Optimal Parsing in Dictionary-Symbolwise Data Compression Schemes

G. Della Penna\textsuperscript{1}, A. Langiu\textsuperscript{2}, F. Mignosi\textsuperscript{1}, A. Ulisse\textsuperscript{1}

\textsuperscript{1} Dipartimento di Informatica, Università degli Studi dell’Aquila, Italy
\textsuperscript{2} Dipartimento di Matematica e Appl., Università degli Studi di Palermo, Italy

Abstract. In this paper we introduce dictionary-symbolwise data compression schemes. We describe a method that, under some natural hypothesis, allows to obtain an optimal parse of any input or of any part of an input. This method can also be used to approximate the optimal parse in the general case and, under some additional hypothesis, it gives rise to on-line data compression algorithms. Therefore it could be used to improve many common compression programs. As second main contribution, we show how to use DAWG’s and CDAWG’s in a variant of the LZ77 compression scheme. In particular, we give an on-line linear implementation of our method in the case of dictionary-symbolwise algorithms with unbounded history and any on-line statistical coding.

1 Introduction

In the scientific literature there are many data compression methods that can be organized in some “cocktails” to improve compression rates. Some of them use transforms to preprocess the text before the real compression phase. For instance, StarZip in its second version (see [25]) uses the StarNT transform before exploiting bzip2, that, in turns, takes the advantage of the Burrows-Wheeler transform before using a block-sorting text compression algorithm and an Huffman encoding (cf. the introduction to the manual of bzip2 [9]).

There are two famous compression methods that can work together: symbolwise and dictionary encoding. These are sometimes referred to as statistical encoding and parsing (or macro) encoding, respectively. In [2] it is possible to find a survey on these schemes up to the publications date and also some deep relationships among them (see also [26]).

We say that if a data compression algorithm uses both a dictionary and a symbolwise encoding on literal characters, then it belongs to the class of dictionary-symbolwise compressors. A more precise definition will be given in the next section.

Bell, in his 1986 paper (cf. [1]) implements a suggestion made earlier by Storer and Szymanski in 1982 (see [24]), i.e. to use free mixture of dictionary pointers and literal characters in the Lempel-Ziv algorithms, literals only being used when a dictionary pointer takes up more space than the characters it codes. Bell’s implementation of this scheme is called LZSS, and adds an extra bit to each pointer or character to distinguish between them.
Even if in LZSS there is no symbolwise compression, from a purely formal point of view we say that LZSS is dictionary-symbolwise since we can consider the identity as a symbolwise compressor. Moreover, we can say that the whole class of dictionary-symbolwise compressors derived conceptually from it.

Many papers deal with optimal parsing in dictionary schemes (see for instance [24, 22, 15, 2, 5, 16, 22, 18, 27, 17, 11]. For optimal parsing we mean a way of parsing the input string such that the cost of the encoded output string is the cheapest one. Different encodings for dictionary pointers lead to different cost functions. Sometimes the cost of a pointer is considered constant. In this case an optimal parsing obtains the smallest number of possible phrases on any input string. Under certain conditions (which apply to LZ78, LZW, and other existing schemes) this phrase optimality translates into bit optimality. In this paper we consider a natural extension of this optimality notion to the case of dictionary-symbolwise schemes.

A classic way for obtaining an optimal parsing is by reduction to a well-known graph theoretical problem. This fact holds true also for dictionary-symbolwise schemes. This approach, however, is not recommended in [23] because it is too time consuming. In [18] a pruning mechanism is described, that allows a more efficient computation and that still enables the evaluation of an optimal solution for the original graph. The pruning process may be applied in all cases for which the cost function satisfies the triangle inequality.

It is written in [18] that, “the optimal method and its new variant apply to any static dictionary-based compression method with static (fixed- or variable-length) encoding. The elements to be encoded can be of any kind: strings, characters, \((d,l)\) pairs etc., and any combination thereof. The proposed technique thus improves a very broad range of different methods, many of which have been published in the scientific literature or as patents.”

