Parsing with Suffix and Prefix Dictionaries

Martin Cohn and Roger Khazan
Computer Science Department
Brandeis University
Waltham, MA 02254, USA

Abstract

We show that greedy left-to-right (right-to-left) parsing is optimal w.r.t. a suffix (prefix) dictionary. To exploit this observation, we show how to construct a static suffix dictionary that supports on-line, linear-time optimal parsing. From this we derive an adaptive on-line method that yields compression comparing favorably to LZW.

1 Introduction

Optimal string parsing with respect to a dictionary of strings has obvious implications for data compression. The class of pointer-based text compressors work by replacing successive substrings of an input string with addresses or pointers to entries in a dictionary of substrings. The process is analogous to vector quantization, where the dictionary is called the "code book." The decompressor simply concatenates for output substrings corresponding to the successive pointers. Various scenarios can be contemplated: The compressor and decompressor may share a fixed dictionary (sometimes called "static dictionary compression"); the compressor may transmit to the decompressor a dictionary (possibly itself compressed) before transmitting the pointers; or the compressor and decompressor may evolve a common dictionary through the coding process (sometimes called "dynamic dictionary compression." The latter is the basis for the family of compressors called LZ'78 (or LZII), LZSS, and LZW, based on the work of Ziv and Lempel, although even these algorithms may resort to the shared-fixed model once a certain dictionary size has been reached.

The shared, fixed dictionary model is appropriate to applications such as frequent downloading of software or font files, where transmission time and cost warrant the storage cost of a resident dictionary in the host and in its satellites. Large but frequently used files can then be quickly downloaded when needed by the satellites, used, and then discarded. In a more uniformly endowed network, each site may serve as host for a fraction of the files to be shared, equalizing local storage demand.

In the case of a shared, fixed dictionary, a given input string commonly may be parsed in many different ways into dictionary strings. This observation is most conspicuous when the dictionary contains, among others, all the unit-length strings representing the characters of the input alphabet. An optimal encoding is one whose cost in terms of pointers is minimal. When pointer costs are not uniform, optimal encoding is complicated; with uniform pointer costs, an optimal encoding is one with the minimal number of phrases. In this paper we assume the uniform-cost model, and we consider the case where the dictionary has either the *prefix* or *suffix* property. A prefix (suffix) dictionary is one where for any string in the dictionary, all its prefixes (suffixes) must also be in the dictionary. The well-known LZ'78 and LZW compression algorithms dynamically construct prefix dictionaries. We are unaware of suffix-dictionary constructions being published.

We first consider the case of a fixed, shared, prefix (suffix) dictionary, and describe a linear-time, optimal parsing algorithm. We do not consider pretransmission of the dictionary. Finally we discuss how the compressor and decompressor can evolve a common suffix dictionary during transmission. This adaptive shared-dictionary growth is competitive with LZW in compression, and has the same asymptotic time complexity.

2 Optimal Parsings

Suppose we are given a *prefix (suffix) dictionary* D of strings over the alphabet Σ, with the property that for any string in D, all its prefixes (suffixes) are also in D. To assure that greedy parsing can succeed, we assume that all alphabet symbols are in D. We wish to parse an input string $x \in \Sigma^+$ into a minimal number of substrings, each a string in D. The ultimate goal is to find a compressed form of the string x relative to the prefix (suffix) dictionary D; minimizing the number of substrings corresponds to the uniform coding cost for strings in D, under which the length of a compressed input is proportional to the number of substrings in the parsing.

We use the notation $x_i^j, \;\; i \leq j$ for the substring consisting of the i^{th} through the j^{th} symbols of the input string x; x_i denotes i^{th} symbol. $|D|$ denotes the number of strings in D, while d_{max} denotes the length of the longest string in D. The number p of phrases, or substrings in the parsing $x = v_1 v_2 \cdots v_p$ is called the *cost* of the parsing.

