is equal to n. Let i_n be the value of variable i when j becomes n, then we have that $i_n < \hat{\imath} < \hat{j} < n = j$. If the variable j reaches the value n inside the loop $8 - 10$ then the operation *Weighted-Extend*$((i_n, n-1), c)$ has outputted "yes" just before. At line 11 the edge (i_n, n) is inserted into M_c and then (i_n, n) is *c-supermaximal*. This contradict that $(\hat{\imath}, \hat{j})$ is *c-supermaximal*. Otherwise, if the variable j reaches the value n at line 3, then we have two cases. In the first one, *Weighted-Exist*(i_n, n, c) outputs "yes", i.e. the edge (i_n, n) is in E_c. Since $i = i_n < n = j$ line 7 condition is satisfied, *Weighted-Extend*$((i_n, n), c)$ outputs "no" by definition and then (i_n, n) is in M_c, i.e. it is a *c-supermaximal*. That is again a contradiction. In the second case, *Weighted-Exist*(i_n, n, c) outputs "no" one or multiple times while i grows up to a value $i'_n < \hat{\imath}$ by hypothesis. Using the same argumentation as before, (i'_n, n) in M_c leads to a contradiction.

Therefore at a certain moment variable i assumes the value $\hat{\imath}$. Let $j_{\hat{\imath}}$ be the value of variable j in that moment.

We suppose that $j_{\hat{\imath}} \leq \hat{j}$. Since the dictionary $D_{\hat{\imath}}$ is prefix closed and the cost function is prefix nondecreasing, *Weighted-Exist*$(\hat{\imath}, j_{\hat{\imath}}, c)$ outputs "yes" causing the exit from the loop at lines $4 - 6$. At this point, inside the loop $8 - 10$, the variable j reaches the value $\hat{j}$ since *Weighted-Extend*$((\hat{\imath}, j), c)$ outputs "yes" for any j less than $\hat{j}$, while *Weighted-Extend*$((\hat{\imath}, \hat{j}), c)$ outputs "no". Finally, $(\hat{\imath}, \hat{j})$ is inserted into M_c at line 11.

Suppose by contradiction that $j_{\hat{\imath}} > \hat{j}$ when i assumes the value $\hat{\imath}$ at line 5. This may happens only if the edge $(\hat{\imath} - 1, j_{\hat{\imath}})$ has been inserted in M_c in the previous step of the algorithm. Since $\hat{\imath} - 1 < \hat{\imath} < \hat{j} < j_{\hat{\imath}}$ this contradict the hypothesis that $(\hat{\imath}, \hat{\jmath})$ is *c-supermaximal*.

$\square$

Proposition 4.1.4. *For any edge $(i, j) \in E_c$ there exists a* c-supermaximal *edge $(\hat{\imath}, \hat{\jmath})$ containing it, i.e. such that $\hat{\imath} \leq i$ and $j \leq \hat{\jmath}$.*

Proof. We build $(\hat{\imath}, \hat{\jmath})$ in algorithmic fashion. The algorithm is described in what follows in an informal but rigorous way. If edge (i, j) is not *c-supermaximal* then we proceed with a round of *Weighted-Extend*$((i, j), c)$ analogously as described in algorithm Find Supermaximal and we increase j of one unit until *Weighted-Extend* outputs "no". Let j' be the value of j for which *Weighted-Extend* output "no". Clearly $(i, j') \in E_c$ and $(i, j' + 1)$ is not. If (i, j') is not *c-supermaximal* the only possibility is that there exists at least one $i' < i$ such that $(i', j') \in E_c$. At this point we keep iterating previous two steps starting from $(i - 1, j')$ instead of (i, j) and we stops whenever we get a *c-supermaximal* edge, that we call $(\hat{\imath}, \hat{\jmath})$. $\square$

By previous proposition for any node $v \in G_{\mathcal{A},T}$ if there exists a node $i < v$ such that $C((i, v)) = c$ and $d(v) = d(i) + c$ then there exists a *c-supermaximal* edge $(\hat{\imath}, \hat{\jmath})$ containing (i, v) and such that $\hat{\jmath}$ is the *closest* arrival point greater than v. Let us call this *c-supermaximal* edge $(\hat{\imath}_v, \hat{\jmath}_v)$. We use $\hat{\imath}_v$ in next proposition.

Proposition 4.1.5. *Suppose that $v \in G_{\mathcal{A},T}$ is such that there exists a previous node i such that $C((i, v)) = c$ and $d(v) = d(i) + c$. Then $\hat{\imath}_v$ is a predecessor of v, i.e. $d(v) = d(\hat{\imath}_v) + C((\hat{\imath}_v, v))$ and, moreover, $d(\hat{\imath}_v) = d(i)$ and $C((\hat{\imath}_v, v)) = c$.*

Proof. Since $(\hat{\imath}_v, \hat{\jmath}_v)$ contains (i, v) and the dictionary at position $\hat{\imath}_v$ is prefix closed then $(\hat{\imath}_v, v)$ is an edge of $G_{\mathcal{A},T}$. Since $(\hat{\imath}_v, \hat{\jmath}_v)$ has cost smaller than or equal to c then, by the suffix-nondecreasing property, also $(\hat{\imath}_v, v)$ has cost smaller than or equal to c. Since the distance d is nondecreasing we know that $d(\hat{\imath}_v) \leq d(i)$. By very definition of the distance d we know that $d(v) \leq d(\hat{\imath}_v) + C((\hat{\imath}_v, v))$.

Putting all together we have that $d(v) \leq d(\hat{\imath}_v) + C((\hat{\imath}_v, v)) \leq d(i) + c = d(v)$. Hence the inequalities in previous equation must be equalities and, further, $d(\hat{\imath}_v) = d(i)$ and $C((\hat{\imath}_v, v)) = c$. $\square$

Corollary 4.1.6. *For any vertex v, the edge $(\hat{i}_v, v)$ is the last edge of a path of minimal cost from the origin to vertex v.*

Proof. Any edge x in $G_{\mathcal{A},T}$ such that $d(v) = d(x) + C((x,v))$ is the last edge of a path of minimal cost from the origin to vertex v. $\square$

Remark 5. Let us notice that the variable i is increased only at line 05 along the FIND SUPERMAXIMAL algorithm.