In this paper we show a novel method that allows to obtain an optimal parse in linear time of any input or of any part of an input under a natural hypothesis. Our main result can be seen as a generalization to the dictionary-symbolwise case of the main result of [5] and improves the non-greedy parsing considered in [12] and used by gzip. This method can also be used to approximate the optimal parse in the general case, i.e. when the hypothesis does not always hold. We use no pruning mechanism. We just define a graph of linear size (that is a subgraph of generalization of the classical graph model considered in [18]), and we show how to construct this graph in linear time. Under some additional hypothesis, it is possible to build both the graph and the optimal parse in an on-line linear manner.

As second main result we show, for the first time to our best knowledge, how to use DAWG and CDAWG (cf. [3, 8, 13, 14]) in LZ77 compression schemes, instead of using Suffix trees (cf. [21] and reference therein) or hashing (cf. [4, 10]).

This paper is organized as follows. In the next section we will introduce some preliminary results and some basic definitions. In Section 3 we will define a graph of linear size that is a subgraph of the classical graph model and we
will prove, under an hypothesis, that to find an optimal parsing is equivalent to find a shortest path in it. In Section 4, as the second main contribution of this paper, we show how to use DAWG’s and CDAWG’s in LZ compression schemes. In particular in we give an on-line linear implementation of our method in the case of LZ77-symbolwise schemes with unbounded history. In the final section we will describe some variation of our parsing algorithm, describe further researches and state some open problem.

2 Preliminaries and Basic Definitions

We assume that the reader is familiar with dictionary encoding and with some simple statistical encoding such as the Huffman one. Moreover, all over in this section it is assumed that algorithms are sequential and left to right.

A dictionary compression, as authors of [2] point out, achieve compression by replacing consecutive groups of symbols (phrases) with a code. The code can be regarded as an index (or pointer) into a dictionary, which is a list of frequently used phrases. Dictionaries range from an ad-hoc collection of letter pairs to the adaptive dictionaries used for Ziv-Lempel coding (cf [28, 29]). In the case of LZ77 schemes, the dictionary is a part of the previously parsed text and it is sometimes called “history”.

A dictionary algorithm can be fully described by.

1. The dictionary description. This includes a complete algorithmic description on how it is built and updated.
2. The way of coding dictionary pointers.
3. The parsing algorithm.

Generally the parsing is done by a greedy method, i.e. at any stage, the longest matching element from the dictionary is sought. Most dictionary algorithms concentrate on describing the dictionary and the way of coding pointers, and tacitly assume that greedy parsing will be applied.

A symbolwise compression algorithm assigns to each letter \( a_i \) of a text \( T = a_1a_2a_3 \cdots a_n \) a code that can depend on the letter, on the position and on the context.

A dictionary-symbolwise algorithm uses both a dictionary algorithm and a symbolwise compression and parses the text with a free mixture of these dictionary pointers and literal characters. Moreover, it must algorithmically describe how to encode the flag information on what is the algorithm in use (dictionary or symbolwise) at each step. Often, as in the case of LZSS, an extra bit is added to each pointer or character to distinguish between them.

Notice that, at the theoretical level and just for a better understanding of the scheme, all symbols in the text can also be in the dictionary (i.e. the dictionary is general, following a definition of [16]). Therefore, in this case, it must be clear if a character that is a parse phrase will be coded as a pointer to the dictionary or by the code given by the symbolwise compression. In practice, pointers to characters are usually avoided in dictionary-symbolwise algorithms.
A dictionary-symbolwise algorithm can be therefore fully described by:

1. The dictionary description. This includes a complete algorithmic description on how it is built and updated.
2. The way of coding dictionary pointers.
3. The parsing algorithm.
4. The symbolwise algorithm.
5. The encoding algorithm of the flag information.

The parse algorithm decides whether a parse phrase is coded by the symbolwise or by the dictionary compressor.

We subdivide dictionary-symbolwise algorithms into three classes.

The first class includes all algorithms in which, for any text, the encodings of symbols and of dictionary pointers and of the flag informations do not depend on the specific parse. For instance, an algorithm in this class can decide that for any position $i$ the encoding of symbol $a_i$ depends only on the symbol itself and on the previous one $a_{i-1}$ (if it exists) The encoding would be the same both if the parse created by the algorithm includes a dictionary word ending with the letter $a_{i-1}$ or not. As another example, an algorithm in this class can decide that the encoding of symbols is done by a static Huffman algorithm on all symbols of the text.