To put the following results in context, recall that optimal parsings relative to an *arbitrary* fixed dictionary can be found in time $O(|x|^2)$ via dynamic programming, as first proposed in [Wagner'73]. It is an off-line algorithm that requires the entire input before beginning; it is the best, known result. For prefix dictionaries, there is a left-to-right algorithm with cost $O(|x| \times d_{max})$.[HartmanR'85] If the prefix dictionary was formed by a process like LZ parsing, successive insertions can grow by at most one symbol, so $d_{max} = O(\sqrt{|x|})$, and this algorithm runs in time $O(|x|^{3/2})$.

A *greedy* parsing with respect to dictionary D at each step parses the longest phrase that can be found in D. Given a prefix (suffix) dictionary, greedy parsing of the input from left to right (right to left) relative to D is not necessarily optimal. Moreover, the cost of greedy parsing can exceed the optimal cost by a ratio which grows proportionately with input length. For example, let the prefix dictionary $D = \{0, 1, 101\} \bigcup \{10^i | 1 \leq i \leq n\}$, and the input string 1010^n. Optimal parsing is $10, 10^n$, while greedy parsing is $101, 0, 0, \ldots, 0$. [De Agostino'94] This example is certainly contrived to behave badly; in our comparisons we use dictionaries generated by "more natural" means, namely LZ-family parsings.

Our first result is the following:

Theorem: With respect to a suffix dictionary D, greedy parsing left-to-right is optimal; dually, with respect to a prefix dictionary D, greedy parsing right to left is optimal.

Proof: We prove just the suffix-dictionary case. Let $x = u_1, u_2, \ldots, u_p$ be any, possibly optimal, parsing of x into p phrases of D. Suppose u_1, the initial phrase of the parsing, is a proper prefix of the longest dictionary word w that is also a prefix of x. Then by the suffix property, there exists a parsing $x = w, v, u_r, u_{r+1}, \ldots, u_p$, where $r \geq 3$, and $v \in D$ is a suffix of u_{r-1}. The total number of phrases is thus $p - r + 3 \leq p$. By choosing w and each successive phrase greedily, we cannot do worse than any parsing.

Two related observations follow; define $C_{SD}(x)$ to be the cost of parsing x w.r.t. the suffix dictionary SD and C_{PD} the cost w.r.t. the prefix dictionary PD.

Corollary: If $a \in \Sigma$ and $x \in \Sigma^+$ then $C_{SD}(x) \leq C_{SD}(ax)$, $C_{PD}(x) \leq C_{PD}(xa)$.

Proof: We prove the first claim: Consider an optimal parsing of ax w.r.t. SD; if a alone is the initial phrase, then clearly $C_{SD}(x) = C_{SD}(ax) - 1$. Otherwise let $y = az$ be the initial phrase, $z \in \Sigma^+$. Then $az \in SD$ and by the suffix property, $z \in SD$, so z is the initial phrase of some parsing of x, and $C_{SD}(x) \leq C_{SD}(ax)$.

Corollary: If D has either the suffix or the prefix property, for $x, y \in \Sigma^+$, $C_D(x) \leq C_D(xy)$.

3 LZ Dictionaries

The construction of a prefix dictionary may be accomplished by any of several well-known algorithms based on the work of Lempel and Ziv [LempelZ'76,ZivL'78,Welch'84]. We will use two of them, LZ'76 and LZW.

3.1 LZ'76

The parsing proposed by Lempel and Ziv [LZ'76] for measuring the complexity of a finite sequence and, later, to compress [ZL'78] can be described as follows: Repeatedly parse from the input and insert into the dictionary the shortest prefix of the input that is not yet in the dictionary. According to this, the string $0, 01, 1, 011, 00,$ is parsed as shown by the commas, yielding the dictionary D consisting of exactly the phrases shown. We ignore the question of dictionary capacity — what to do if the data structure fills up. The string shown suffices to prove that greedy left-to-right parsing relative to a LZ'76 dictionary can fail to be optimal. Greedy left-to-right parsing relative to D (which is *not* the same as the LZ parsing) consists of the five phrases $00, 1, 1, 011, 00$. Greedy right-to-left parsing yields $0, 011, 011, 00$, with four phrases.

