Proceedings SWAT 88, First Scandinavian Workshop on Algorithm Theory. Lecture Notes in Computer Science, vol. 318, pp. 176–189. Halmstad, Sweden (1988)
3. Baeza-Yates, R., Schott, R.: Parallel searching in the plane. Comput. Geom. Theor. Appl. **5**, 143–154 (1995)
4. Blum, A., Raghavan, P., Schieber, B.: Navigating in Unfamiliar Geometric Terrain. In: On Line Algorithms, pp. 151–155, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, Providence RI (1992) Preliminary Version in STOC 1991, pp. 494–504
5. Demaine, E., Fekete, S., Gal, S.: Online searching with turn cost. Theor. Comput. Sci. **361**, 342–355 (2006)
6. Gal, S.: Minimax solutions for linear search problems. SIAM J. Appl. Math. **27**, 17–30 (1974)
7. Gal, S.: Search Games, pp. 109–115, 137–151, 189–195. Academic Press, New York (1980)
8. Hipke, C., Icking, C., Klein, R., Langetepe, E.: How to Find a point on a line within a Fixed distance. Discret. Appl. Math. **93**, 67–73 (1999)
9. Kao, M.-Y., Reif, J.H., Tate, S.R.: Searching in an unknown environment: an optimal randomized algorithm for the cow-path problem. Inf. Comput. **131**(1), 63–79 (1996) Preliminary version in SODA '93, pp. 441–447
10. Lopez-Ortiz, A.: On-Line Target Searching in Bounded and Unbounded Domains: Ph. D. Thesis, Technical Report CS-96-25, Dept. of Computer Sci., Univ. of Waterloo (1996)
11. Lopez-Ortiz, A., Schuierer, S.: The Ultimate Strategy to Search on m Rays? Theor. Comput. Sci. **261**(2), 267–295 (2001)
12. Papadimitriou, C.H., Yannakakis, M.: Shortest Paths without a Map. Theor. Comput. Sci. **84**, 127–150 (1991) Preliminary version in ICALP '89
13. Schuierer, S.: Lower bounds in on-line geometric searching. Comput. Geom. **18**, 37–53 (2001)

Detour

► Planar Geometric Spanners
► Dilation of Geometric Networks
► Geometric Dilation of Geometric Networks

Deutsch Algorithm

► Quantum Algorithm for the Parity Problem

Deutsch–Jozsa Algorithm

► Quantum Algorithm for the Parity Problem

DHT

► P2P

Dictionary-Based Data Compression

1977; Ziv, Lempel

TRAVIS GAGIE, GIOVANNI MANZINI
Department of Computer Science,
University of Piemonte, Oriental, Alessandria, Italy

Synonyms

LZ compression; Ziv–Lempel compression; Parsing-based compression

Problem Definition

The problem of lossless data compression is the problem of compactly representing data in a format that admits the faithful recovery of the original information. Lossless data compression is achieved by taking advantage of the redundancy which is often present in the data generated by either humans or machines.

Dictionary-based data compression has been "the solution" to the problem of lossless data compression for nearly 15 years. This technique originated in two theoretical papers of Ziv and Lempel [15,16] and gained popularity in the "80s" with the introduction of the Unix tool `compress` (1986) and of the gif image format (1987). Although today there are alternative solutions to the problem of lossless data compression (e. g., Burrows-Wheeler compression and Prediction by Partial Matching), dictionary-based compression is still widely used in everyday applications: consider for example the zip utility and its variants, the modem compression standards V.42bis and V.44, and the transparent compression of pdf documents. The main reason for the success of dictionary-based compression is its unique combination of compression power and compression/decompression speed. The reader should refer to [13] for a review of several dictionary-based compression algorithms and of their main features.

