Paweł Gawrychowski

Wyszukiwanie wzorca w skompresowanym tekście

Rozprawa doktorska napisana pod kierunkiem
prof. Krzysztofa Lorysia, Uniwersytet Wrocławski

Instytut Informatyki
Uniwersytet Wrocławski
2011

Streszczenie

W niniejszej rozprawie rozważamy szereg problemów związanych z wyszukiwaniem wzorca w skompresowanym tekście. W dzisiejszych czasach coraz więcej informacji jest przetwarzanych i gromadzonych tylko i wyłącznie w postaci cyfrowej. W związku z tym coraz bardziej istotne jest rozwijanie efektywnych metod kompresji, które pozwalają na zaoszczędzenie zasobów (a więc i pieniędzy) potrzebnych na ich przechowywanie, gdzie efektywne należy rozumieć jako szybkie oraz umożliwiające istotne zmniejszenie rozmiaru. Z drugiej strony, chcemy nie tylko przechowywać dane, ale także co jakiś czas odpowiadać na dotyczące ich pytania. Oczywiście zawsze możemy w tym celu po prostu rozkompresować informacje, ale po włożeniu sporego wysiłku w kompresję wydaje się to marnotrawstwem. O wiele lepszym pomysłem jest próba przetworzenia danych w ich skompresowanej postaci. To, czy taki pomysł ma szanse na sukces, oczywiście zależy od konkretnej metody kompresji i od tego co rozumiemy przez przetwarzanie. W rozprawie skupiamy się na najbardziej naturalnym (a jednocześnie łatwym do sformułowania) sposobie przetwarzanie danych, którym jest wyszukiwanie wzorca, oraz metodach kompresji opartych na algorytmie Lempela-Ziva (używanych między innymi w formatach `zip`, `gzip`, `tiff` czy `png`). Poprawiamy większość z istniejących wyników dotyczących tego typu problemów.

(1) W rozdziale 5 pokazujemy, że dla tekstów skompresowanych metodą Lempela-Ziva-Welcha problem wyszukiwania wzorca może być rozwiązany w czasie liniowym, co zamyka problem po raz pierwszy rozważany przez Amira, Bensona i Faracha w roku 1994 [**4**].

(2) W rozdziale 6 pokazujemy, że jeżeli zarówno wzorzec jak i tekst są skompresowane metodą Lempela-Ziva-Welcha, problem wyszukiwania wzorca także może być rozwiązany w czasie liniowym.

(3) W rozdziale 7 podajemy dwa proste algorytmy wyszukiwania wielu wzorców w tekście skompresowanym metodą Lempela-Ziva-Welcha. Co prawda ich złożoności nie są liniowe, ale są one istotnie prostsze niż wcześniej znane rozwiązania dla jednego wzorca (o takim samym czasie działania).

(4) W rozdziale 8 pokazujemy, że dla tekstów skompresowanym metodą Lempela-Ziva problem wyszukiwania wzorca może być rozwiązany w (deterministycznym) czasie $\mathcal{O}(n \log \frac{N}{n} + m)$. Istotnie poprawia to wcześniej znane rozwiązanie podane przez Faracha i Thorupa [**17**], które działa w (oczekiwanym) czasie $\mathcal{O}(n \log^2 \frac{N}{n} + m)$.

Niektóre z pomysłów potrzebnych do uzyskania powyższych wyników mogą być użyte także w innych problemach. W szczególności, fragment algorytmu wyszukiwania wzorca

w tekście skompresowanym metodą Lempela-Ziva pozwala na podanie optymalnego rozwiązania dla problemu haszowania podsłów rozważanego przez Faracha i Muthukirshnana [**16**], a pomysł użyty w jednym z algorytmów wyszukiwania wielu wzorców może być wykorzystany do podania lepszego rozwiązania dla pewnego problemu związanego ze znajdowaniem funkcji najlepiej pasującej do danych typów argumentów, który był rozważany przez Alstrupa i innych [**2**].

Część z wyników uzyskanych w rozprawie została już opublikowana na międzynarodowych konferencjach. Liniowy algorytm wyszukiwania wzorca w tekście skompresowanym metodą Lempela-Ziva-Welcha był przedstawiony na konferencji SODA 2011 [**22**], a liniowo-logarytmiczny algorytm wyszukiwania wzorca w tekście skompresowanym metodą Lempela-Ziva na ESA 2011 [**23**].

Paweł Gawrychowski

Pattern Matching in Compressed Text

Ph.D. Thesis
Supervisor:
prof. Krzysztof Loryś, University of Wrocław

Institute of Computer Science
University of Wrocław
2011

Contents

CHAPTER 1

Introduction

With the recent explosion in the amount of the digital data we produce, developing good compression methods seems not only interesting but even necessary. Of course the processing capabilities of a modern hardware are growing very rapidly, but so is the amount of data we gather and store. Hence while inventing new compression methods has been a very active research area since decades, with more and more aspects of our lives being processed digitally the necessity of developing very efficient compression algorithms is becoming even more important. Efficient should be understood as both fast and achieving good compression ratio: any compression scheme should be evaluated taking both criteria into account as both the running time of compression/decompression and the space required cost us money.