The second class includes all algorithms not belonging to the first class in which, for any text, both the encoding of dictionary pointers starting in position $i$ and the encoding of symbol $a_i$ in the symbolwise compressor depend only on previous parsed symbols, i.e. on the portion of the text already parsed. For instance an algorithm that performs a dynamic Huffman algorithm only on the literals considered by the generated parse, belongs to this class.

The third class includes all remaining dictionary-symbolwise algorithms. For instance an algorithm that performs a static Huffman algorithm only on all the literals considered by the generated parse, belongs to this class.

Suppose now that we have defined all the features described above that fully describe a dictionary-symbolwise algorithm except for the parsing algorithm, i.e. we have described the dictionary, the dictionary pointers, the flag information and the symbolwise compressor. We call it a dictionary-symbolwise scheme.

To define what is an optimal parsing, in such dictionary-symbolwise scheme, we must specify a cost function $C$ that associates to any coded text a real number greater than or equal to 0. An optimal parsing of a text $T$ is a parsing $T = u_1 \cdots u_s$ where for each $i$ it is defined a flag boolean function $Fl$ that, for $i = 1, \ldots, s$ indicates whether the word $u_i$ has to be coded as a pointer or as a symbol, such that the encoding of the couple $(u_1 \cdots u_s, Fl)$ minimize the function $C$. Typically the function $C$ is chosen to be the length of the coded text, but, in general it is a real valued function, whose domain is the set of all possible strings over the final encoding alphabet. Usually the function cost can be considered additive. This means that in a dictionary-symbolwise scheme, function $C$ is defined if it is given the cost of the encodings of dictionary pointers, the cost of the encodings of symbols and the cost of the encoding of the flag
boolean function $F_l$. In this case the cost turns out to be the sum of all costs separately. In practice there are few obvious details to add to put all these encodings together at an encoding price that can often be included in each single cost. In what follows, it will be assumed that the function cost is additive without explicitly mention it again.

Classically, in the case of pure dictionary methods, the problem of finding an optimal parse is reduced to find a shortest path in a graph. More precisely, a directed, labelled graph $G = (V, E)$ is defined for the given text $T = a_1a_2a_3 \cdots a_n$ of length $n$. The set of vertices is $V = \{0, 1, \ldots, n, n\}$, with vertex $i$ corresponding to the character $a_i$ for $i \leq n$, and $n$ corresponding to the end of the text. $E$ is the set of directed edges where an ordered pair $(i, j)$, with $i < j$, belongs to $E$ if and only if the corresponding substring of the text, that is the sequence of characters $v = a_i \cdots a_{j-1}$, is a member of the dictionary. The label $L_{i,j}$ is defined for every edge $(i, j) \in E$ as the cost of encoding the pointer to $v$ for the given encoding scheme at hand. The problem of finding the optimal parsing of the text, relative to the given dictionary and encoding scheme, therefore reduces to the well-known problem of finding the shortest path in $G$ from vertex 0 to vertex $n$. Graph $G$ contains no cycles and it is already naturally ordered in a topological order. Therefore by a simple dynamic programming method, the shortest path can be found in $O(|E|)$. Unfortunately the number of edges in $E$ can be quadratic and this motivated the search of suboptimal alternatives (cf. [15]) or pruning techniques (cf. [18]).

Our approach is different. We looked for some hypothesis that, in the more general setting of the dictionary-symbolwise scheme, could allow us to build a linear size subgraph of the classical one while maintaining the correspondence between optimal parsing of the text and shortest path in $G$ from vertex 0 to vertex $n$.

3 Modeling with a graph

For any dictionary-symbolwise scheme having the property that any algorithm within this scheme falls in the first class, we can define a directed, labelled graph $G = (V, E)$ for the given text $T = a_1a_2\cdots a_n$ of length $n$ in the following way.