Key Results

Let T be a string drawn from an alphabet Σ. Dictionary-based compression algorithms work by parsing the input into a sequence of substrings (also called words) $T_1, T_2, \ldots, T_d$ and by encoding a compact representation of these substrings. The parsing is usually done incrementally and on-line with the following iterative procedure. Assume the encoder has already parsed the substrings $T_1, T_2, \ldots, T_{i-1}$. To proceed, the encoder maintains a dictionary of potential candidates for the next word T_i and associates a unique codeword with each of them. Then,

it looks at the incoming data, selects one of the candidates, and emits the corresponding codeword. Different algorithms use different strategies for establishing which words are in the dictionary and for choosing the next word T_i. A larger dictionary implies a greater flexibility for the choice of the next word, but also longer codewords. Note that for efficiency reasons the dictionary is usually not built explicitly: the whole process is carried out implicitly using appropriate data structures.

Dictionary-based algorithms are usually classified into two families whose respective ancestors are two parsing strategies, both proposed by Ziv and Lempel and today universally known as LZ78 [16] and LZ77 [15].

The LZ78 Algorithm

Assume the encoder has already parsed the words $T_1, T_2, \ldots, T_{i-1}$, that is, $T = T_1 T_2 \cdots T_{i-1} \hat{T}_i$ for some text suffix $\hat{T}_i$. The LZ78 dictionary is defined as the set of strings obtained by adding a single character to one of the words $T_1, \ldots, T_{i-1}$ or to the empty word. The next word T_i is defined as the longest prefix of $\hat{T}_i$ which is a dictionary word. For example, for $T = aabbaaabaabaabba$ the LZ78 parsing is: $a, ab, b, aa, aba, abaa, bb, a$. It is easy to see that all words in the parsing are distinct, with the possible exception of the last one (in the example the word a). Let T_0 denote the empty word. If $T_i = T_j\alpha$, with $0 \le j < i$ and $\alpha \in \Sigma$, the codeword emitted by LZ78 for T_i will be the pair (j, α). Thus, if LZ78 parses the string T into t words, its output will be bounded by $t \log t + t \log |\Sigma| + \Theta(t)$ bits.

The LZ77 Algorithm

Assume the encoder has already parsed the words $T_1, T_2, \ldots, T_{i-1}$, that is, $T = T_1 T_2 \cdots T_{i-1} \hat{T}_i$ for some text suffix $\hat{T}_i$. The LZ77 dictionary is defined as the set of strings of the form $w\alpha$ where $\alpha \in \Sigma$ and w is a substring of T starting in the already parsed portion of T. The next word T_i is defined as the longest prefix of $\hat{T}_i$ which is a dictionary word. For example, for $T = aabbaaabaabaabba$ the LZ77 parsing is: $a, ab, ba, aaba, abaabb, a$. Note that, in some sense, $T_5 = abaabb$ is defined in terms of itself: it is a copy of the dictionary word $w\alpha$ with w starting at the second a of T_4 and extending into T_5! It is easy to see that all words in the parsing are distinct, with the possible exception of the last one (in the example the word a), and that the number of words in the LZ77 parsing is smaller than in the LZ78 parsing. If $T_i = w\alpha$ with $\alpha \in \Sigma$, the codeword for T_i is the triplet (s_i, ℓ_i, α) where s_i is the distance from the start of T_i to the last occurrence of w in $T_1 T_2 \cdots T_{i-1}$, and $\ell_i = |w|$.

Entropy Bounds

The performance of dictionary-based compressors has been extensively investigated since their introduction. In [15] it is shown that LZ77 is optimal for a certain family of sources, and in [16] it is shown that LZ78 achieves asymptotically the best compression ratio attainable by a finite-state compressor. This implies that, when the input string is generated by an ergodic source, the compression ratio achieved by LZ78 approaches the entropy of the source. More recent work has established similar results for other Ziv–Lempel compressors and has investigated the rate of convergence of the compression ratio to the entropy of the source (see [14] and references therein).

It is possible to prove compression bounds without probabilistic assumptions on the input, using the notion of *empirical entropy*. For any string T, the order k empirical entropy $H_k(T)$ is the maximum compression one can achieve using a uniquely decodable code in which the codeword for each character may depend on the k characters immediately preceding it [6]. The following lemma is a useful tool for establishing upper bounds on the compression ratio of dictionary-based algorithms which hold pointwise on every string T.