3.2 LZW Dictionary

A modified parsing and encoding was proposed by Welch [Welch'84] and forms the basis of UNIX **compress**, a constituent of **pkzip** and other compression packages. The parsing rule here is to initialize the dictionary with all single symbols in Σ, then repeatedly parse from the input its longest prefix that is already in the dictionary, while inserting into the dictionary the shortest prefix that is not yet in the dictionary. For the string 0100100011 the final dictionary is $\{0, 1, 01, 10, 00, 010, 000, 011\}$ Greedy left-to-right parsing yields $010, 010, 00, 1, 1$ with five phrases. The optimal, greedy right-to-left parsing is 010,01,00,011 with only four phrases.

3.3 Experimental Results

For each input file in the Calgary Corpus, a static dictionary was constructed (without size bound) by a left-to-right pass over the data, using the LZ'76 rule, which tends to form longer dictionary words than does LZW. The input was then parsed greedily left-to-right and greedily right-to-left, and the numbers of phrases recorded. The dictionary size is simply the number of phrases in the LZ'76 parsing. Under uniform coding, the code length of each dictionary entry would be $\lceil \lg_2 |D| \rceil$ bits. The resulting compression ratio shown, $bytes_{out}/bytes_{in}$, is slightly pessimistic, insofar as two-level encoding or progressive-length encoding could have saved a few percent. On the other hand, input characters in some of the files could have been reckoned as costing seven bits. No compression results are claimed for the LZ'76 parsing, because its true coding cost would have to include the cost of a raw character along with each pointer.

File Name	Size (bytes)	Number of Phrases LZ'76	GLR	GRL	Percent Difference	Encoding Bytes	Compression Ratio
bib	111261	21407	20683	19024	8.0	35670	32.06
book1	768771	131007	129904	124239	4.3	248478	32.32
book2	610856	102421	99390	93871	5.6	187742	30.73
geo	102400	26211	30282	25979	1.4	48711	47.57
news	377109	73389	71550	67643	5.4	109920	29.15
obj1	21504	5983	5852	5665	3.2	9206	42.81
obj2	246814	50858	51158	46599	8.9	93198	37.76
paper1	53161	12130	11697	11049	5.5	19336	36.37
paper2	82199	17320	16977	16036	5.5	30068	36.58
pic	513216	26576	28014	25709	8.2	48205	09.39
progc	39611	9394	9015	8511	5.5	14895	37.60
progl	71646	13543	12802	11908	6.9	20839	29.09
progp	49379	9764	9196	8566	6.8	14991	30.36
trans	93695	18125	16577	15259	7.9	28611	30.54

Table 1: Compression of Calgary Corpus with Static Prefix Dictionary: Numbers of Parsed Phrases, Percent Difference between GLR and GRL Parsing, Coded Size, and Compression Ratios for GRL.

4 Suffix Dictionaries

The dual claim of the main result says that left-to-right greedy parsing is optimal with respect to a *suffix* dictionary. If a static suffix dictionary is assumed available to both compressor and decompressor, as in file distribution, left-to-right parsing gives optimal compression, and can be performed on-line. Alternatively, we propose a way by which common dictionaries can be evolved during compression and decompression of the first part of the input, in the spirit of LZ algorithms. The difference is that suffix rather than prefix dictionaries must be built. We first consider the use of static dictionaries, their performance as compressors, and the data structures necessary for fast parsing. Then we consider adaptive dictionary growth, either for its own sake or as a prelude to static-dictionary compression. We conclude by outlining a linear-time adaptive algorithm.

4.1 Static Suffix Dictionaries

In the trie structure for a prefix dictionary every node represents a dictionary entry. So the trie has exactly as many internal nodes as there are dictionary entries, and greedy left-to-right parsing entails following a path from the root until a failure is encountered. This means that parsing proceeds in linear time, and the constants depend only on alphabet size, not on dictionary size. In contrast, the trie implementing a suffix dictionary can have some internal nodes that do not correspond to any dictionary entry. There are two consequences: First, the number of nodes can be quadratic in $|D|$, the number of dictionary entries. The prefix trie that contains all prefixes of 0^n1^n has exactly $2n$ nodes; the (suffix) trie containing the reversals of these prefixes has n^2 nodes. The case $n = 4$ is illustrated in Figure 1. Second, since greedy parsing entails following a path from the root until no exit is possible, a naive approach might proceed the full depth of the trie only to discover no match beyond depth 1. The corresponding time bound would be $O(nd_{max})$, because d_{max} is the depth of the trie. (It is interesting that this is identical to the bound of Hartman and Rodeh.) We next show how augmenting the trie can guarantee an $O(n)$ time bound.