4.2 The Subgraph $G'_{\mathcal{A},T}$

In what follows we describe a graph $G'_{\mathcal{A},T}$ that is a subgraph of $G_{\mathcal{A},T}$ and that is such that for any node $v \in G_{\mathcal{A},T}$ there exists a minimal path from the origin to v in $G'_{\mathcal{A},T}$ that is also a minimal path from the origin to v in $G_{\mathcal{A},T}$. The proof of this property, that will be stated in the subsequent proposition, is a consequence of Proposition 4.1.5 and Corollary 4.1.6.

We describe the building of $G'_{\mathcal{A},T}$ in an algorithmic way.

The set of nodes of $G'_{\mathcal{A},T}$ is the same of $G_{\mathcal{A},T}$. First of all we insert all the symbolwise edges of $G_{\mathcal{A},T}$ in $G'_{\mathcal{A},T}$. Let now $\mathcal{C}$ be the set of all possible costs that any dictionary edge has. This set can be build starting from $G_{\mathcal{A},T}$, but, in all known meaningful situations, the set $\mathcal{C}$ is usually well known and can be ordered and stored in an array in a time that is linear in the size of the text.

For any $c \in \mathcal{C}$ we use algorithm FIND SUPERMAXIMAL to obtain the set M_c. Then, for any $(i,j) \in M_c$, we insert in $G_{\mathcal{A},T}$ all the prefix of (i,j) except those which are contained in another *c-supermaximal* edge $(i', j') \in M_c$. In detail, for any *c-supermaximal* edge $(i,j) \in M_c$, let $(i', j') \in M_c$ be the previous *c-supermaximal* edge overlapping (i,j), i.e. $j' = \max_h\{(s,h) \in M_c | i < h < j\}$. Notice that this j' could not exist but, if it exists then by Proposition 4.1.2 there exists a unique i' such that $(i', j') \in M_c$. If (i', j') exists, then we add in $G'_{\mathcal{A},T}$ all the edges of the form (i,x), where $j' < x \leq j$, with label $L_{(i,x)} = c$. If (i', j') does not exist, then we add in $G'_{\mathcal{A},T}$ all the edges of the form (i,x), where $i < x \leq j$, with label $L_{(i,x)} = c$. In both cases, If such an edge (i,x) is already in $G'_{\mathcal{A},T}$, we just set the label $L_{(i,x)}$ to $\min\{L_{(i,x)}, c\}$. This concludes the construction of $G'_{\mathcal{A},T}$.

The algorithm Build $G'_{\mathcal{A},T}$ is formally stated in the Tabel 4.2.

```
BUILD G'_{A,T}
CREATE node 0
FOR v = 1 TO |T|
    CREATE node v
    CREATE symbolwise edge (v − 1, v)
    L(v − 1, v) ← C((v − 1, v))
ENDFOR
FOR ANY increasing c ∈ C
    M_c ← FIND SUPERMAXIMAL (c)
    j' ← 0
    FOR ANY (i, j) ∈ M_c left-to-right
        FOR ANY x | max{j', i} < x ≤ j
            IF (i, x) ∉ G'_{A,T} THEN
                CREATE edge (i, x)
                L(i, x) ← c
            ELSE
                L(i, x) ← min{L(i, x), c}
            ENDIF
        ENDFOR
        j' ← j
    ENDFOR
ENDFOR
```

Table 4.2: The pseudocode of the Build $G'_{\mathcal{A},T}$ algorithm.

Remark 6. Notice that for any cost c the above algorithm add in $G'_{\mathcal{A},T}$ at most a linear number of edges.

Let us notice that the graph $G'_{\mathcal{A},T}$ is a subgraph of $G_{\mathcal{A},T}$. Nodes and smbolwise edges are the same in both graphs by definition of $G'_{\mathcal{A},T}$. The edges (i,x) we add to $G'_{\mathcal{A},T}$, are the prefix of a *c-supermaximal* edge (i,j) of $G_{\mathcal{A},T}$. Since that the dictionary D_i is prefix closed, then all the edges (i,x) are also edges of $G_{\mathcal{A},T}$.

Proposition 4.2.1. *For any node $v \in G_{\mathcal{A},T}$, any minimal path from the origin to v in ${G'}_{\mathcal{A},T}$ is also a minimal path from the origin to v in $G_{\mathcal{A},T}$.*

Proof. The proof is by induction on v. If v is the origin there is nothing to prove. Suppose now that v is greater than the origin and let (i, v) the last edge of a minimal path in $G_{\mathcal{A},T}$from the origin to v. By inductive hypothesis there exists a minimal path $\mathcal{P}$ from the origin to i in ${G'}_{\mathcal{A},T}$ that is also a minimal path from the origin to i in $G_{\mathcal{A},T}$. Since (i, v) is a symbolwise arc then it is also in ${G'}_{\mathcal{A},T}$ and the concatenation of above minimal path $\mathcal{P}$ with (i, v) is a minimal path from the origin to v in ${G'}_{\mathcal{A},T}$ that is also a minimal path from the origin to v in $G_{\mathcal{A},T}$.

Suppose now that (i, v) is a dictionary arc and that its cost is c. Since it is the last edge of a minimal path we have that $d(v) = d(i) + c$. By Proposition 4.1.5 $d(v) = d(\hat{i}_v) + C((\hat{i}_v, v))$ and, moreover, $d(\hat{i}_v) = d(i)$ and $C((\hat{i}_v, v)) = c$. By Corollary 4.1.6, the edge $(\hat{i}_v, v)$ is the last edge of a path of minimal cost from the origin to vertex v. By inductive hypothesis there exists a minimal path $\mathcal{P}$ from the origin to $\hat{i}_v$ in ${G'}_{\mathcal{A},T}$ that is also a minimal path from the origin to i in $G_{\mathcal{A},T}$. Since $(\hat{i}_v, v)$ has been added by construction in ${G'}_{\mathcal{A},T}$, the concatenation of above minimal path $\mathcal{P}$ with $(\hat{i}_v, v)$ is a minimal path from the origin to v in ${G'}_{\mathcal{A},T}$ that is also a minimal path from the origin to v in $G_{\mathcal{A},T}$. □

Let us notice that it is possible to create the dictionary edges of ${G'}_{\mathcal{A},T}$ without an explicit representation in memory of all the M_c arrays. This is just an implementation detail that enhance speed and memory usage of the Build ${G'}_{\mathcal{A},T}$ algorithm in practice, without changing its order of complexity. The point is that we can insert the *c-supermaximal* edges and their prefix directly in the graph as soon as they are found along a *Find Supermaximal* execution. The correctness of this approach is a direct consequence of the following Remark 7.