Lemma 1 ([6, Lemma 2.3]) *Let $T = T_1 T_2 \cdots T_d$ be a parsing of T such that each word T_i appears at most M times. Then, for any $k \ge 0$*

$$d \log d \le |T| H_k(T) + d \log(|T|/d) + d \log M + \Theta(kd + d),$$

where $H_k(T)$ is the k-th order empirical entropy of T. □

Consider, for example, the algorithm LZ78. It parses the input T into t distinct words (ignoring the last word in the parsing) and produces an output bounded by $t \log t + t \log |\Sigma| + \Theta(t)$ bits. Using Lemma 1 and the fact that $t = O(|T|/\log T)$, one can prove that LZ78's output is at most $|T| H_k(T) + o(|T|)$ bits. Note that the bound holds for any $k \ge 0$: this means that LZ78 is essentially "as powerful" as any compressor that encodes the next character on the basis of a finite context.

Algorithmic Issues

One of the reasons for the popularity of dictionary-based compressors is that they admit linear-time, space-efficient implementations. These implementations sometimes require non-trivial data structures: the reader is referred to [12] and references therein for further reading on this topic.

Greedy vs. Non-Greedy Parsing

Both LZ78 and LZ77 use a greedy parsing strategy in the sense that, at each step, they select the longest prefix of the unparsed portion which is in the dictionary. It is easy to see that for LZ77 the greedy strategy yields an optimal parsing; that is, a parsing with the minimum number of words. Conversely, greedy parsing is not optimal for LZ78: for any sufficiently large integer m there exists a string that can be parsed to $O(m)$ words and that the greedy strategy parses in $\Omega(m^{3/2})$ words. In [9] the authors describe an efficient algorithm for computing an optimal parsing for the LZ78 dictionary and, indeed, for any dictionary with the prefix-completeness property (a dictionary is prefix-complete if any prefix of a dictionary word is also in the dictionary). Interestingly, the algorithm in [9] is a one-step lookahead greedy algorithm: rather than choosing the longest possible prefix of the unparsed portion of the text, it chooses the prefix that results in the longest advancement in the *next* iteration.

Applications

The natural application field of dictionary-based compressors is lossless data compression (see, for example [13]). However, because of their deep mathematical properties, the Ziv–Lempel parsing rules have also found applications in other algorithmic domains.

Prefetching

Krishnan and Vitter [7] considered the problem of prefetching pages from disk into memory to anticipate users' requests. They combined LZ78 with a pre-existing prefetcher P_1 that is asymptotically at least as good as the best memoryless prefetcher, to obtain a new algorithm P that is asymptotically at least as good as the best finite-state prefetcher. LZ78′s dictionary can be viewed as a trie: parsing a string means starting at the root, descending one level for each character in the parsed string and, finally, adding a new leaf. Algorithm P runs LZ78 on the string of page requests as it receives them, and keeps a copy of the simple prefetcher P_1 for each node in the trie; at each step, P prefetches the page requested by the copy of P_1 associated with the node LZ78 is currently visiting.

String Alignment

Crochemore, Landau and Ziv-Ukelson [4] applied LZ78 to the problem of sequence alignment, i. e., finding the cheapest sequence of character insertions, deletions and substitutions that transforms one string T into another T' (the cost of an operation may depend on the character or characters involved). Assume, for simplicity, that $|T| = |T'| = n$. In 1980 Masek and Paterson proposed an $O(n^2/\log n)$-time algorithm with the restriction that the costs be rational; Crochemore et al.'s algorithm allows real-valued costs, has the same asymptotic cost in the worst case, and is asymptotically faster for compressible texts.