While storing the data is an important task in itself, we do not want to keep it just for the sake of gathering more and more terabytes. We want to process the stored data efficiently on demand. Of course we can always simply uncompress the data and work with the uncompressed representation, but after putting a lot of effort into compressing it might seem like a waste of time (and money). This suggest a natural research direction: could we process the compressed representation without wasting time to uncompress it? Or, in other words, can we use the potentially high compression ratio to accelerate the computation? Answer to such question clearly depends on what to process mean and what is the compression method chosen. Probably the most natural and ubiquitous task concerning processing data is detecting an occurrence of a given pattern inside. Such *pattern matching problem* has been thoroughly studied for the case of uncompressed text, and many different optimal linear time solutions are known, starting with the first linear time solution of [**38**], then optimized for the delay between reading two consecutive letters of the text in [**20, 31**], the amount of additional memory used [**21**], and the total number of comparisons [**8**], to name just a few improvements, with the two most well-known algorithms being Knuth-Morris-Pratt [**31**] and Boyer-Moore [**7**]. When the text is compressed, we get the *compressed pattern matching problem*. Its complexity clearly depends on the particular compression scheme chosen. If we use any nonadaptive compression, the problem becomes rather simple. On the other hand, adaptive compression methods based on the Lempel-Ziv method [**48**] seems to be quite challenging to deal with efficiently, mostly because of the fact that the same fragment of a text might be encoded differently depending on its exact location. Moreover, in some variants of this method the compressed representation might be **exponentially** smaller than the original data, which suggest that achieving a complexity depending only (or mostly) on the former might be quite challenging. An even more challenging generalization is the *fully compressed pattern matching problem* where both the pattern

and the text are compressed. Assuming that the pattern is compressed as well makes the question substantially more difficult as most of the efficient pattern matching algorithms are based on (more or less computationally expensive) preprocessing suffixes or prefixes of the pattern, and if it is compressed we do not have enough time to look at all of them explicitly.

In this thesis we investigate the complexity of compressed pattern matching for Lempel-Ziv family of compression methods. More specifically, we focus on Lempel-Ziv [**48**] and Lempel-Ziv-Welch [**46**] compression. We also consider a generalization in which multiple pattern are given, and we aim to detect an occurrence of any of them. We are mostly concerned with the non-compressed pattern variants as they seem to be more justified from the more practical point of view. Nevertheless, the fully compressed variant seems fascinating from a more theoretical angle, and we also consider such generalization. It turns out that we are able to improve most of the previously known results in this area. Furthermore, some of our solutions are optimal, and some of our ideas can be used to give better algorithms for other (in one case, seemingly unrelated) problems.

In the next chapter we briefly review the history of the (fully) compressed pattern matching and the previously known results.

CHAPTER 2

Previous work and our contribution

The compressed pattern matching problem has been studied quite intensively. Recall that in this problem given a text $t1[..N]$ and a pattern $p[1..m]$ we need to check if there is an occurrence of p in t without decompressing t, i.e., the running time should depend on the size n of the compressed representation of t alone rather than on the original length N. According to Amir and Benson [**3**], a solution to such problem is *efficient* if its time complexity is $o(N)$, *almost optimal* if its time complexity is $\mathcal{O}(n \log m + m)$, and *optimal* if the complexity is $\mathcal{O}(n + m)$.

There has been a substantial amount of work devoted to developing fast algorithms for pattern matching in both Lempel-Ziv and its simplified version Lempel-Ziv-Welch [**46**] compressed texts. In [**4**] Amir, Benson and Farach introduced two algorithms with time complexities $\mathcal{O}(n + m^2)$ and $\mathcal{O}(n \log m + m)$ for Lempel-Ziv-Welch compressed texts. The pattern preprocessing time was soon improved by Kosaraju to get $\mathcal{O}(n + m^{1+\epsilon})$ time complexity [**34**]. Then Farach and Thorup considered the general Lempel-Ziv case and developed an algorithm working in (randomized) time $\mathcal{O}(n \log^2 \frac{N}{n} + m)$ [**17**], with a version simplified for Lempel-Ziv-Welch working in (also randomized) time $\mathcal{O}(n \log \frac{N}{n} + m)$. This seems rather surprising as in the general case the compressed representation might be exponentially more succinct than the original data, i.e., n might be of order $\mathcal{O}(\log N)$, while in Lempel-Ziv-Welch compression n is always $\Omega(\sqrt{N})$. The solution of Farach and Thorup consists of two phases, called *winding* and *unwinding*, the first one uses a cleverly chosen potential function, and the second one adds fingerprinting in the spirit of string hashing first developed by Karp and Rabin [**28**]. While a recent result of [**26**] shows that the winding can be performed in just $\mathcal{O}(n \log \frac{N}{n})$ time, it is not clear how to use that to improve the whole complexity (or remove randomization). Besides the algorithm of Farach and Throup, the only other result that can be applied to the general case we are aware of is the work of Kida *et al.* [**30**]. They considered the so-called *collage systems* allowing to capture many existing compression schemes (but in fact not more expressive than the so-called self-referential Lempel-Ziv scheme, which will be defined later), and developed an efficient pattern matching algorithm for them. While it does not apply directly to the Lempel-Ziv compression, we can transform a Lempel-Ziv parse into a non-truncating collage system with a slight increase in the size, see Chapter 8. The running time (and space usage) of the resulting algorithm is $\mathcal{O}(n \log \frac{N}{n} + m^2)$. While m^2 might be acceptable from a practical point of view, removing the quadratic dependency on the pattern length seems to be a nontrivial and fascinating challenge from a more theoretical angle, especially given that for some highly compressible texts n might be much smaller than m. Citing [**30**], even decreasing