Remark 7. Given a cost c, the edges (i, x) used by the Build ${G'}_{\mathcal{A},T}$ algorithm inside the block at lines $10 - 20$ are those for which the *Weighted-Extend* and the *Weighted-Exist* operations of the *Find Supermaximal*(c) algorithm report a positive answer.

4.3 The Dictionary-Symbolwise Flexible Parsing Algorithm

We can now finally describe the *Dictionary-symbolwise flexible parsing.*

The *Dictionary-symbolwise flexible parsing* firstly uses algorithm BUILD $G'_{\mathcal{A},T}$ and then uses the classical SINGLE SOURCE SHORTEST PATH (SSSP) algorithm (see [7, Ch. 24.2]) to recover a minimal path from the origin to the end of graph $G_{\mathcal{A},T}$. The correctness of the above algorithm is stated in the following theorem and it follows from the above description and from Prop. 4.2.1.

Theorem 4.3.1. Dictionary-symbolwise flexible parsing *is graph optimal.*

Notice that graphs $G_{\mathcal{A},T}$ and $G'_{\mathcal{A},T}$ are directed acyclic graphs (DAG) and their nodes from 1 to n, where 1 is the origin or the unique source of the graph and $n = |T|$ is the last node, are topologically ordered and linked by simbolwise edges. Recall that, given a node v in a weighted DAG, the classic solution to the SSSP is composed by two steps. The first one computes the distance and a predecessor of any node in the graph. It is accomplished by performing a visit on all the nodes in topological order and making a *relax* on any outgoing edge. Therefore, for any node v from 1 to n and for any edge (v, v') in the graph, the *relax* of (v, v') sets the distance and the predecessor of v' to v if $d(v) + C((v, v')) < d(v')$. The classic algorithm uses two arrays, $\pi[]$ and $p[]$, to store distance and predecessor of nodes.
The second step recovers the shortest path by following backward the predecessors chain from the last node to the origin of the graph and reverting it. From this simple analysis follows that if we know all the outgoing edges of any node in topological order then we can do directly the *relax* operation on them without having an explicit representation of the graph.

Let us suppose to have an *online* version of the BUILD $G'_{\mathcal{A},T}$ algorithm, where for any i from 1 to $|T|$, only edges (i, j) are created on the graph. We want now to merge the *online* BUILD $G'_{\mathcal{A},T}$ algorithm to the *relax* step of the SSSP algorithm. We maintain the two arrays $\pi[]$ and $p[]$ of linear size w.r.t. the text size, containing the distance and the predecessor of any node and we replace any edge creation or label updating with the relax operation.

About the *online* version of the BUILD $G'_{\mathcal{A},T}$, we can use the Remark 5 to make a kind of parallel run of the FIND SUPERMAXIMAL algorithm for any cost c, maintaining the variables i synchronized on the same value. Moreover, we use the Remark 7 to handle directly the edge creation as soon as they are found. We address all of these variations to the BUILD $G'_{\mathcal{A},T}$, the *Find Supermaximal* as well as the merge with the SSSP algorithm in order to obtain the DICTIONARY-SYMBOLWISE FLEXIBLE PARSING algorithm. The pseudo code of the DICTIONARY-SYMBOLWISE FLEXIBLE PARSING algorithm is reported in Table 4.3.

Let us notice that above algorithm uses only one dictionary at one time and never needs to use previous version of the dynamic dictionary. Recall that the dictionary is used by the *Weighted-Exist* and the *Weighted-Extend* operations. This is a direct consequence of the fact that any edge (i, j) refers to the dictionary D_i and that after edge (i, j) creation, only edge (p, q) with $p \geq i$ can be created.

Proposition 4.3.2. *Any practical implementations of the* Dictionary-symbolwise flexible parsing *does not require to explicitly represent the graph $G'_{\mathcal{A},T}$ regardless of its size. Since $G'_{\mathcal{A},T}$ nodes are visited in topological order by classic SSSP solutions, the algorithm needs to maintain just two linear size arrays, i.e. the array of node distances and the array of node predecessors, in order to correctly compute an optimal parsing.*

Let us summarize the Dictionary-Symbolwise Flexible Parsing algorithm requirements. Given a text T of size n the Dictionary-Symbolwise Flexible Parsing algorithm uses

- $O(n)$ space for the $\pi[]$ and $p[]$ arrays, regardless of the graph $G'_{\mathcal{A},T}$ size that is not really built, plus the dictionary structure.
- $O(|E|)$ time to analyze all the edges of the graph $G'_{\mathcal{A},T}$.
- it is not online because the backward recover of the parsing from the $p[]$ array.

With respect to the original Flexible Parsing algorithm we gain the fact that it can work with variable costs of pointers and that it is extended to

```
DICTIONARY-SYMBOLWISE FLEXIBLE PARSING
FOR i FROM 0 TO |T| − 1
    RELAX (i, i + 1, C((i, i + 1)))
    FOR ANY c ∈ 𝒞
        IF i = j_c THEN
            j_c ← 1 + j_c
        ENDIF
        IF j_c ≤ |T| AND Weighted-Exist (i, j_c, c) = "yes" THEN
            RELAX (i, j_c, C((i, j_c)))
            WHILE Weighted-Extend ((i, j), c) = "yes" DO
                j_c ← 1 + j_c
                RELAX (i, j_c, C((i, j_c)))
            ENDWHILE
            j_c ← 1 + j_c
        ENDIF
    ENDFOR
ENDFOR
RETURN REVERSE (v)

RELAX (u, v, c)
IF π[u] + c < π[v] THEN
    π[v] ← π[u] + c
    p[v] ← u
ENDIF

REVERSE (v)
IF v > 0 THEN
    REVERSE (p[v])
ENDIF
RETURN v
```

Table 4.3: The pseudocode of DICTIONARY-SYMBOLWISE FLEXIBLE PARSING algorithm, the RELAX and the REVERSE procedures. The distance array $\pi[]$ and the predecessor array $p[]$ are initialized to 0. Notice that the algorithm uses a different j_c variable for any c value.

the dictionary-symbolwise case. This cover for instance the LZW-like and the LZ77-like cases. But we lose the fact that the original one was "on-line". A minimal path has to be recovered, starting from the end of the graph backward. But this is an intrinsic problem that cannot be eliminated. Even if the dictionary edges have just one possible cost, in the dictionary-symbolwise case it is possible that any minimal path for a text T is totally different from any minimal path for the text Ta, that is the previous text T concatenated to the symbol a. The same can happen when we have a (pure) dictionary case with variable costs of dictionary pointers. In both cases, for this reason, there cannot exists any "on-line" optimal parsing algorithms, and, indeed, the original flexible parsing fails being optimal in the dictionary case when costs are variable.