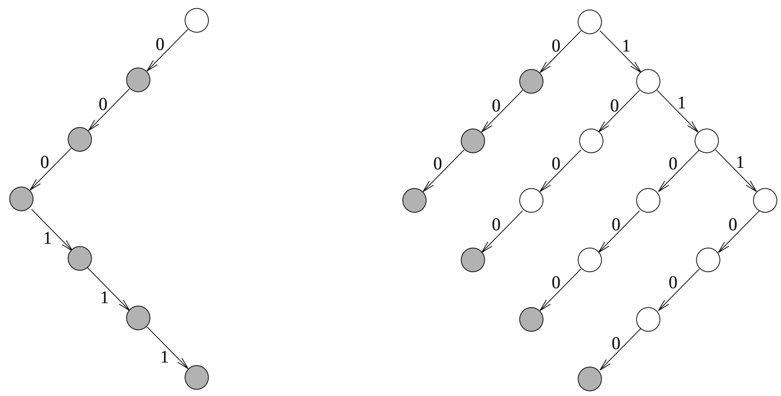

Figure 1: Tries for Prefixes of 0^41^4 and their Reversals. Word Nodes are Darkened.

4.2 Implementation

In this section we describe a data structure for the static suffix dictionary that permits left-to-right parsing in time linear with input length. In a later section we expand this to an adaptive compression algorithms. Before giving the construction, we remark that since the decompressor's task is to convert node pointers into strings, *its* dictionary should consist of reversed dictionary words stored in a cursor array with parent pointers.

The compressor's data structure is an augmented trie. Recall that the suffix trie contains internal nodes some of which represent dictionary words, and some of which don't; the former we shall call "word nodes." Starting with the suffix-dictionary trie, we add two items to each internal node q: First, a pointer $P(q)$ to its nearest ancestor which is a word node; if q is itself a word node, $P(q) = q$. Second, a "failure function" $F(q)$, which points to the internal node in the trie which represents the substring between $P(q)$ and q. This substring is a suffix of the string q, so must appear in the trie. The preprocessing is accomplished by any traversal of the suffix trie that assures that at every node, the parent's $F()$ value has already been defined. Depth-first or breadth-first traversal suffice. At each node the traversal performs the following:

/$*$ Define SUCC(q', a) to be the successor of node q' under symbol a $*$/
if (q is a word node) **then** $P(q) \leftarrow q$; $F(q) \leftarrow q_0$;
else $P(q) \leftarrow P(q')$; $F(q) \leftarrow F(q')a$; /$*$ where q is SUCC(q', a) $*$/

Trie traversals can be performed in time proportional to the number of nodes.

Now the input is traversed left to right. For each input symbol that causes a failure, we parse the current ancestral word node and follow the failure-function edge, until the trie path itself can be followed. At this point the next input symbol is read. (See the procedure below.) After this terminates, any remaining terminal substring is handled by an additional loop: **do** PARSE $P(q)$; $q \leftarrow F(q)$ **until** $(q = q_0)$. The critical observation is that traversal takes time proportional to the number of symbols read. Each symbol takes us one node deeper into the trie, but failure edges always lead to a shallower node, so the total number of edges traversed is at most twice the number of input symbols.

Algorithm SD_PARSE (q_0, x, n)
  $q \leftarrow q_0$;
  for i from 1 **to** n
    while (($q' \leftarrow$ SUCC(q, x_i)) = FAIL) PARSE $P(q)$; $q \leftarrow F(q)$;
    $q \leftarrow q'$;
end

4.3 Dynamic Dictionary

Another approach is to consider the case of adaptive suffix-dictionary growth by compressor and decompressor. We will sketch an algorithm, that runs in linear time. The method we propose is a variant on Lempel/Ziv compression, wherein the parsing builds a dictionary by inserting substrings of the input.