the dependency to $m^{1.5} \log m$ (the best preprocessing complexity known for the LZW case [**34**] at the time) "is a challenging problem".

It is also worth noting than Navarro and Raffinot developed a practical $\mathcal{O}(n\frac{m}{w})$ algorithm exploiting bit-parallelism [**39**].

There has been also a substantial amount of research devoted to a more general problem of *fully compressed pattern matching*, where both the text and the pattern are compressed. For the case of Lempel-Ziv-Welch compression, Gąsieniec and Rytter [**25**] developed a $\mathcal{O}((n+m)\log(n+m))$ time algorithm, where n and m are the compressed sizes of the text and the pattern, respectively. If we consider the more general Lempel-Ziv version, it is not obvious if we can solve the problem in polynomial time as the compression ratio might be exponential. Nevertheless, it is possible, as first shown by Gąsieniec *et al.* [**24**]. More efficient solutions are possible using the concept of *straight-line programs*, which by a result of Rytter [**42**] can be used to efficiently capture the Lempel-Ziv compression (with just a logarithmic increase in size) but are substantially more convenient to work with. As shown by Karpiński *et al.* [**29**], fully compressed pattern matching for such programs can be solved in $\mathcal{O}((n+m)^4 \log(n+m))$ time. A faster $\mathcal{O}(n^2m^2)$ time solution has been later given by Miyazaki *et al.* [**37**], and Lifshits decreased it even further to $\mathcal{O}(n^2m)$ [**35**]. We will be interested only in the Lempel-Ziv-Welch case, though.

A natural and interesting generalization of the above compressed pattern matching problem is to consider *multiple compressed pattern matching*, where instead of just one pattern we are given their collection $p_1, p_2, \ldots, p_k$ (which, for example, can be a set of forbidden words from a dictionary). It is known that extending one of the algorithms given by Amir *et al.* results in a $\mathcal{O}(n + M^2)$ running time for such general case, where $M = \sum_i |p_i|$ [**30**]. It seems realistic that the set of the patterns is very large, and hence M^2 addend in the running time might be substantially larger than n.

The following table summarizes the previously known results.

LZW	$\mathcal{O}(n + m^2)$	Amir, Benson, Farach SODA 1994 [**4**]
	$\mathcal{O}(n \log m)$	
	$\mathcal{O}(n + m^{1+\epsilon})$	Kosaraju FSTTCS 1995 [**34**]
	$\mathcal{O}((n+m)\log(n+m))$	fully compressed, Gąsieniec and Rytter DCC 1999 [**25**]
	$\mathcal{O}(n + M^2)$	multiple patterns, Kida *et al.* DCC 1998 [**30**]
	$\mathcal{O}(n \log \frac{N}{n} + m)$	expected time, Farach and Thorup STOC 1995 [**17**]
LZ	$\mathcal{O}(n \log^2 \frac{N}{n} + m)$	

In this thesis we improve all of the above time bounds. More specifically, we get the following results.

(1) In Chapter 5 we show that compressed pattern matching can be solved in optimal linear time for Lempel-Ziv-Welch compressed texts, assuming that the alphabet consists of integers which can be sorted in linear time. This closes the line of research started by Amir, Benson and Farach in 1994.
(2) In Chapter 6 we show that fully compressed pattern matching can be solved in optimal linear time for Lempel-Ziv-Welch compressed texts, again assuming that the alphabet consists of integers which can be sorted in linear time. This might be seen as a little bit surprising as compressing both the pattern and the

text seems to create a significantly more complex situation than compressing the text alone. Nevertheless, by building upon the linear time algorithm from Chapter 5 we are able to develop an optimal linear time solution for the fully compressed case as well.

(3) In Chapter 7 we show that multiple compressed pattern matching can be solved in $\mathcal{O}(n \log M + M)$ or $\mathcal{O}(n + M^{1+\epsilon})$ time for Lempel-Ziv-Welch compressed texts. While the running time is not linear, both algorithms are very simple. In particular, the latter is significantly simpler that the single pattern solution of Kosaraju [**34**]. We base both solutions on reducing the original question to purely geometric problems which seem to be particularly enjoyable to work with. One of the ideas we use to tackle those geometric problems can be used to give an improved (and simpler) solution for the seemingly unrelated *bridge color* problem which we show in Appendix A.

