Implementation of algebraic algorithms for approximate pattern matching on compressed strings

Maria Fedorkina and Alexander Tiskin

1. Pattern matching and the LCS problem

Approximate matching is a natural generalization of classical (exact) pattern matching, allowing for some character differences between the pattern and a matching substring of the text. Given a pattern string p of length m and a text string t of length $n \geq m$, approximate pattern matching asks for all the substrings of the text that are similar to the pattern. We consider the classical approach to string comparison based on the following numerical measure of string similarity:

Definition 1.1. *Let a, b be strings. The* longest common subsequence (LCS) score *lcs(a, b) is the length of the longest string that is a subsequence of both a and b. Given strings a, b, the* LCS problem *asks for the LCS score lcs(a, b).*

Definition 1.2. *Given strings a, b, the* semi-local LCS problem *asks for the LCS scores as follows:*

- *the whole a against every substring of b* (string-substring LCS)
- *every prefix of a against every suffix of b* (prefix-suffix LCS)
- *every suffix of a against every prefix of b* (suffix-prefix LCS)
- *every substring of a against the whole b* (substring-string LCS)

In particular, string-substring LCS is closely related to approximate pattern matching, where a short fixed pattern string is compared to various substrings of a long text string.

2. LCS and the sticky braid monoid

The algebraic approach to the semi-local LCS problem is based on the monoid of sticky braids.

Definition 2.1. *The* sticky braid monoid *(a.k.a. the 0-Hecke Monoid of the symmetric group) of order n, denoted T_n is the monoid generated by the identity element ι and $n-1$ generators $g_1, \ldots, g_{n-1}$ defined by the relations:*

- *$g_i^2 = g_i$ for all i (idempotence)*
- *$g_i g_j = g_j g_i$ for all i, j, $j - i \geq 2$ (far commutativity)*
- *$g_i g_j g_i = g_j g_i g_j$ for all i, j, $j - i = 1$ (braid relations)*

where $i, j \in [1 : n-1]$.

The sticky braid monoid consists of $n!$ elements, which can be represented canonically by permutations of order n. An algorithm for multiplication of sticky braids (in permutation form) in time $O(n \log n)$ was given by the second author [4].

Intuitively speaking, the behavior of the LCS score under concatenation of strings is isomorphic to monoid multiplication of sticky braids. Therefore, it is possible to calculate the semi-local LCS of two strings a, b by partitioning one of the strings into two substrings and calling the algorithm recursively to obtain the semi-local LCS scores for each substring against the other string. The semi-local LCS scores in the subproblem and in the main problem are represented implicitly by sticky braids (in permutation form), and the subproblems are composed by sticky braid multiplication.

3. Grammar-compressed strings

Nowadays nearly all data used in science and technology are compressed. From an algorithmic viewpoint, it is natural to ask whether compressed strings can be processed efficiently without decompression. Early examples of such algorithms were given e.g. by Amir et al. [1] and by Rytter [3]; for a recent survey on the topic, see Lohrey [2]. Efficient algorithms for compressed strings can also be applied to achieve speedup over ordinary string processing algorithms for plain strings that are highly compressible.

The following generic compression model is well-studied, and covers many data compression formats used in practice.

Definition 3.1. *Let t be a string of length n. String t is said to be* grammar-compressed *if it is generated by a context-free grammar. A* context-free grammar *of length $\bar{n}$ is a sequence of $\bar{n}$ statements. A statement numbered k, $1 \leq k \leq \bar{n}$, has either the form $t_k = \alpha$ where α is an alphabet character, or the form $t_k = t_i t_j$ for some i, j, $1 \leq i, j < k$. For convenience we will also allow statements of the form $t_k = \epsilon$, where ϵ is the empty string.*

We will be discussing algorithms for the comparison of a plain (uncompressed) pattern string p of length m and a text string t of length n, compressed by a context-free grammar of length $\bar{n}$. The algorithm of Section 2 can be applied to perform approximate pattern matching efficiently in this setting.

The recursive nature of grammar compression makes it natural to apply the sticky braid approach. Since a statement produces a string that is the concatenation of two strings produced by previous statements, the calculation of the implicit semi-local LCS scores for the statement requires only multiplying the sticky braids corresponding to the previous statements using our implementation of the algorithm. The resulting algorithm takes $O(m\bar{n}\log m)$ time, as it makes $\bar{n}$ calls of the sticky multiplication subroutine, each running in time $O(m \log m)$.

4. Results

We have implemented the algorithm of [4]; to the best of our knowledge, it is the first existing implementation of this rather intricate algorithm. We have also implemented the algorithm of Section 3 calculating the semi-local LCS scores of a pattern p of length m and a text t compressed by a context-free grammar of length $\bar{n}$, and examined its performance on several examples of grammar-compressed strings.

Example 4.1. *The $\bar{n}$-th* Fibonacci string *is generated by the following context-free grammar:*

$$t_1 = B \quad t_2 = A \quad t_3 = t_2 t_1 \quad t_4 = t_3 t_2 \quad \ldots \quad t_{\bar{n}} = t_{\bar{n}-1} t_{\bar{n}-2}$$

E.g. the seventh Fibonacci string is "ABAABABAABAAB".

The length n of the $\bar{n}$-th Fibonacci string grows exponentially in $\bar{n}$. This suggests that our algorithm, running in time $O(m\bar{n}\log m)$ independent of n, should be substantially faster than the standard dynamic programming algorithm for calculating the LCS of two strings, running in time $O(mn)$.