One would like to reverse the Welch algorithm and do the following: Suppose the input string is x_1^n, and the algorithm has already parsed x_1^i. At the next step, parse and encode by a dictionary pointer the longest prefix of x_{i+1}^n that can be found in the dictionary. Then insert into the dictionary the string $x_i^i x_{i+1}^n$, i.e. the longest match prepended by the symbol immediately preceding it in the input. Unfortunately, this doesn't work, as can be seen from the input 010101..01. The dictionary begins with 0 and 1; it gains 01 and 101, but never anything else. The algorithm lacks "guaranteed progress" [SeeryZ'77] whereby a new dictionary entry is guaranteed at every parse.

Here is a new algorithm that does guarantee progress and compares favorably with LZW: We assume that D contains all the symbols of Σ. Suppose the input string is x_1^n, and the algorithm has already parsed x_1^t into the $i-1$ phrases $v_1 v_2 \cdots v_{i-1}$. At the next step, parse and encode by a dictionary pointer the longest prefix of x_{t+1}^n that can be found in the dictionary; call it v_i. Then insert into the dictionary the string $u_i v_i$ where u_i is the shortest suffix of x_1^t such that $u_i v_i$ is not yet in the dictionary. The decompressor knows the strings v_i and x_1^t, and therefore can compute and insert $u_i v_i$ into its own dictionary. It is clear that, except when parsing the very first character of x_1^n, this rule guarantess progress in the sense that it always inserts a new phrase into the dictionary. It could fail to do so only if the string $x_1^t v_i$ were already in the dictionary, but that is impossible, because at this point $x_1^t v_i$ has not been read, let alone parsed.

The longest match v_i from position $t+1$ is bounded by $O(d_{max})$, as is the shortest prefix u_i such that $u_i v_i$ is not in the dictionary. We now establish a bound on d_{max} under the new insertion policy.

Lemma: $|u_i| \leq |v_{i-1}| + 1$
Proof: At the time of the $i-1^{st}$ parse, the string $v_{i-1}v_i$ was not in the dictionary. Otherwise the greedy parse would have taken it, rather than just v_{i-1}. At the i^{th} parse, $v_{i-1}v_i$ could have just entered (consider the input 010101011), but $av_{i-1}v_i$ could not be present, otherwise its suffix $v_{i-1}v_i$ must also have been present.

Corollary: Let d_i be the dictionary string inserted at the i^{th} parse; then $|d_i| \leq |v_{i-1}| + |v_i| + 1$.

Corollary Let P be the total number of phrases in the parsing of an input. Summing the previous result we have $\sum_{i=1}^{P} d_i \leq 2n + P$.

Theorem: Let D be a suffix dictionary constructed as described above from an input of length n, and let d_{max} denote the length of the longest string ever inserted into D. Then $d_{max} = O(\sqrt{n})$.
Proof: If D has a string of length d_{max}, by the suffix property it must also have strings of lengths $1, 2, 3, \ldots, d_{max}-1$. Therefore $\sum_{i=1}^{d_{max}} i \leq \sum_{i=1}^{P} |d_i| \leq 2n + P \leq 3n$. It follows that $d_{max} = O(\sqrt{n})$.

4.4 Implementation

For the dynamic, on-line implementation, the suffix dictionary is built as the data are parsed and encoded. The data structure that is built is similar to that of the static case but with a slightly modified failure function. The failure function of a word node is no longer routinely reset to the root, but to $F(q')a$ as with any other node. This allows us to find the next string to be inserted into the dictionary, namely the unique one-character left extension of the longest dictionary word ending at the point of failure, without backtracking. It also facilitates the insertion of just the suffix of the new entry that is lacking in the dictionary. These properties are vital to ensure linear-time parsing. Our structure extends classical string-matching techniques [AhoC'75, KnuthMP'77]. The parsing algorithm differs from these classical string matching techniques in its use of dynamic insertion and in its application to parsing. A more complete description, including the use of path compression, is planned.