(4) In Chapter 8 we show that compressed pattern matching can be solved in (deterministic) $\mathcal{O}(n \log \frac{N}{n} + m)$ time for Lempel-Ziv compressed texts, assuming that the alphabet consists of integers which can be sorted in linear time and our computational model allows integer division of numbers consisting of $\log N$ bits. If such division is not possible, our running time increases to $\mathcal{O}(n \log N + m)$ and we give a matching lower bound. While a logarithmic improvement in the running time might seems marginal, one should note that for highly LZ compressible texts such factor can actually be of order n, and hence for such inputs our improvement is very substantial. Moreover, the previously known solution required randomization, which seems to be a very powerful paradigm in this context, while ours is fully deterministic.

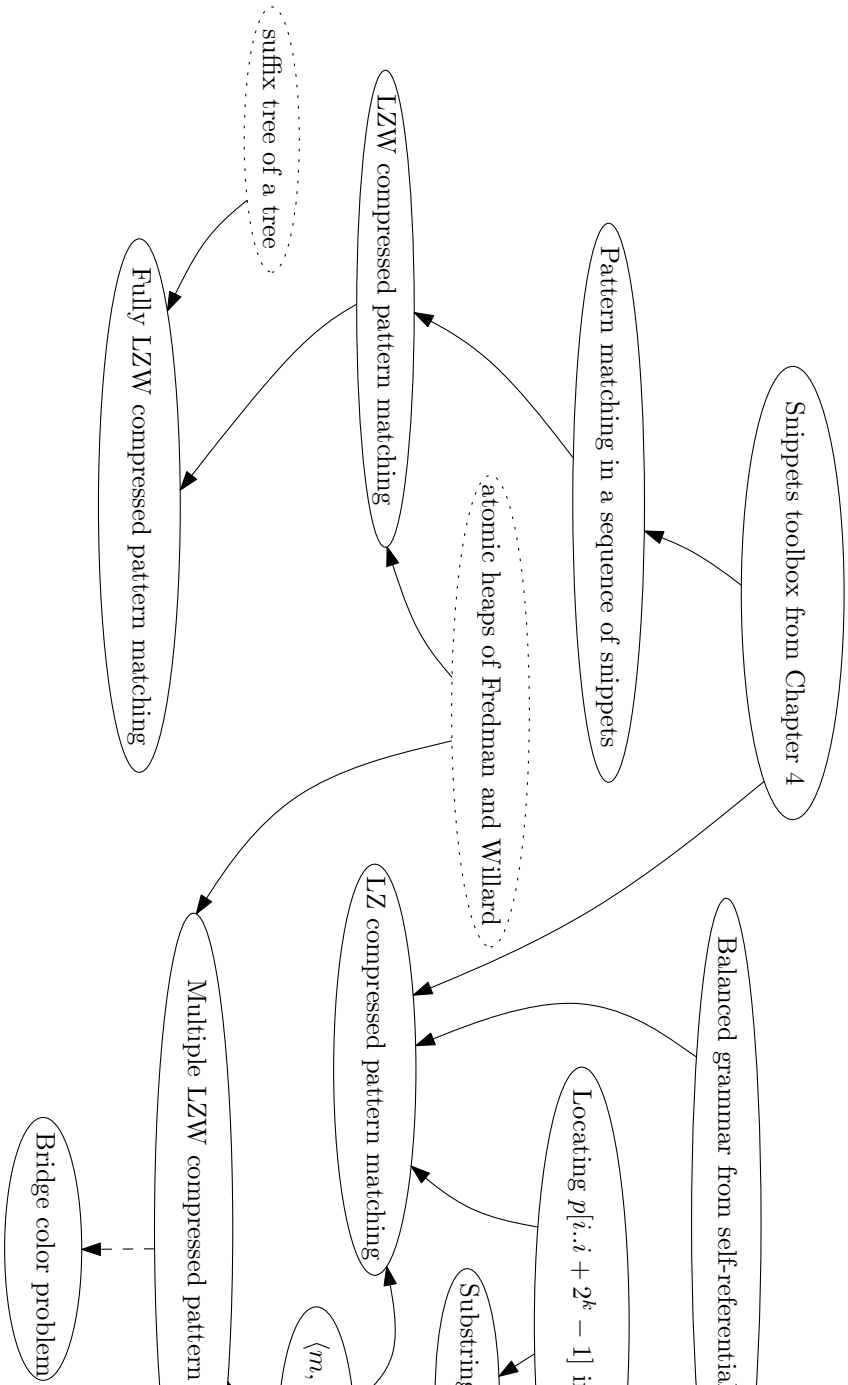
The following table summarizes our results.