The idea behind both algorithms is to break into blocks the matrix $A[1\ldots n, 1\ldots n]$ used by the obvious $O(n^2)$-time dynamic programming algorithm. Masek and Paterson break it into uniform-sized blocks, whereas Crochemore et al. break it according to the LZ78 parsing of T and T'. The rationale is that, by the nature of LZ78 parsing, whenever they come to solve a block $A[i\ldots i', j\ldots j']$, they can solve it in $O(i' - i + j' - j)$ time because they have already solved blocks identical to $A[i\ldots i'-1, j\ldots j']$ and $A[i\ldots i', j\ldots j'-1]$ [8]. Lifshits, Mozes, Weimann and Ziv-Ukelson [8 recently used a similar approach to speed up the decoding and training of hidden Markov models.

Compressed Full-Text Indexing

Given a text T, the problem of compressed full-text indexing is defined as the task of building an index for T that takes space proportional to the entropy of T and that supports the efficient retrieval of the occurrences of any pattern P in T. In [10] Navarro proposed a compressed full-text index based on the LZ78 dictionary. The basic idea is to keep two copies of the dictionary as tries: one storing the dictionary words, the other storing their reversal. The rationale behind this scheme is the following. Since any non-empty prefix of a dictionary word is also in the dictionary, if the sought pattern P occurs within a dictionary word, then P is a suffix of some word and easy to find in the second dictionary. If P overlaps two words, then some prefix of P is a suffix of the first word—and easy to find in the second dictionary—and the remainder of P is a prefix of the second word—and easy to find in the first dictionary. The case when P overlaps three or more words is a generalization of the case with two words. Recently, Arroyuelo et al. [1] improved the original data structure in [10]. For any text T, the improved index uses $(2+\epsilon)|T|H_k(T) + o(|T|\log|\Sigma|)$ bits of space, where $H_k(T)$ is the k-th order empirical entropy of T, and reports all occ occurrences of P in T in $O(|P|^2\log|P| + (|P| + occ)\log|T|)$ time.

Independently of [10], in [5] the LZ78 parsing was used together with the Burrows-Wheeler compression algorithm to design the first full-text index that uses

$o(|T|\log|T|)$ bits of space and reports the *occ* occurrences of P in T in $O(|P| + occ)$ time. If $T = T_1T_2\cdots T_d$ is the LZ78 parsing of T, in [5] the authors consider the string $T_\$ = T_1\$T_2\$\cdots\$T_d\$$ where \$ is a new character not belonging to Σ. The string $T_\$$ is then compressed using the Burrows-Wheeler transform. The \$'s play the role of anchor points: their positions in $T_\$$ are stored explicitly so that, to determine the position in T of any occurrence of P, it suffices to determine the position with respect to any of the \$'s. The properties of the LZ78 parsing ensure that the overhead of introducing the \$'s is small, but at the same time the way they are distributed within $T_\$$ guarantees the efficient location of the pattern occurrences.

Related to the problem of compressed full-text indexing is the compressed matching problem in which text and pattern are given together (so the former cannot be preprocessed). Here the task consists in performing string matching in a compressed text without decompressing it. For dictionary-based compressors this problem was first raised in 1994 by A. Amir, G. Benson, and M. Farach, and has received considerable attention since then. The reader is referred to [11] for a recent review of the many theoretical and practical results obtained on this topic.

Substring Compression Problems

Substring compression problems involve preprocessing T to be able to efficiently answer queries about compressing substrings: e. g., how compressible is a given substring s in T? what is s's compressed representation? or, what is the least compressible substring of a given length ℓ? These are important problems in bioinformatics because the compressibility of a DNA sequence may give hints as to its function, and because some clustering algorithms use compressibility to measure similarity. The solutions to these problems are often trivial for simple compressors, such as Huffman coding or run-length encoding, but they are open for more powerful algorithms, such as dictionary-based compressors, BWT compressors, and PPM compressors. Recently, Cormode and Muthukrishnan [3] gave some preliminary solutions for LZ77. For any string s, let $C(s)$ denote the number of words in the LZ77-parsing of s, and let LZ77(s) denote the LZ77-compressed representation of s. In [3] the authors show that, with $O(|T|\,\text{polylog}(|T|))$ time preprocessing, for any substring s of T they can: *a*) compute LZ77(s) in $O(C(s)\log|T|\log\log|T|)$ time, *b*) compute an approximation of $C(s)$ within a factor $O(\log|T|\log^*|T|)$ in $O(1)$ time, *c*) find a substring of length ℓ that is close to being the least compressible in $O(|T|\ell/\log\ell)$ time. These bounds also apply to general versions of these problems, in which queries specify another substring t in T as context and ask about compressing substrings when LZ77 starts with a dictionary already containing the words in the LZ77 parsing of t.