We ran several experiments to examine the performance of our algorithm. In our experiments we generated the pattern strings randomly, drawing each character independently and equiprobably from the subset of letters of the Latin alphabet {'A', 'B', 'C'}. Using our algorithm, we calculated the LCS score for the pattern against a grammar-compressed Fibonacci string. We also calculated the LCS score for the pattern and the uncompressed Fibonacci string using dynamic programming and compared the resulting running times.

LCS calculation times (ms)				
Pattern length	Compressed text length	Uncompressed text length	Sticky braids (plain v. compressed)	Dynamic programming (plain v. uncompressed)
4	16	987	1	0
16	16	987	6	1
64	16	987	23	6
256	16	987	74	18
4	24	46368	2	15
16	24	46368	7	55
64	24	46368	29	211
256	24	46368	116	819
4	32	2178309	2	673
16	32	2178309	10	2550
64	32	2178309	38	9778
256	32	2178309	173	40124

We can see that even though dynamic programming performs better on short strings, the sticky braid algorithm starts to perform faster on longer strings. Additionally, the sticky braid algorithm keeps working even on larger Fibonacci strings that do not fit into the computer's memory uncompressed.

Fibonacci strings are an artificial construct that is not often used in practice. We now consider a more natural type of compression: the classical compression schemes LZ78 and LZW by Ziv, Lempel and Welch [6, 5].

Example 4.2. *The LZ78 and LZW compression schemes can both be represented a context-free grammar consisting of three sections:*

- *in the first section, all statements are of the form $t_k = \alpha$;*
- *in the second section, the first statement is of the form $t_k = \epsilon$ and all the following statements are of the form $t_k = t_i t_j$, where statement i, $i < k$, is from the second section, and statement j is from the first section;*
- *in the third section, the first statement is of the form $t_k = \epsilon$ and all the following statements are of the form $t_k = t_{k-1} t_j$, where statement $k-1$ is from the third section, and statement j is from the second section.*

We call context-free grammars of this form LZ-grammars.

The LZ-grammar corresponding to LZ78 or LZW compression might not be substantially shorter than the length of an uncompressed string. We construct a class of LZ-grammars corresponding to LZ78 compression that generates strings of length n growing quadratically in the grammar's length $\bar{n}$.

Example 4.3. *The* LZ78max-grammar *of length $\bar{n} = 3r + 3$ is an LZ-grammar defined as follows:*

$$\begin{array}{lll}
t_0 = \epsilon & u_0 = \epsilon & v_0 = \epsilon \\
t_1 = \alpha_1 & u_1 = u_0 t_1 & v_1 = v_0 u_1 \\
t_2 = \alpha_2 & u_2 = u_1 t_2 & v_2 = v_1 u_2 \\
\dots & \dots & \dots \\
t_r = \alpha_n & u_r = u_{r-1} t_r & v_r = v_{r-1} u_r
\end{array}$$

where α_k is the k-th character of the alphabet.
E.g. the LZ78max-grammar of length $18 = 3 \cdot 5 + 3$ generates the string "AABABCABCDABCDE" of length 15.

We ran several experiments to examine the performance of our algorithm. In our experiments we generated the pattern strings randomly, drawing each character independently and equiprobably from the uppercase letters of the Latin alphabet. Using our algorithm, we calculated the LCS score for the pattern against an LZ78max-grammar. We also calculated the LCS score for the pattern and the uncompressed LZ78max-grammar string using dynamic programming and compared the resulting running times.

LCS calculation times (ms)				
Pattern length	Compressed text length	Uncompressed text length	Sticky braids (plain v. compressed)	Dynamic programming (plain v. uncompressed)
4	195	2145	15	0
16	195	2145	48	2
64	195	2145	169	8
256	195	2145	687	35
4	1539	131841	94	39
16	1539	131841	339	142
64	1539	131841	1376	532
256	1539	131841	5774	2242
4	12291	8394753	763	2594
16	12291	8394753	2807	9599
64	12291	8394753	12282	35108
256	12291	8394753	49187	153342

Again, we see that even though dynamic programming performs better on short strings, the sticky braid algorithm starts to perform faster on longer strings.

5. Conclusion and future work

Our experiments demonstrate that the algebraic string comparison approach of [4] is not only of theoretical interest, but can also give substantial speedups on problems of practical significance, such as approximate matching on grammar-compressed strings, which includes the classical LZ78 and LZW compression schemes as a special case. Further work may involve generalising our implementation to deal with scoring schemes other than LCS (e.g. edit distance matching), and using it for efficient approximate pattern matching on large compressed datasets.

References

[1] A Amir, G Benson, and M Farach. Let Sleeping Files Lie: Pattern Matching in Z-Compressed Files. *Journal of Computer and System Sciences*, 52(2):299–307, 1996.

[2] M Lohrey. Algorithmics on SLP-compressed strings: a survey. *Groups Complexity Cryptology*, 4(2):241–299, 2012.

[3] W Rytter. Algorithms on Compressed Strings and Arrays. In *Proceeedings of SOFSEM*, volume 1725 of *Lecture notes in Computer Science*, pages 48–65, 1999.

[4] Alexander Tiskin. Fast Distance Multiplication of Unit-Monge Matrices. *Algorithmica*, 71:859–888, 2015.

[5] T A Welch. A Technique for High-Performance Data Compression. *Computer*, 17(6):8–19, 1984.

[6] G Ziv and A Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24:530–536, 1978.

Maria Fedorkina
St. Petersburg School of Physics, Mathematics, and Computer Science
Higher School of Economics
St. Petersburg, Russia
e-mail: `s17b2_fedorkina@179.ru`

Alexander Tiskin
Dept. of Mathematics and Computer Science
St. Petersburg University
St. Petersburg, Russia
e-mail: `a.tiskin@spbu.ru`