LZW	$\mathcal{O}(n + m)$	
	$\mathcal{O}(n + m)$	fully compressed
	$\mathcal{O}(n + M^{1+\epsilon})$	multiple patterns
	$\mathcal{O}(n \log M + M)$	
LZ	$\mathcal{O}(n \log \frac{N}{n} + m)$	

The method we use to achieve the above time bounds can be applied to other problems. In particular, fragment of our $\mathcal{O}(n \log \frac{N}{n} + m)$ time solution for Lempel-Ziv compressed texts can be used to given an optimal algorithm for a slight relaxation of the *substring hashing* problem considered by Farach and Muthukrishnan [**16**]. Furthermore, ideas from our improved multiple compressed pattern matching solution can be (surprisingly) applied to the *bridge color problem* considered by Alstrup *et al.* [**2**], who improved on the previous work by Ferragina *et al.* [**18**]. As a result we get a simpler algorithm with the same query time and space requirements, but just linear preprocessing cost. We believe that some of our methods can be successfully applied to other problems as well.

The following diagram visualizes the contents of the thesis and connections between its parts.

Some of the results contained in the thesis already appeared in well-known international conferences. The linear time LZW-compressed pattern matching algorithm described in Chapter 5 was published in SODA 2011 [**22**], and $\mathcal{O}(n \log \frac{N}{n} + m)$ time LZ-compressed pattern matching algorithm described in Chapter 5 was accepted to ESA 2011 [**23**].


Snippets toolbox from Chapter 4
Pattern matching in a sequence of snippets
LZW compressed pattern matching
suffix tree of a tree
Fully LZW compressed pattern matching
atomic heaps of Fredman and Willard
Balanced grammar from self-referential LZ parse
Locating $p[i..i+2^k-1]$ in the suffix tree
Substrings hashing
LZ compressed pattern matching
$\langle m, \log m \rangle$ locator
Multiple LZW compressed pattern matching
Bridge color problem

CHAPTER 3

Preliminaries

Let $\Sigma = \{a_1, a_2, a_3, \ldots\}$ be an ordered alphabet. We will consider two cases: either Σ is fixed and finite (for example, $\Sigma = \{0, 1\}$) or consists of integers which can be sorted in linear time. All of our algorithms work in the same complexity for both cases, but ensuring that the complexity does not increase when the alphabet is non-constant will require some additional nontrivial ideas.

We will consider strings over Σ given in a Lempel-Ziv compressed form. A string in such form is represented as a sequence of blocks, each block being either a single character or any substring of the already defined prefix. More precisely, the Lempel-Ziv representation of a string $t[1\,..\,N]$ is a sequence of triples $(start_i, len_i, next_i)$ for $i = 1, 2, \ldots, n$, where n is the size of the representation, $start_i$ and len_i are nonnegative integers, and $next_i \in \Sigma$. Such triple refers to a fragment of the text $t[start_i\,..\,start_i + len_i - 1]$ and defines $t[1 + \sum_{j<i} len_j\,..\,\sum_{j\le i} len_j] = t[start_i\,..\,start_i + len_i - 1]next_i$. We require that $start_i = 0$ iff $len_i = 0$ and $start_i \le \sum_{j<i} len_j$ if $len_i > 0$. The representation is not self-referential if all fragments we are referring to are already defined, i.e., $start_i + len_i - 1 \le \sum_{j<i} len_j$ for all i. In the self-referential case we allow descriptions like $[(0, 0, a), (1, \ell, b)]$ which defines $a^{\ell+1}b$. In the non self-referential case such word requires a description of size $\Theta(\log \ell)$. The sequence of triples is often called the *LZ parse* of text. While it is known how to compute this parse in linear time [**41**], from the purely practical point of view this linear complexity is not completely satisfactory due to the high constants involved. Hence it makes sense to consider also slightly simpler compression schemes. The most well-known of such schemes is the Lempel-Ziv-Welch representation, where we split the string into a sequence of *codewords*, called the *LZW parse*, with each codeword being either a single character, or a previously defined codeword concatenated with a single character. The resulting compression method enjoys a particularly simple encoding/decoding process, but unfortunately requires outputting at least $\Omega(\sqrt{N})$ codewords. Still, its simplicity and good compression ratio achieved on real life instances make it an interesting model to work with.

In Chapters 5 and 8 we are interested in a variation of the classical pattern matching problem: given a pattern $p[1\,..\,m]$ and a text $t[1\,..\,N]$, does p occur in t? In our case t is given in a compressed form. We wish to achieve a running time depending on the size n of this compressed representation, not the length of t itself. Additionally, we would like to keep the memory complexity as low as $\mathcal{O}(m)$, preferably with the algorithm reading the compressed representation of t just once from left to right. If the pattern does occur in the text, we would like to get the position of its first occurrence. Later in Chapter 7 we will consider a multiple patterns version where we are given a collection of k patterns $p_1[1\,..\,m_1], p_2[1\,..\,m_2], \ldots, p_k[1\,..\,m_k]$ of total length $M = \sum_i m_i$ and should detect if any

of those patterns occurs in a compressed text $t[1\,..\,N]$. Another variation we will consider in Chapter 6 is the fully compressed version where we are given a pattern $p[1\,..\,M]$ and a text $t[1\,..\,N]$, both in compressed forms of size n and m, respectively, and are required to check if p occurs in t. While the meaning of m and M changes between chapters, we believe it will be clear from the context.