Grammar Generation

Charikar et al. [2] considered LZ78 as an approximation algorithm for the NP-hard problem of finding the smallest context-free grammar that generates only the string T. The LZ78 parsing of T can be viewed as a context-free grammar in which for each dictionary word $T_i = T_j\alpha$ there is a production $X_i \to X_j\alpha$. For example, for T = *aabbaaabaabaabba* the LZ78 parsing is: *a*, *ab*, *b*, *aa*, *aba*, *abaa*, *bb*, *a*, and the corresponding grammar is: $S \to X_1\ldots X_7X_1, X_1 \to a, X_2 \to X_1b, X_3 \to b, X_4 \to X_1a, X_5 \to X_2a, X_6 \to X_5a, X_7 \to X_3b$. Charikar et al. showed LZ78's approximation ratio is in $O((|T|/\log|T|)^{2/3}) \cap \Omega(|T|^{2/3}\log|T|)$; i. e., the grammar it produces has size at most $f(|T|)\cdot m^*$, where $f(|T|)$ is a function in this intersection and m^* is the size of the smallest grammar. They also showed m^* is at least the number of words output by LZ77 on T, and used LZ77 as the basis of a new algorithm with approximation ratio $O(\log(|T|/m^*))$.

URL to Code

The source code of the gzip tool (based on LZ77) is available at the page http://www.gzip.org/. An LZ77-based compression library zlib is available from http://www.zlib.net/. A more recent, and more efficient, dictionary-based compressor is LZMA (Lempel–Ziv Markov chain Algorithm), whose source code is available from http://www.7-zip.org/sdk.html.

Cross References

- ► Arithmetic Coding
- ► The Burrows-Wheeler Transform
- ► Compressed Text Indexing
- ► Boosting Textual Compression

Recommended Reading

1. Arroyuelo, D., Navarro, G., Sadakane, K.: Reducing the space requirement of LZ-index. In: Proc. 17th Combinatorial Pattern Matching conference (CPM), LNCS no. 4009, pp. 318–329, Springer (2006)
2. Charikar, M., Lehman, E., Liu, D., Panigraphy, R., Prabhakaran, M., Sahai, A., Shelat, A.: The smallest grammar problem. IEEE Trans. Inf. Theor. **51**, 2554–2576 (2005)
3. Cormode, G., Muthukrishnan, S.: Substring compression problems. In: Proc. 16th ACM-SIAM Symposium on Discrete Algorithms (SODA '05), pp. 321–330 (2005)