The set of vertices is $V = \{0, 1, \ldots, n\}$, with vertex $i$ corresponding to the character $a_i$ for $i \leq n$, and $n$ corresponding to the end of the text. $E$ is the set of directed edges. $L$ is the set of labels, where $L_{i,j}$ is the label of the edge $(i, j)$. We conceptually distinguish two kinds of edges. Edges of the first kind are of the form $(i, i + 1)$ for any $i < n$. Hence the graph contains the base path, following the definition in [16]. The label $L_{i,i+1}$ is the cost of the coding of symbol $a_i$ given by the symbolwise compression algorithm plus the cost of a flag to be sent in the case $a_i$ is chosen to be part of a parse.

Edges of the second kind come from the dictionary, and more precisely from a greedy parsing. An ordered pair $(i, j)$, with $i < j$, belongs to $E$ if and only if the corresponding substring of the text, that is, the sequence of characters $v = a_i \cdots a_{j-1}$ is the longest matching element from the dictionary starting from
letter $a_i$. The label $L_{i,j}$ of the edge $(i, j)$ of the second kind is the cost of encoding the pointer to $v$ for the given encoding scheme at hand plus the cost of a flag to be sent in the case $v$ is chosen to be part of a parse.

Notice that, from a purely formal point of view, we can have edges of the first and of the second kind from vertex $i$ to vertex $i + 1$, and therefore $G$ should be a labeled multigraph. But, since the cost of pointer is usually much bigger that the cost of a symbol, to avoid these situations, in this paper we do not include in $G$ edges of the second kind having length equal to one. All the results presented in this paper do not change if these edges are added to $G$.

Notice that the graph above can be considered as a subgraph of the natural generalization of the classical graph. Nevertheless, since there are algorithms that fall in the second or third class, we state the following remark.

**Remark 1.** The graph described above is always well defined, even for dictionary-symbolwise schemes that include algorithms belonging to the second or third class, except for the label function. If a scheme includes an algorithm that belongs to the second class it is possible to define labels for each parsing algorithm, not for the scheme. If a scheme includes an algorithm that belongs to the third class it is possible to define labels by using approximations of the expected costs.

Even in the case of a pure dictionary scheme, the labels of the classical graph cannot be always precisely defined. This is more evident in the case of dictionary-symbolwise scheme and obliged us to find out and formally define the three classes. As an example of a pure dictionary scheme where labels cannot be “a priori” defined, consider a LZ77 scheme where the encodings of dictionary pointers are subsequently compressed by a static Huffman algorithm, and where the cost function is the final length. It is clear that the label, i.e. the cost of a single pointer edge in the classical graph depends on all other pointers in the chosen parse.

Graph $G$ contains no cycles and it is already naturally ordered in a topological order. Thus, by a simple dynamic programming method, the shortest path can be found in $O(|E|)$. This is $O(n)$ because each vertex $i$ has at most two outgoing edges, one of the first kind, i.e. $(i, i + 1)$, and one of the second kind.

From now on in this section let us suppose that we have a fixed dictionary-symbolwise scheme. We suppose that all algorithms in this scheme belong to the first class and that furthermore we have defined a cost function $C$.

We state now a first hypothesis on the function $C$ and on the dictionary of the fixed parsing scheme.

**Hypothesis 1** The cost of encoding dictionary pointers plus the cost of encoding the flag information for any dictionary pointer is a positive constant $c$. The cost of encoding symbols plus the cost of encoding the flag information is greater than or equal to zero and always smaller than $c$. The dictionary is suffix closed.

**Theorem 1.** Suppose we have a fixed dictionary-symbolwise scheme such that any algorithm within this scheme falls in the first class defined in previous section. Suppose further that Hypothesis 1 holds true. The problem of finding the
optimal parsing of the text, reduces to the problem of finding the shortest path in
$G$ from vertex 0 to vertex $n$.