On the other hand our algorithm is suitable when the text is divided in several contiguous blocks and, therefore, in practice there is not the need to process the whole text but it suffices to end the current block in order to have the optimal parsing (relative to that block).

4.4 Time and Space Analyses

In this section we analyze the *Dictionary-symbolwise flexible parsing* in both LZ78 and LZ77-like algorithm versions.

LZ78 Case

Concerning LZ78-like algorithms, the dictionary is prefix closed and it is implemented by using the LZW variant. We do not enter into the details of this technique. We just recall that the cost of pointers increases by one unit whenever the dictionary size is "close" to a power of 2. The moment when the cost of pointers increases is clear to both encoder and decoder. In our dictionary-symbolwise setting, we suppose that the flag information has constant cost. We assume therefore that it takes $O(1)$ time to determine the cost of a dictionary edge.

The maximal cost that a pointer can assume is smaller than $\log_2(n)$ where n is the text size. Therefore the set $\mathcal{C}$ of all possible costs of dictionary edges has logarithmic size and it is cheap to calculate.

In [27] the operations *Extend* and *Contract* are presented. It is also presented a linear size data structure called trie-reverse-trie-pair that allows to execute both those operations in $O(1)$ time. The operation $\mathit{Extend}(w, a)$ says whether the phrase wa is in the currently used dictionary. The operation $\mathit{Contract}(w)$ says whether the phrase $w[2 : |w|]$ is in the current dictionary.

Since at any position we can calculate in $O(1)$ time the cost of an edge, we can use the same data structure to perform our operations of *Weighted-Extend* and of *Weighted-Exist* in constant time as follow. In order to perform a $\mathit{Weighted\text{-}Extend}((i, j), c)$ we simply execute the operation $\mathit{Extend}(w, a_{j+1})$ with $w = T[i + 1 : j]$, i.e. the phrase associated to the edge (i, j), and then, if the answer is "yes", we perform a further check in $O(1)$ time on the cost of the found edge $(i, j + 1)$. Therefore, $\mathit{Weighted\text{-}Extend}((i, j), c)$ is equal to $\mathit{Extend}(T[i+1 : j], a_{j+1})$ AND $C((i, j+1)) \leq c$.

In order to perform a $\mathit{Weighted\text{-}Exist}((i, j), c)$ we simply use the *contract* on the phrase $a_i w$, where $w = T[i + 1 : j]$, and, if the answer is "yes" we perform a further check in $O(1)$ time on the cost of the found edge (i, j). Therefore, $\mathit{Weighted\text{-}Exist}(i, j, c)$ is equal to $\mathit{Contract}(a_i T[i + 1 : j])$ AND $C((i, j)) \leq c$.

At a first look, the algorithm Build $G'_{\mathcal{A},T}$ would take $O(n \log n)$ time. But, since there is only one active cost at any position in any LZW-like algorithms, then if $c < c'$ then $M_c \subseteq M_{c'}$, as stated in the following proposition.

Definition 4.5. We say that a cost function C is *LZW-like* if for any i the cost of all dictionary pointers in D_i is a constant c_i and that for any i, $0 \leq i < n$ one has that $c_i \leq c_{i+1}$.

Proposition 4.4.1. *If the cost function C is* LZW-like*, one has that if $c < c'$ then $M_c \subseteq M_{c'}$.*

Proof. We have to prove that for any $(i, j) \in M_c$ then $(i, j) \in M_{c'}$. Clearly if $(i, j) \in M_c$ then its cost is smaller than or equal to $c < c'$. It remains to prove that (i, j) is c'-supermaximal, e.g. that $\forall p, q \in V$, with $p < i$ and $j < q$, the arcs $(p, j), (i, q)$ are not in $E_{c'}$. Since $(i, j) \in M_c$ and since the cost of (i, j) is by hypothesis equal to c_i, we have that $c_i \leq c$. If arc (p, j) is in $E_{c'}$ then its cost is $c_p \leq c_i \leq c$ and therefore it is also in E_c contradicting the

c-supermaximality of (i, j). If arc (i, q) is in $E_{c'}$ then its cost is $c_i \leq c$ and therefore it is also in E_c contradicting the c-supermaximality of (i, j). $\square$

At this point, in order to build $G'_{\mathcal{A},T}$ we proceed in an incremental way. We build M_c for the smallest cost. Then, we start from the last built M_c to build $M_{c'}$, where c' is the smallest cost grater than c. And so on until all the costs are examined. We insert any edge (i, j) only in the set M_c where c is the real cost of the (i, j) edge. In this way, we avoid to insert the same edge (i, j) in more than one M_c since that the algorithm will insert eventually the edge (i, j) from the set M_c with the minimal cost $c = C((i, j))$.

A direct consequence of above approach, we have that only a linear size of edge are inserted in the graph $G'_{\mathcal{A},T}$.

The overall time for building $G'_{\mathcal{A},T}$ is therefore linear, as well as its size. The SINGLE SOURCE SHORTEST PATH over $G'_{\mathcal{A},T}$, that is a DAG topologically ordered, takes linear time (see [7, Ch. 24.2]).

In conclusion we state the following proposition.

Proposition 4.4.2. *Suppose that we have a dictionary-symbolwise scheme, where the dictionary is LZ78-like and the cost function is LZW-like. The symbolwise compressor is supposed to be, as usual, linear time. Using the trie-reverse-trie-pair data structure,* Dictionary-Symbolwise flexible parsing *is linear.*

LZ77 Case

Concerning LZ77, since the dictionary is prefix closed, we have that the Dictionary-Symbolwise Flexible Parsing is an optimal parsing. We exploit the discreteness of the cost function C when it is associated to the length of the codewords of a variable length code, like Elias codes or Huffman codes, to bound the cardinality of the set $\mathcal{C}$ to $O(log\ n)$. Indeed let us call $\hat{c}$ the maximum cost of any dictionary pointer, e.g. the longest and the most far one under the length-distance paradigm. Even if the cost actually depends on the text T, it usually has an upper bound that depends on the encoding and on the dictionary constrains and we can assume it to be $\hat{c} = O(log\ n)$, with $|T| = n$.