The computational model we are going to use is the standard RAM allowing direct and indirect addressing, addition, subtraction, integer division and conditional jump with word size $w \geq \max\{\log m, \log n, \log N\}$. One usually allows multiplication as well in this model but we do not need it, and the only place where we use integer division (which in some cases is known to significantly increase the computational power), is the proof of Lemma 8.1. We do not assume that any other operation (like, for example, taking logarithms) can be performed in constant time on arbitrary words of size w. Nevertheless, because of the n addend in the final running time, we can afford to preprocess the results on words of size $\log n$ and hence assume that some additional (reasonable) operations can be performed in constant time on such inputs. By such assumption we can implement a very efficient dictionary storing a bounded number of elements.

Lemma 3.1 (atomic heaps of Fredman and Willard [**19**])**.** *It is possible to maintain a collection of sets $S(i) \subseteq \{0, 2, \ldots, n-1\}$ so that inserting, removing and finding successor in each of those sets work in constant time (amortized for insert and remove, worst case for find) as long as $|S(i)| \leq \log^c n$ for all i, assuming a $\mathcal{O}(n)$ time and space preprocessing, where c is any (but fixed) constant. Finding successor can be improved to work in worst case constant time.*

As usually, $|w|$ stands for the length of w, $w[i\,..\,j]$ refers to its fragment of length $j - i + 1$ beginning at the i-th character, where characters are numbered starting from 1, and w^r is the reversed w. A *border* of a string $w[1\,..\,|w|]$ is a proper fragment which is both a prefix and a suffix of w, i.e., $w[1\,..\,i] = w[|w| - i + 1\,..\,|w|]$ with $i < |w|$. We identify such fragment with its length and say that $\mathrm{border}(w) = \{i_1, \ldots, i_k\}$ is the set of all borders of w. Whenever we say *the border* of a word, we mean its longest border. A *period* of a string $w[1\,..\,|w|]$ is an integer d such that $w[i] = w[i+d]$ for all $1 \leq i \leq |w|-d$. We call w aperiodic if its period is at least $\frac{|w|}{2}$, and periodic otherwise. Note that d is a period of w iff $|w| - d$ is a border. Whenever we say *the period* of a word, we mean its shortest period. The following lemma is a well-known property of periods.

Lemma 3.2 (Periodicity lemma)**.** *If both d and d' are periods of w, and $d + d' \leq |w| + \gcd(d, d')$, then $\gcd(d, d')$ is a period as well.*

At a very high level some of our procedures resemble the old and well-known Knuth-Morris-Pratt pattern matching algorithm: we process the prefixes of the text one-by-one and for each of them compute the longest prefix of p which is a suffix of the current prefix. Recall that if we already have such longest prefix of a given $t[1..i]$ and need to compute the longest match for $t[1..i+1]$, there are two possibilities. Either we extend the current longest prefix by $t[i+1]$, or we can safely replace it with the longest border. A simple amortized argument shows that even though we might have to consider a lot of different borders before we are able to extend the prefix, the total complexity of this algorithm is linear. Nevertheless, in some extensions of this algorithm we need to avoid performing the computation for every possible border of a word, and it turns out that

this is possible due to Lemma 3.2. More specifically, instead of processing them one-by-one we can split them into $\log |w|$ groups with all borders in a single group creating one arithmetic progression, and hope to process them all at once due to the following lemma.

Lemma 3.3. *If the longest border of w is of length $b \geq \frac{|w|}{2}$ then all borders of length at least $\frac{|w|}{2}$ create one arithmetic progression. More specifically,* $\text{border}(w) \cap \left\{\frac{|w|}{2}, \ldots, |w|\right\} = \left\{|w| - \alpha d : 1 \leq \alpha \leq \frac{|w|}{2d}\right\}$*, where $d = |w| - b$ is the period of w. We call this set the long borders of w.*

Proof. First note that $d = |w| - b$ is the period of $|w|$ so any αd is also a period, as long as $\alpha \leq \frac{|w|}{d}$. Thus all elements of the arithmetic progression in the lemma are borders. We must show that there are no other borders of length at least $\frac{|w|}{2}$. Let b' be any such border. Observe that $d' = |w| - b'$ is a period and $d + d' \leq |w|$ so from the periodicity lemma $\gcd(d, d')$ is a period as well. But d is the shortest period so d divides d' and b' belongs to the progression. □

We preprocess the pattern p using standard tools (suffix trees [**45**] built for both the pattern and the reversed pattern, and lowest common ancestor queries [**5**]) to get the following primitives.