Proof. Under the hypotheses of the theorem, the graph $G$ is fully defined, including labels. Suppose now that we have an optimal parsing, represented by the couple $(u_1 \cdots u_s, F_l)$ where the boolean function $F_l$ for $i = 1, \ldots, s$ indicates whether the word $u_i$ has to be coded as a pointer or as a symbol. Any sub-parsing of an optimal parsing must be also optimal, for otherwise we could replace it with another with smaller cost. If all the parse phrases of this optimal parsing are either symbols or are greedy choices of elements in the dictionary, then this parsing corresponds to a path in graph $G$ and, since it is optimal, it is a shortest path. Suppose now that in this optimal parse there are some phrases that are non greedy choices and let $u_1$ be the one with the smallest index. Let us suppose that $u_i$ is the parse of text symbols $a_j \cdots a_{j+k}$ where $k = |u_i| - 1$. A greedy choice of an element in the dictionary would therefore give $a_j \cdots a_{j+k'}$ with $k' > k$.

We consider now two cases. If position $j + k'$ is an end position of a phrase $u_i$ in the optimal parsing then we can replace the parse phrases $u_i \cdots u_{i'}$ with the single dictionary phrase $a_j \cdots a_{j+k'}$. This last phrase has a cost that is smaller than or equal to the cost of $u_i \cdots u_{i'}$ because $u_i$ is a pointer phrase and because we suppose that Hypothesis 1 holds. If position $j + k'$ is not an end position of a phrase in the optimal parsing then there must exists a parse phrase $u_i$ that parses the of text symbols $a_j \cdots a_{j+k}$ with $j' \leq j + k'$ and $j' + h > j + k'$. Since the dictionary is suffix closed, we can replace the parse phrases $u_i \cdots u_{i'}$ with the two phrases $a_j \cdots a_{j+k'}, a_{j+k'+1} \cdots a_{j'+h}$. These two phrases by Hypothesis 1 have a cost that is smaller than or equal to $u_i \cdots u_{i'}$.

In both cases we have obtained a new optimal parsing where the first phrase that is not a greedy choice is strictly closer to the end of the text than in the original optimal parsing. By iterating this argument (more formally by induction on the number $j$) we obtain an optimal parsing where all phrases are either symbols or greedy dictionary choices, and this concludes the proof.

We notice that the condition on the constant cost of pointers stated in Hypothesis 1 has been considered in several research papers (cf. [5, 20] and references therein) and have been usually considered for LZ78 and similar schemes. For what concerns LZ77 and derived schemes that use non constant costs of pointers, we notice that the algorithm for shortest path in $G$ from vertex 0 to vertex $n$ induce a parsing algorithm that strictly improves the non greedy parsing considered in [12] and in gzip and corresponds to an unbounded MAXDELTA. The algorithm for shortest path in $G$ can also be defined when considering schemes such that some algorithm in this scheme belongs to the second or to the third class. Indeed, following Remark 1, in the case of the second class the labels in the graph $G$ can be set for any given parsing algorithm. In this case the parsing algorithm is the one induced by the shortest path. The labels of edges outgoing vertex $i$ can be defined once all previous labels and the parsing up to $i$ have been created. This is possible because the shortest path algorithm proceeds left to right following the topological order, and, once the new labels have
been set, the algorithm can further proceed. When considering schemes such that some algorithm in this scheme belongs to the third class, following Remark 1, it is possible to define labels by using approximations of the expected costs. In all these cases the obtained parse can be considered as an approximation of the optimal one.

4 DAWGs in LZ77 Algorithms and On-line Solution

In this section we want to show how to build the graph $G$, in the case of the LZ77 algorithm (or in a similar algorithm) possibly in linear time and in an on-line manner.

Several techniques are available at the moment. We distinguish between building the graph without the labels and labeling it. For building the graph without the labels one can use classical data structures such as suffix trees or truncated suffix trees (see \cite{21,19} and references therein) where it is shown how to find the longest match that is present in the dictionary and how to find out a pointer to it in the dictionary (or, equivalently, a position). In LZ77 and in its variations, one must be careful in defining the dictionary. When we are looking for a longest match starting from position $i$ of the text, the dictionary can be all the text previously seen, or the latest $N$ symbols of the text up to current position. Some variants allow the longest match to start in any position smaller than the current one, i.e. the longest match can self-overlap, i.e. the dictionary can include letters after the current position providing that they are at least explored during the phase of searching the longest match. In other variants the dictionary includes a lookahead buffers, etc.