4.5 Experimental Data

Table 2 shows the results of a) the dynamic on-line algorithm (DynSD) just described and b) greedy left-to-right parsing w.r.t. the static dictionary SD formed by DynSD. The first is compared to LZW, and the second to optimal parsing w.r.t. the static dictionary formed by LZ'76. The comparisons suggest that the dynamic algorithm is as effective as LZW and that the dictionary formed by DynSD contains substrings at least as "typical" as those parsed by LZ'76.

Calgary Corpus		Dynamic LR Parsings		Optimal Static Parsing	
Name	Size	DynSD	LZW	GLR/SD	GRL/LZ'76
bib	111261	24857	26861	17218	19024
book1	768771	152299	154227	121921	124239
book2	610856	114671	120684	86961	93871
geo	102400	43477	42839	26466	25979
news	377109	87233	90905	63161	67643
obj1	21504	8704	9068	5050	5665
obj2	246814	64789	68091	42728	46599
paper1	53161	14633	15370	9872	11049
paper2	82199	20457	21332	14830	16036
pic	513216	34954	35058	26442	25709
progc	39611	11264	11979	7461	8511
progl	71646	15147	16525	10148	11908
progp	49379	10942	12017	7228	8566
trans	93695	19463	22441	12338	15259
		Table 2a		Table 2b	

Table 2a) Comparison of dynamic LR parsing with LZW parsing.
Table 2b) Comparison of greedy parsings w.r.t. static dictionaries created by the SD and LZ'76 algorithms. For SD, GLR is optimal; for LZ'76, GRL is optimal.

5 Conclusion

We have shown that left-to-right (right-to-left) greedy parsing is optimal with respect to a suffix (prefix) dictionary. This observation lead to an on-line (off-line) compression algorithm using a static suffix (prefix) dictionary, which compares favorably to LZ'76 compression. Even more interesting is an on-line algorithm using a dynamically built suffix dictionary. This might be used to build a dictionary later to be used for optimal, greedy parsing. But experimental results suggest that the adaptive construction alone compresses better than LZW, and needn't be just an adjunct to static compression. A future publication will present a complete description and analysis of this algorithm, possibly with the addition of path-compression. It also remains to be proven that the adaptive construction/compression is asymptotically optimal; an adaptation of the proof in [LZ'78] should be possible.

6 Bibliography

[AhoC'75] A.V. Aho and M.J. Corasick, "Efficient String Matching: an Aid to Bibliographic Search", Comm. *ACM* 18(6) (1975) pp. 333-340.

[De Agostino'94] S. De Agostino, "Sub-Linear Algorithms and Complexity Issues for Lossless Data Compression", Master's Thesis, Brandeis University, (May 1994)

[HartmanR'85] A. Hartman and M. Rodeh "Optimal Parsing of Strings", *Combinatorial Algorithms on Words*, Springer-Verlag (A. Apostolico and Z. Galil, editors), 155-167.

[KnuthMP'77] D.E. Knuth, J.H. Morris, and V.R. Pratt, "Fast Pattern Matching in Strings", *SIAM J. on Computing* 6(2) (1977) pp. 323-350.

[LempelZ'76] A. Lempel and J. Ziv "On the Complexity of Finite Sequences", *IEEE Transactions on Information Theory* 22:1, 75-81.

[SeeryZ'77] J. B. Seery and J. Ziv, "A Universal Data Compression Algorithm: Description and Preliminary Results", *77-1212-6*, Bell Laboratories, Murray Hill, N.J.

[Wagner'73] R. A. Wagner, "Common Phrases and Minimum-Space Text Storage", *Communications of the ACM* 16:3, 148-152.

[Welch'84] T. A. Welch, "A Technique for High-Performance Data Compression", *IEEE Computer* 17:6, 8-19.

[ZivL'78] J. Ziv and A. Lempel, "Compression of Individual Sequences Via Variable-Rate Coding", *IEEE Transactions on Information Theory* 24:5, 530-536.