Lemma 3.4. *Pattern p can be preprocessed in linear time so that given i, j, k representing any two fragments $p[i\,..\,i+k]$ and $p[j\,..\,j+k]$ we can find their longest common prefix (suffix) in constant time.*

While there are a few different proofs of the above lemma known in the literature, the solution of [**5**] uses the concept of a *range minimum query* structure. We will extensively use such structures in Chapter 7 and hence state the following lemma separately.

Lemma 3.5. *Given an integer array $a[1\,..\,n]$ we can build in linear time and space a range minimum query structure* $\text{RMQ}(a)$ *which allows computing the maximum $a[k]$ over all $k \in \{i, i+1, \ldots, j\}$ for a given i, j in constant time.*

By applying the above lemma one can develop a very efficient algorithm for finding the lowest common ancestor of any two vertices of a tree [**5**].

Lemma 3.6. *Given a tree on n vertices we can build in linear time and space a structure which allows computing the lowest common ancestor* $\text{LCA}(u, v)$ *of any two vertices u, v in constant time.*

A direct application of the above lemma to the suffix tree gives Lemma 3.4. Another similar concept which we are going to extensively use are *level ancestor queries*. We want to preprocess tree so that given any vertex v and an integer k we can efficiently retrieve its k-th ancestor. It turns out that such queries can be answered in constant time after a linear preprocessing [**6**].

Lemma 3.7. *Given a tree on n vertices we can build in linear time and space a structure which allows computing the k-th ancestor of any vertex v in constant time.*

Using the suffix trees we can extract the longest suffix of a given fragment which is a prefix of the whole pattern efficiently.

Lemma 3.8. *Pattern p can be preprocessed in linear time so that given any fragment $p[i\,..\,j]$ we can find its longest prefix (suffix) which is a suffix (prefix) of the whole pattern in constant time, assuming we know the (explicit or implicit) vertex corresponding to $p[i\,..\,j]$ in the suffix tree built for p (reversed p).*

PROOF. We assume that the suffix tree is built for p concatenated with a special terminating character, say \$. Because of this special additional character, each suffix of p corresponds to some leaf of the suffix tree. For each leaf we consider the edge connecting it to the parent. If the label of this edge consists of just single letter, we mark the parent. Then finding the longest prefix which is a suffix of the whole p reduces to finding the lowest marked vertex on a given path leading the root, which can be precomputed for all vertices in linear time. □

We will also require a quick access to the borders of both prefixes and suffixes of p.

Lemma 3.9. *Pattern p can be preprocessed in linear time so that we can find the border of each its prefix (suffix) in constant time.*

PROOF. Simply do the standard preprocessing from the Knuth-Morris-Pratt algorithm for both p and the reversed p. This preprocessing results in computing the border of each possible prefix. □

We will use the suffix array SA built for p [**27**]. For each suffix of p we store its position inside SA, and treat the array as a sequence of strings rather than a permutation of $\{1, 2, \ldots, |p|\}$. Given any word w, we will say that it occurs at position i in the SA if w begins $p[SA[i]\,..\,|p|]$. Similarly, the fragment of SA corresponding to w is the (maximal) range of entries at which w occurs.

To develop an efficient pattern matching algorithm for LZ compressed texts in Chapter 8 we will need the concept of *straight-line programs*. Such program is a context-free grammar in the Chomsky normal form which nonterminals $X_1, X_2, \ldots, X_s$ can be ordered in such a way that each X_i occurs exactly once as a left side, and whenever $X_i \to X_j X_k$ it holds that $j, k < i$. Such grammar derives exactly one string, and we identify each nonterminal with the unique string it derives, so $|X|$ stands for the length of the string derived from X. We call a straight-line program (SLP) *balanced* if for each production $X \to YZ$ both $|Y|$ and $|Z|$ are bounded by a constant fraction of $|X|$.

CHAPTER 4

Snippets toolbox

In this section we develop a few efficient procedures operating on fragments of the pattern, which we call *snippets*:

Definition 4.1. A snippet is any substring $p[i\,..\,j]$ of the pattern p. If $i = 1$ we call it a prefix snippet, if $j = m$ a suffix snippet. A sequence of snippets of size k is a concatenation of k substrings of the pattern $p[i_1\,..\,j_1]\ldots p[i_k\,..\,j_k]$.

One could argue that snippets are simply substrings of the pattern and hence the name is redundant. We believe that repeating "substring of the pattern" multiple times makes the proofs more difficult to follow and prefer to call them snippet for the sake of brevity.

We identify snippets with the substrings they represent, and use $|s|$ to denote the length of the string represented by a snippet s. A snippet is stored as a pair of integers (i, j). Sometimes we will also require that a link to the corresponding node (either explicit or implicit) in the suffix tree and the longest suffix which is a prefix of the whole pattern (in case of a prefix snippet this is of course the whole fragment) are available. If this information is available, we call the snippet *extended*.