At the moment, to our best knowledge, factor automata and compacted factor automata were not used in LZ77 alike algorithms even if they have some interesting properties that favorably compare to suffix trees. Compact factor automata (CDAWGs) indeed require in average less memory than suffix trees or truncated suffix trees over the same texts. Moreover there exist on-line linear algorithms for building them, even with a sliding window (see \cite{14,13}).

In this section we show how to use CDAWGs in a LZ77 variation, where the longest match in position $i$ can start in any position smaller than the current one. The parsing algorithm is the one induced by our shortest path algorithm in our graph $G$. In order to be well defined, this dictionary-symbolwise algorithm has to describe the symbolwise algorithm together the encoding pointers algorithm and the way of encoding the flag information. Clearly, if the global algorithm must be linear and on-line, every sub-algorithm must also be on-line. Moreover the labels of graph $G$ must be well defined. In this section we are mainly interested in how to define the edges of $G$ by using CDAWGs and not on how to define labels. For this last purpose the reader can see the discussion that follows the proof of Theorem 1.

In the classical algorithm described in \cite{13} for building on-line in linear time the CDAWG of a given text, two information are always available in constant time at any step of the algorithm. The first is the length of the longest suffix
of the text that appears in another position \( j \) of the text and the second is this number \( j \). The second information is important to describe pointers and their subsequent encoding: however, for the moment, we do not use it.

Suppose then that we have a text \( T = a_1a_2\cdots a_n \) and that we can have, using the CDAWG of \( T \), in linear time (even on-line) the array \( S \) where \( S[i] \), \( i = 1, \ldots n \) is the length of the longest suffix of \( a_1a_2\cdots a_i \) that also appears in another previous position. For instance if \( T = ababaababaababaababaababaababaababaababaababaababaababaababaababaababaababaababaababaababaababaabab... \) then \( S[1] = 0 \) (the empty word \( \epsilon \) is the longest suffix of \( a \) that appears previously), \( S[2] = 0, S[3] = 1, S[4] = 1, S[5] = 2 \), etc. The complete sequence of \( S[i] \), for \( i = 1, \ldots, 21 \) is

\[
0, 0, 1, 1, 2, 3, 2, 3, 4, 5, 6, 4, 5, 6, 7, 8, 9, 10, 11, 7, 8.
\]

For any text \( T \), \( S[1] = 0 \) and, therefore, we have no information from this number.

We want to find the array \( L \) where \( L[i], i = 1, \ldots n - 1 \) is the length of the longest prefix of \( a_{i+1} \cdots a_n \) that also appears in another previous position of the text \( T \).

For instance, if \( T \) is the text of previous example, \( L[1] = 0 \) (the empty word \( \epsilon \) is the longest prefix of of \( baababaababaababaababaababaababaababaababaababaababaababaababaababaababaababaababaababaababaababaababaababaabab... \) that appears previously), \( L[2] = 1, L[3] = 3, L[4] = 2 \), etc. The complete sequence of \( L[i] \), for \( i = 1, \ldots, 20 \) is

\[
0, 1, 3, 2, 6, 5, 4, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1.
\]

If we have array \( L \) then we can describe the pointer edges in graph \( G \). For instance, since \( L[4] = 2 \) then there is an edge outgoing vertex 4 and reaching vertex 6 = 4 + 2 or equivalently \((4, 6)\) is an edge of \( G \). Since \( L[8] = 11 \) then there is an edge outgoing vertex 8 and reaching vertex 19 = 8 + 11 or equivalently \((8, 19)\) is an edge of \( G \).

The question now is: there exists a linear time algorithm, possibly on-line, that starting from array \( S \) gives us array \( L \)?