4. Crochemore, M., Landau, G., Ziv-Ukelson, M.: A subquadratic sequence alignment algorithm for unrestricted scoring matrices. SIAM J. Comput. **32**, 1654–1673 (2003)
5. Ferragina, P., Manzini, G.: Indexing compressed text. J. ACM **52**, 552–581 (2005)
6. Kosaraju, R., Manzini, G.: Compression of low entropy strings with Lempel–Ziv algorithms. SIAM J. Comput. **29**, 893–911 (1999)
7. Krishnan, P., Vitter, J.: Optimal prediction for prefetching in the worst case. SIAM J. Comput. **27**, 1617–1636 (1998)
8. Lifshits, Y., Mozes, S., Weimann, O., Ziv-Ukelson, M.: Speeding up HMM decoding and training by exploiting sequence repetitions. Algorithmica to appear doi:10.1007/s00453-007-9128-0
9. Matias, Y., Şahinalp, C.: On the optimality of parsing in dynamic dictionary based data compression. In: Proceedings 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '99), pp. 943–944 (1999)
10. Navarro, G.: Indexing text using the Ziv–Lempel trie. J. Discret. Algorithms **2**, 87–114 (2004)
11. Navarro, G., Tarhio, J.: LZgrep: A Boyer-Moore string matching tool for Ziv–Lempel compressed text. Softw. Pract. Exp. **35**, 1107–1130 (2005)
12. Şahinalp, C., Rajpoot, N.: Dictionary-based data compression: An algorithmic perspective. In: Sayood, K. (ed.) Lossless Compression Handbook, pp. 153–167. Academic Press, USA (2003)
13. Salomon, D.: Data Compression: the Complete Reference, 4th edn. Springer, London (2007)
14. Savari, S.: Redundancy of the Lempel–Ziv incremental parsing rule. IEEE Trans. Inf. Theor. **43**, 9–21 (1997)
15. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. Inf. Theor. **23**, 337–343 (1977)
16. Ziv, J., Lempel, A.: Compression of individual sequences via variable-length coding. IEEE Trans. Inf. Theor. **24**, 530–536 (1978)

Dictionary Matching

► Sequential Multiple String Matching

Dictionary Matching and Indexing (Exact and with Errors)

2004; Cole, Gottlieb, Lewenstein

MOSHE LEWENSTEIN
Department of Computer Science, Bar Ilan University, Ramat-Gan, Israel

Synonyms

Approximate dictionary matching; Approximate text indexing

Problem Definition

Indexing and *dictionary matching* are generalized models of pattern matching. These models have attained importance with the explosive growth of multimedia, digital libraries, and the Internet.

1. Text Indexing: In text indexing one desires to preprocess a text t, of length n, and to answer where subsequent queries p, of length m, appear in the text t.
2. Dictionary Matching: In dictionary matching one is given a dictionary D of strings $p_1, \ldots, p_d$ to be preprocessed. Subsequent queries provide a query string t, of length n, and ask for each location in t at which patterns of the dictionary appear.

Key Results

Text Indexing

The *indexing* problem assumes a large text that is to be preprocessed in a way that will allow the following efficient future queries. Given a query pattern, one wants to find all text locations that match the pattern in time proportional to the *pattern length and to the number of occurrences*.

To solve the indexing problem, Weiner [14] invented the *suffix tree* data structure (originally called a *position tree*), which can be constructed in linear time, and subsequent queries of length m are answered in time $O(m \log |\Sigma| + tocc)$, where *tocc* is the number of pattern occurrences in the text.

Weiner's suffix tree in effect solved the indexing problem for exact matching of fixed texts. The construction was simplified by the algorithms of McCreight and, later, Chen and Seiferas. Ukkonen presented an online construction of the suffix tree. Farach presented a linear time construction for large alphabets (specifically, when the alphabet is $\{1, \ldots, n^c\}$, where n is the text size and c is some fixed constant). All results, besides the latter, work by handling one suffix at a time. The latter algorithm uses a divide and conquer approach, dividing the suffixes to be sorted to even-position suffixes and odd-position suffixes. See the entry on Suffix Tree Construction for full details. The standard query time for finding a pattern p in a suffix tree is $O(m \log |\Sigma|)$. By slightly adjusting the suffix tree one can obtain a query time of $O(m + \log n)$, see [12].

Another popular data structure for indexing is suffix arrays. Suffix arrays were introduced by Manber and Myers. Others proposed linear time constructions for linearly bounded alphabets. All three extend the divide and conquer approach presented by Farach. The construction in [11] is especially elegant and significantly simplifies the divide and conquer approach, by dividing the suffix set into three groups instead of two. See the entry on Suffix Array Construction for full details. The query time for suffix arrays is $O(m + \log n)$ achievable by embedding additional lcp (longest common prefix) information into the