Operations *Weighted-Exist* and *Weighted-Extend* can be implemented in linear space and constant time by using classical suffix tree or other solutions when the dictionary is a LZ77-like one. For instance, in [10] it is shown how to compute the Longest Previous Factor (LPF) array in liner time. Recall that $T[i, LPF[i]]$ is the longest factor already seen in the text at some position $i' < i$. It is easy to see that following relations hold. The operation *Weighted-Exist* (i, j, c) outputs "yes" $\iff j \leq i + LPF[i]$ AND $C((i, j)) \leq c$ and the operation *Weighted-Extend* $((i, j), c)$ outputs "yes" $\iff j < i + LPF[i]$ AND $C((i, j + 1)) \leq c$. We recall that we are also assuming that it is possible to compute the cost of a given edge in constant time. Therefore, we use linear time and space to build the LPF array and then any operation *Weighted-Exist* or *Weighted-Extend* take just constant time.

Suppose to have a dictionary-symbolwise scheme, where the dictionary is LZ77-like and the dictionary pointer encoding, the symbolwise encoding and the flag information encoding are any variable-length encoding one. The use of the codeword length as cost function leads to a function that assumes integer values. Given $\hat{c}$ the maximum cost of any dictionary pointer with $\hat{c} \leq log(n)$, the Dictionary-Symbolwise Flexible Parsing runs in $O(n\ log\ n)$ time and space.

Let us notice that in most of the common LZ77 dictionary implementation, as it is in the Deflate compression tool, our assumption about the computation of edge cost in $O(1)$ time is not trivial to obtain.

Obviously, we are interested, for compression purpose, to the smallest cost between all the possible encoding of a phrase. For instance, the use of the length-distance pair as dictionary pointer leads to multiple representation of the same (dictionary) phrase since this phrase can occur more then once in the (already seen) text.

Since the closest occurrence uses the smallest distance to be represented, the cost of encoding the phrase using this distance is usually the smallest one, accordingly to the used encoding method.

A practical approach that looks for the above smallest distance makes use of hash tables, built on fixed length phrases.

A new data structure able to answer to the edge cost query in constant time and able to support the *Weighted-Exist* and the *Weighted-Extend* operation in the case of LZ-77 dictionaries, will be introduced in next chapter.

Chapter 5

The Multilayer Suffix Tree

We introduce here an online full-text index data structure that is able to find the rightmost occurrence of any factor or an occurrence which bit representation has equal length (Query 1). It has linear space complexity and it is built in $O(n\ log\ n)$ amortized time, where n is the size of the text. It is able to answer to the Query 1, given a pattern w, in $O(|pattern|\ loglog\ n)$. Furthermore, we will show how to use this structure to support the Weighted-Exist and the Weighted-Extend operations used by the *Dictionary-Symbolwise Flexible Parsing* algorithm in $O(1)$ time.

5.1 Background and Definitions

Let $Pos(w) \subset \mathbb{N}$ the set of all the occurrences of $w \in Fact(T)$ in the text $T \in \Sigma^*$, where $Fact(T)$ is the set of the factors of T. Let $Offset(w) \subset \mathbb{N}$ the set of all the occurrence offsets of $w \in Fact(T)$ in the text T, i.e. $x \in Offset(w)$ *iff* x is the distance between the position of an occurrence of w and the end of the text T. For instance, given the text $T = babcabbababb$ of length $|T| = 12$ and the factor $w = abb$ of length $|w| = 3$, the set of positions of w over T is $Pos(w) = \{4, 9\}$. The set of the offsets of w over T is $Offset(w) = \{7, 2\}$. Notice that $x \in Offset(w)$ *iff* exists $y \in Pos(w)$ such that $x = |T| - y - 1$. Since the offsets are function of occurrence positions, there is a bijection between $Pos(w)$ and $Offset(w)$, for any factor w.

Given a number encoding method, let $Bitlen : \mathbb{N} \to \mathbb{N}$ a function that to a number x associates the length in bit of the encoding of x. Let us consider the equivalence relation *having equal code bit-length* on the set *Offset*(w). The numbers $x, y \in$ *Offset*(w) are bit-length equivalent *iff* $Bitlen(x) = Bitlen(y)$. Let us notice that the *having equal code bit-length* relation induces a partition on *Offset*(w).

Definition 5.1. The *rightmost* occurrence of w over T is the offset of the occurrence of w that appears closest to the end of the text, if w appears at least once over T, otherwise it is not defined.

Notice that for any factor $w \in Fact(T)$, the rightmost offset of w is defined as follows.

$$rightmost(w) = \begin{cases} \min\{x \mid x \in \mathit{Offset}(w)\} & \text{if } \mathit{Offset}(w) \neq \emptyset \\ \text{not defined} & \text{if } \mathit{Offset}(w) = \emptyset \end{cases}$$

Let us notice that referring to the rightmost occurrence of a factor in an online algorithmic fashion, where the input text is processed left to right, corresponds to referring to the rightmost occurrence over the text *already processed*. Indeed, if at a certain algorithm step we have processed the first i symbols of the text, the rightmost occurrence of w is the occurrence of w closest to the position i of the text reading left to right.

Definition 5.2. Let $rightmost_i(w)$ be the rightmost occurrence of w over T_i, where T_i is the prefix of the text T ending at the position i in T. Obviously, $rightmost_n(w) = rightmost(w)$ for $|T| = n$.

In many practical algorithms, like in the data compression field, the text we are able to refer to is just a portion of the whole text. Let $T[j : i]$ be the factor of the text T starting from the position j and ending to the position i. We generalize the definition of $rightmost(w)$ over a factor $T[j : i]$ of T as follows.

Definition 5.3. Let $rightmost_{j,i}(w)$ be the rightmost occurrence of w over $T[j : i]$, where $T[j : i]$ is the factor of the text T starting at the position j and ending at the position i of length $i-j+1$. Obviously, $rightmost_{1,n}(w) = rightmost(w)$ for $|T| = n$.

The online full-text index we are going to introduce is able to answer to the *rightmost equivalent length* query in constant time, also referred hereby as *Query 1.* The Query 1 is more formally stated as below.

Definition 5.4. (Query 1) Given a text $T \in \Sigma^*$, a pattern $w \in \Sigma^*$ and a point i in time at which the prefix of the text T_i has been processed, the query *rightmost equivalent length* provides an occurrence offset $x \in [rightmost_i(w)]$ of w, where $[rightmost_i(w)]$ is the equivalence class induced by the relation *having equal code bit-length* containing $rightmost_i(w)$, i.e. the *rightmost* occurrence of w in T_i.