The answer is yes and a complete description of the algorithm is the following. For \( i \) ranging from 1 up to \( n \) pick any two consecutive values such that \( S[i+1] \neq S[i] + 1 \). For instance in previous example the sequence of values \( i \) such that \( S[i+1] \neq S[i] + 1 \) is 1, 3, 6, 11, 19, 21. The value \( n \) (in previous example \( n = 21 \)) is always considered in this sequence. The couple of consecutive such values are \((1, 3), (3, 6), (6, 11), (11, 19), (19, 21)\). If \((i, i')\) is such a couple, define for \( j \) ranging from \( i - S[i] \) up to \((i' - S[i']) - 1 \), \( L[j] = i - j \).

For instance if \((i, i') = (1, 3)\) we have that \( i - S[i] = 1 - 0 = 1 \), \((i' - S[i']) - 1 = (3 - 1) - 1 = 1 \) and therefore \( L[j] = L[1] = i - j = 1 - 1 = 0 \).

If \((i, i') = (3, 6)\) we have that \( i - S[i] = 3 - 1 = 2 \), \((i' - S[i']) - 1 = (6 - 3) - 1 = 2 \) and therefore \( L[j] = L[2] = i - j = 3 - 2 = 1 \).

If \((i, i') = (6, 11)\) we have that \( i - S[i] = 6 - 3 = 3 \), \((i' - S[i']) - 1 = (11 - 6) - 1 = 4 \) and therefore for \( j = 3 \) \( L[j] = L[3] = i - j = 6 - 3 = 3 \) and for \( j = 4 \) \( L[j] = L[4] = i - j = 6 - 4 = 2 \).

To complete the algorithm we define for \( j \) ranging from \( n - S[n] \) up to \( n \), \( L[j] = n - j \).
The reason why this algorithm works relies on the fact that if the longest suffix that appears in a previous position of \( a_1 \cdots a_i \) has length \( h \) and the longest suffix the appears in a previous position of \( a_1 \cdots a_{i+1} \) has length smaller than \( h + 1 \), then the longest prefix of \( a_{i-h} \cdots a_n \) that appears in a position smaller than \( i-h \) has length \( h \). The formal proof of the correctness of this algorithm is left to the reader.

5 Further Improvements and Conclusions

In the case of LZ77 and similar algorithms, the cost of a dictionary pointer usually logarithmically depends also on the size of the parse phrase that the pointer is referring to. Therefore, if a portion of a text can be parsed in more than one way by dictionary pointers, usually the smaller cost is obtained by the most asymmetrical parse, i.e. when the lengths of the parse phrases are as far as possible from an uniform length. For this reason we made the following heuristic that will allow a better flexibility of parsing. We add some new edges to graph \( G \), one at most for each vertex. If \((i, j)\) is a pointer edge of \( G \) and if \((j, t)\) is another pointer edge, we look for the smallest integer \( j' \), with \( i < j' \leq j \) such that \((j', t)\) is still an edge of \( G \). If \( j' \neq j \) then we add \((i, j')\) to the set of edges of graph \( G \). We keep unchanged the rest of the parsing techniques described in this paper. An easy standard method allow us to efficiently obtain these new edges. Indeed we define a new array \( v[t] \), \( t = 1, \ldots, n \), where in \( V[t] \) it is stored the smallest \( j' \) such that \((j', t)\) is an edge of the graph. Array \( V \) can be build as soon as the edges are added to \( G \). Therefore simple table lookup allow to find vertex \( j' \) starting from vertex \( j \).

Another improvement can come from making cocktails among dictionary-symbolwise algorithms. For instance, in real texts, some symbol can be totally predictable from the previous one, i.e. a symbolwise algorithm could potentially erase them, while a dictionary-symbolwise algorithm has to pay in any case the flag information. Therefore, if before performing a dictionary-symbolwise algorithm one use a compressor such as the anti-dictionary one described in [7, 6] or some variation of PPM*, that erases predictable symbols, there are chances to improve the compression ratio.

We have experimented a prototypal simplified version of some of the algorithms described here using a simple Huffman as symbolwise algorithm and the results seems encouraging. For instance, we obtained a gain of more than ten per cent over the file Book1 of Calgary corpus compared to gzip -9, while keeping comparable decompression speed. Much more experiments have to be done, mixing other symbolwise compressors. Clearly many classical problems and results obtained in the case of dictionary algorithms are still open in the case of dictionary-symbolwise ones.
References