5.2 The Idea

There are many classic full text index data structures able to represent the set $Fact(T[1:i])$, like the suffix tree, the suffix array, the suffix automata and others. Many of them can easily be preprocessed in the *offline* fashion to make they able to find the rightmost occurrence of any factor over the whole text efficiently, but none of them can answer to the above Query 1.

The main idea of this new data structure, is based on a twofold observation. The fist observation is that the equivalence relation *having equal code bit-length* that induces a partition on *Offset*(w), for any w, also induces a partition on the set of all the possible offsets over a text T independently of a specific factor, i.e. on the set $[1..|T|]$. The second one is that for the encoding methods for which the $Bitlen$ function is a monotonic function, any equivalence class in $[1..|T|]$ is composed by contiguous points in $[1..|T|]$. Indeed, given a point $p \in [1..|T|]$, the equivalence class $[p]$ is equal to the set $[j..i]$, with $j \leq p \leq i$, $j = min\{x \in [p]\}$ and $i = max\{x \in [p]\}$.

Putting these observations all together suggests that the Query 1 can be addressed by a set of classic full text indexes, each of whom is devoted to some classes of the equivalence relation *having equal code bit-length.* We assume that we know the set of all the maximum element of any equivalence class and we call it $M = \{m_1, m_2, ..., m_s\}$, with $m_1 < m_2 < ... < m_s$.

Suppose to have at time i, one suffix tree SW_α for each $\alpha \in M$, to represent all the factor of $T[i - \alpha : i]$, i.e. one suffix tree for the sliding window of length α. Obviously, if a phrase w is in SW_α then w is in SW_β where $\alpha \leq \beta$ and $\alpha, \beta \in M$.

Proposition 5.2.1. *If a pattern w is in SW_α and is not in SW_β, where α is the maximum of the values $m_x \in M$ smaller than β, the code bit length of the rightmost occurrence of w in T_i, is equal to the code bit length of any occurrence of w in SW_α.*

Using above proposition, we are able to answer to the Query 1 once we find the smallest suffix tree containing the rightmost occurrence of the pattern. What follows is the trivial search of the suffix tree with the smallest sliding window, that contains an occurrence of the given pattern.

Given a pattern w, we look for w in SW_{m_1}. If w is in SW_{m_1}, then all the occurrences of w in $T[i - m_1 : i]$ belong to the class of the rightmost occurrence of w over T. If w is not in SW_{m_1}, then we look for any occurrence of w is in SW_{m_2}. If w is in SW_{m_2}, since it is not in SW_{m_1}, any occurrence of w in SW_{m_2} belong to the rightmost occurrence of w over T. Continuing in this way, once we found an occurrence of w in SW_{m_x}, this occurrence correctly answer to the Query 1.

In the following proposition we exploit the discreteness of the bit length function of common variable-length codewords. We consider to have a set of text indexes for sliding window, where the sliding window sizes are one for any value of the bit length function.

Proposition 5.2.2. *Given a variable-length code for the offsets and a pattern w, if the codewords bit length function is a monotone function, then all the occurrence offsets of w in the text index for the smallest sliding window where w appears at least once, belong to the [rightmost] class.*

Using the online suffix tree for sliding window data structure, introduced by Larson in [26] and later refined by Senft in [34], to represent SW_{m_x}, we are able to find an occurrence of a given pattern in time proportional to the pattern, the following proposition holds.

Proposition 5.2.3. *The simple data structure showed below, is able to answer to the Query 1 in time proportional to the pattern size times the cardinality of the set M.*

Let us now focusing on the set M. Since that many of the classic variable length codes for integers, like the Elias's γ-codes, produce codewords of length proportional to the logarithm of the represented value, we can assume that the cardinality of M is $O(log\ |T|)$. Since that $|T_i| = i$, in the *online* fashion, we have that latter proposition becomes as follows.

Proposition 5.2.4. *Using any classic variable length code method, the above data structure is able to answer to the Query 1 in* $O(|pattern|\ log\ i)$ *time.*

A similar result is due to Amir et al. (see [1]) that use $O(n)$ time, but it does not support *Weighted-Exist* and *Weighted-Extend* operations in constant time.

Based on the idea that using a binary search on the indexes SW_x, the query time can be reduced to $O(|pattern|\ loglog\ i)$, in the next section we introduce an *online* data structure able to answer to the Query 1 in $O(|pattern|\ loglog\ i)$ time and support *Weighted-Exist* and *Weighted-Extend* operations in constant time. This data structure uses $O(n)$ space and $O(n\ log\ n)$ amortized building time.

5.3 The Data Structure

Let us now introduce the Multilayer Suffix Tree data structure. It is based on the data structure presented above, where the SW_{m_x} suffix trees are organized in layers and their nodes are equipped with extra links and some extra nodes.

For each SW_{m_x} we use a classic suffix tree for sliding windows (see for instance the suffix tree base data structure presented in [18, 26, 34]). It allows to know one occurrence of a phrase w in constant time, but it cannot be directly used to find the rightmost occurrence. We think that it is possible to adapt our data structure to work with other index for sliding window (see for instance [22, 29, 35]).

Let call SW_{max} the first layer, where max is the maximum allowed offset. Each suffix tree SW_{m_x} lies on a layer and layers are ordered from the larger to the smallest sliding window. From now on we will refer to suffix trees or layers indifferently. Let think to the first layer as the top layer and the smallest one as the deeper layer.

Since that edges in suffix trees may have labels longer of one character, the ending point of a pattern may be either a node or a point in middle of an edge. A classic notation to refer to a general point in suffix tree is a threefold vector composed by a *locus*, a symbol and an offset. The locus is the node closest to the point on the path from the root, the symbol is the discriminant between the edges outgoing from the locus, and the offset tell how many characters of the edge are before the referred point.

More formally, given a pattern $w \in Fact(T)$, let S_T be the suffix tree for the text T. Let be $w = uv$, with $u, v \in Fact(T)$, where u is the longest prefix of w ending at a node p in S_T and v is the prefix of the edge where the pattern w ends out. Let call p the *locus* of w, $|v|$ the offset and $v[1]$ the discriminant character representing the point of w in S_T. If w ends to a node in S_T, then $w = u\epsilon$, this node is the locus, the length is equal to 0 and the discriminant character is any $c \in \Sigma$.

As direct consequence of the increasing size of the sliding windows, we have that if a pattern exists in a layer, it is in all the layers above it. Generally, if a pattern ends up to an internal node in some layer, there are internal nodes corresponding to the same pattern in all the above layers.

We address Query 1 using the binary search to find the deeper layer containing the given pattern.

Chapter 6

Conclusion

In this thesis we present some advancement on dictionary-symbolwise theory. We describe the *Dictionary-Symbolwise Flexible Parsing*, a parsing algorithm that extends the *Flexible Parsing* (see [28]) to *variable* costs and to the dictionary-symbolwise domain. We prove its optimality for prefix-closed dynamic dictionaries under some reasonable assumption. *Dictionary-Symbolwise Flexible Parsing* is *linear* for LZ78-like dictionaries and even if it is not able to run online it allows to easily make a block programming implementation. In the case of LZ77-like dictionary, we have obtained the $O(n \log n)$ complexity as authors of [17] recently did by using a completely different subgraph.

Last but not least, our algorithm allows to couple classical LZ-like compressors with several symbolwise methods to obtain dictionary-symbolwise algorithms with a proof of parsing optimality.

We have also proved in Section 2.5 that dictionary-symbolwise compressors can be asymptotically better than optimal pure dictionary compression algorithms in compression ratio terms.

We conclude this thesis with two open problems.

1. Theoretically, LZ78 is better on memoryless sources than LZ77. Experimental results say that when optimal parsing is in use it happens the opposite. Prove this fact both in pure dictionary case and in dictionary-symbolwise case.

2. Common symbolwise compressors are based on the arithmetic coding

approach. When these compressors are used, the costs in the graph are almost surely noninteger and, moreover, the graph is usually not well defined. The standard workaround is to use an approximation strategy. A big goal should be finding an optimal solution for these important cases.

Bibliography

[1] Amihood Amir, Gad M. Landau, and Esko Ukkonen. Online time stamped text indexing.

[2] R. Arratia and M. Waterman. The Erd"os-Rényi strong law for pattern matching with given proportion of mismatches. *Annals of Probability*, 4:200–225, 1989.

[3] Timothy C. Bell, John G. Cleary, and Ian H. Witten. *Text compression.* Prentice Hall, 1990.

[4] Timothy C. Bell and Ian H. Witten. The relationship between greedy parsing and symbolwise text compression. *J. ACM*, 41(4):708–724, 1994.

[5] Maxime Chochemore, Laura Giambruno, Alessio Langiu, Filippo Mignosi, and Antonio Restivo. Dictionary-symbolwise flexible parsing.

[6] Martin Cohn and Roger Khazan. Parsing with prefix and suffix dictionaries. In *Data Compression Conference*, pages 180–189, 1996.

[7] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, Cambridge, MA, second edition, 2001.

[8] Maxime Crochemore, Chiara Epifanio, Alessandra Gabriele, and Filippo Mignosi. On the suffix automaton with mismatches. In Jan Holub and Jan Zdárek, editors, *CIAA*, volume 4783 of *Lecture Notes in Computer Science*, pages 144–156. Springer, 2007.

[9] Maxime Crochemore, Laura Giambruno, Alessio Langiu, Filippo Mignosi, and Antonio Restivo. Dictionary-symbolwise flexible parsing. In Costas S. Iliopoulos and William F. Smyth, editors, *IWOCA*, volume 6460 of *Lecture Notes in Computer Science*, pages 390–403. Springer, 2010.

[10] Maxime Crochemore and Lucian Ilie. Computing longest previous factor in linear time and applications. *Inf. Process. Lett.*, 106(2):75–80, 2008.

[11] Maxime Crochemore and Thierry Lecroq. Pattern-matching and text-compression algorithms. *ACM Comput. Surv.*, 28(1):39–41, 1996.

[12] Giuseppe Della Penna, Alessio Langiu, Filippo Mignosi, and Andrea Ulisse. Optimal parsing in dictionary-symbolwise data compression schemes. `http://math.unipa.it/~alangiu/OptimalParsing.pdf`, 2006, unpublished manuscript.

[13] C. Epifanio, A. Gabriele, and F. Mignosi. Languages with mismatches and an application to approximate indexing. In *Proceedings of the 9th International Conference Developments in Language Theory (DLT05)*, LNCS 3572, pages 224–235, 2005.

[14] C. Epifanio, A. Gabriele, F. Mignosi, A. Restivo, and M. Sciortino. Languages with mismatches. *Theor. Comput. Sci.*, 385(1-3):152–166, 2007.

[15] P. Fenwick. Symbol ranking text compression with shannon recodings. *Journal of Universal Computer Science*, 3(2):70–85, 1997.

[16] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.

[17] Paolo Ferragina, Igor Nitto, and Rossano Venturini. On the bit-complexity of lempel-ziv compression. In *SODA '09: Proceedings of the Nineteenth Annual ACM -SIAM Symposium on Discrete Algorithms*, pages 768–777, Philadelphia, PA, USA, 2009. Society for Industrial and Applied Mathematics.

[18] E. R. Fiala and D. H. Greene. Data compression with finite windows. *Commun. ACM*, 32:490–505, April 1989.

[19] A. Gabriele, F. Mignosi, A. Restivo, and M. Sciortino. Indexing structure for approximate string matching. In *Proc. of CIAC'03*, volume 2653 of *LNCS*, pages 140–151, 2003.

[20] Alan Hartman and Michael Rodeh. *Optimal parsing of strings*, pages 155–167. Springer - Verlag, 1985.

[21] R. Nigel Horspool. The effect of non-greedy parsing in ziv-lempel compression methods. In *Data Compression Conference*, pages 302–311, 1995.

[22] Shunsuke Inenaga, Ayumi Shinohara, Masayuki Takeda, and Setsuo Arikawa. Compact directed acyclic word graphs for a sliding window. In *SPIRE*, pages 310–324, 2002.

[23] Tae Young Kim and Taejeong Kim. On-line optimal parsing in dictionary-based coding adaptive. *Electronic Letters*, 34(11):1071–1072, 1998.

[24] S. Rao Kosaraju and Giovanni Manzini. Compression of low entropy strings with lempel–ziv algorithms. *SIAM J. Comput.*, 29(3):893–911, 2000.

[25] Alessio Langiu. Optimal parsing in dictionary-symbolwise compression algorithms. Master's thesis, University of Palermo, 2008. `http://math.unipa.it/~alangiu/Tesi_Alessio_Langiu_MAc.pdf`.

[26] N. Jesper Larsson. Extended application of suffix trees to data compression. In *Data Compression Conference*, pages 190–199, 1996.

[27] Yossi Matias, Nasir Rajpoot, and S"uleyman Cenk Sahinalp. The effect of flexible parsing for dynamic dictionary-based data compression. *ACM Journal of Experimental Algorithms*, 6:10, 2001.

[28] Yossi Matias and S"uleyman Cenk Sahinalp. On the optimality of parsing in dynamic dictionary based data compression. In *SODA*, pages 943–944, 1999.

[29] Joong Chae Na, Alberto Apostolico, Costas S. Iliopoulos, and Kunsoo Park. Truncated suffix trees and their application to data compression. *Theor. Comput. Sci.*, 304:87–101, July 2003.

[30] David Salomon. *Data compression - The Complete Reference, 4th Edition.* Springer, 2007.

[31] David Salomon. *Variable-length Codes for Data Compression.* Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 2007.

[32] Khalid Sayood. *Introduction to Data Compression.* Morgan Kaufmann, 1996.

[33] Ernst J. Schuegraf and H. S. Heaps. A comparison of algorithms for data base compression by use of fragments as language elements. *Information Storage and Retrieval*, 10(9-10):309–319, 1974.

[34] M. Senft. Suffix tree for a sliding window: An overview. In *Proceedings of WDS'05, Part 1*, pages 41–46, 2005.

[35] Martin Senft and Tomáš Dvořák. Sliding cdawg perfection. In *Proceedings of the 15th International Symposium on String Processing and Information Retrieval*, SPIRE '08, pages 109–120, Berlin, Heidelberg, 2009. Springer-Verlag.

[36] James A. Storer and Thomas G. Szymanski. Data compression via textural substitution. *J. ACM*, 29(4):928–951, 1982.

[37] W. Szpankowski. *Average Case Analysis of Algorithms on Sequences.* Wiley, 2001.

[38] Robert A. Wagner. Common phrases and minimum-space text storage. *Commun. ACM*, 16(3):148–152, 1973.

[39] T. A. Welch. A technique for high-performance data compression. *IEEE Computer*, january:8–19, 1984.

Appendix A

Experiments

We now discuss about some experiments. Readers must keep into account that the results of this paper are mainly theoretical and that they apply to a very large class of compression algorithms. Due to this, the use of different methods of encoding for dictionary pointers as well as for symbolwise encoding and for the flag information encoding together with the dictionary constrains leads to different performances. Performances about time and space are strongly dependent on the programming language in use and on the programmers abilities. Therefore we decided to focus only on compression ratio.

We here discuss two particular cases that allow to compare our results with some well know commercial compressors. The first one is related to LZ78-like dictionary and Huffman codes. The second one concerns LZ77-like dictionaries with several window sizes and Huffman codes. We compare the obtained compression ratio with the gzip, zip and cabarc compression tools. The encoding method in use is a semi static Huffman codes.

In the first experiment, using a simple semi static Huffman coding as symbolwise compressor, we improved the compression ratio of the *Flexible Parsing* with LZW-dictionary by 3 to 5 percent on texts such as the bible.txt file or the prefixes of English Wikipedia data base (see Table A.1). We obtain that the smaller is the file the greater is the gain.

We have experimental evidence that many of the most relevant LZ77-like commercial compressors are, following our definition, dictionary-symbolwise

File (size)	bible.txt (4047392 Byte)	enwik (100 MB)
gzip -9	29.07%	36.45%
lzwfp	30.09%	35.06%
lzwhds	25.84%	31.79%

Table A.1: Compression ratio comparison of some LZW-like compressors and the gzip tool. (*gzip -9* is the gzip compression tool with the -9 parameter for maximum compression. *lzwfp* is the Flexible Parsing algorithm of Matias-Rajpoot-Sahinalp with a LZW-like dictionary. *lzwhds* is our Dictionary-Symbolwise Flexible Parsing algorithm with LZW-like dictionary and Huffman codes.)

File (size)	bible.txt (4047392 Byte)	enwik (100 MB)
gzip -9	29.07%	36.45%
gzip by 7zip	27.44%	35.06%
zip by 7zip	25.99%	33.72%
cabarc	22.13%	28.46%
lzhds-32KB	27.47%	35.02%
lzhds-64KB	26.20%	33.77%
lzhds-2MB	22.59%	28.82%
lzhds-16MB	22.51%	26.59%

Table A.2: Compression ratio comparison of some LZ77-like compressors. (*gzip -9* is the gzip compression tool with the -9 parameter for maximum compression. *gzip by 7zip* is the gzip compression tool implemented in the 7-Zip compression suite. *zip by 7zip* is the 7-Zip implementation of the zip compression tool. *cabarc* is the MsZip cabinet archiving tool also known as cabarc (version 5.1.26 with -m lzx:21 option used). *lzhds-x* is our Dictionary-Symbolwise Flexible Parsing with LZ77-like dictionary of different dictionary sizes, as stated in the suffix of the name, and Huffman codes.)

File (size)	bible.txt (4047392 Byte)	enwik (100 MB)
gzip -9 / lzhds-32KB	105.82%	104.08%
gzip by 7zip / lzhds-32KB	99.89%	100.11%
zip by 7zip / lzhds-64KB	99.19%	99.85%
cabarc / lzhds-2MB	97.96%	98.75%

Table A.3: Ratio between the compression ratio of different LZ77-like compressors. All the involved compressors, except for the gzip one, seam to have an optimal parsing strategy. (See Table A.2 Caption for compressor descriptions.) Notice that on each row there are compressors having the same windows size.

algorithms and they use an optimal parsing (see Table A.2 and Table A.3). In Table A.3 is shown the ratio between compression performances of compressors with similar constrains and encoding. Indeed, *gzip* and *lzhds-32KB* use a LZ77-like dictionary of 32KB, *zip* and *lzhds-64KB* have dictionary size of 64KB. *cabarc* and *lzhds-2MB* use 2MB as dictionary size. They all use Huffman codes. We notice that a difference of about 5 percent is due to parsing optimality while small differences of about 2 percent are due to implementation details like different codeword space and different text block handling. We think that *gzip* and *zip* implementations in the 7-Zip compression suite and *cabarc* have an optimal parsing, even if this fact is not clearly stated or proved.