In this section we focus on developing a few procedures allowing efficient processing of pairs of snippets. Using these procedures as basic building blocks, we will later construct efficient algorithms for pattern matching in LZW and LZ compressed strings. For the former, we will try to simulate the Knuth-Morris-Pratt algorithm operating on whole snippets instead of single characters. For the latter, the idea is slightly more involved, and we defer its description till the appropriate section. We assume that the pattern has been preprocessed using Lemma 3.4 and Lemma 3.9.

The first result shows how to detect an occurrence of the pattern in a concatenation of two snippets. We will perform a lot of such operations, and a constant time implementation is crucial.

Lemma 4.1. *Given a prefix snippet and a suffix snippet we can detect an occurrence of the pattern in their concatenation in constant time.*

Proof. We need to answer the following question: does p occur in $p[1\,..\,i]p[j\,..\,m]$? Or, in other words, is there $x \in \text{border}(p[1\,..\,i])$ and $y \in \text{border}(p[j\,..\,m])$ such that $x + y = m$? Note that either $x \geq \frac{|p[1..i]|}{2}$ or $y \geq \frac{|p[j..m]|}{2}$, and without losing the generality assume the former. From Lemma 3.3 we know that all such possible values of x create one arithmetic progression. More specifically, $x = i - \alpha d$, where $d \leq \frac{i}{2}$ is the period of $p[1\,..\,i]$ extracted using Lemma 3.9. We need to check if there is an occurrence of p in $p[1\,..\,i]p[j\,..\,m]$ starting after the αd-th character, for some $0 \leq \alpha \leq \frac{i}{d}$. For any

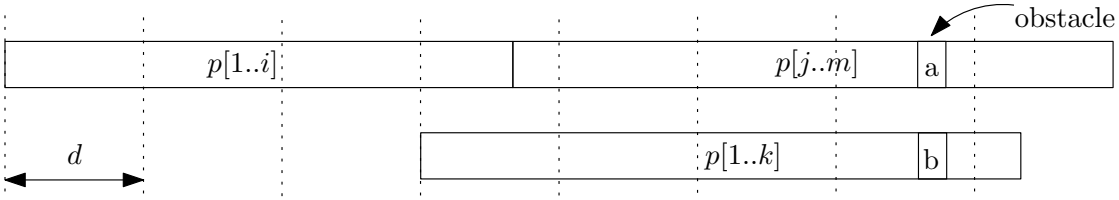


FIGURE 1. Detecting an occurrence in a concatenation of two snippets.

such possible interesting shift, there will be no mismatch in $p[1\,..\,i]$. There might be a mismatch in $p[j\,..\,m]$, though.

Let $p[1\,..\,k]$ be the longest prefix of p for which d is a period (so $k \geq i$). Such k can be calculated efficiently by looking up the longest common prefix of $p[d+1\,..\,m]$ and the whole p. We shift $p[1\,..\,k]$ by $\left\lfloor \frac{\min(i, i+|p[j..m]|-m)}{d} \right\rfloor d$ characters. Note this is the maximal shift of the form αd which, after extending $p[1\,..\,k]$ to the whole p, does not result in sticking out of the right end of $p[j\,..\,m]$. Then compute the leftmost mismatch of the shifted $p[1\,..\,k]$ with $p[j\,..\,m]$, see Figure 1. Position of the first mismatch, or its nonexistence, allows us to eliminate all but one interesting shift. More precisely, we have two cases to consider.

(1) There is no mismatch. If $k = m$ we are done, otherwise $p[k+1] \neq p[k+1-d]$, meaning that choosing any smaller interesting shift results in a mismatch.
(2) There is a mismatch. Let the conflicting characters be a and b and call the position at which a occurs in the concatenation the *obstacle*. Observe that we must choose a shift αd so that $p[1\,..\,k]$ shifted by αd is completely on the left of the obstacle. On the other hand, if $p[1\,..\,k]$ shifted by $(\alpha+1)d$ is completely on the left as well, shifting $p[1\,..\,k]$ by αd results in a mismatch because $p[k+1] \neq p[k+1-d]$ and $p[k+1-d]$ matches with the corresponding character in $p[j\,..\,m]$. Thus we may restrict our attention to the largest shift for which $p[1\,..\,k]$ is on the left of the obstacle.

Having identified the only interesting shift, we verify if there is a match using one longest common prefix query on p. More precisely, if the shift is αd, we check if the common prefix of $p[i-\alpha d\,..\,m]$ and $p[j\,..\,m]$ is of length $|p[i-\alpha d\,..\,m]|$. Overall, the whole procedure takes constant time. □

The remaining part of this section is devoted to showing how to compute the longest prefix of the pattern which is a suffix of a concatenation of two snippets. Unfortunately, a constant time implementation seems rather unlikely here. In the above proof we had the nice observation that an occurrence corresponds to a long border of either the left or the right part, and here we do not know that. We start with a technical lemma.

Lemma 4.2. *Given a prefix snippet $p[1\,..\,i]$ and a snippet $p[j\,..\,k]$ we can find the longest long border b of $p[1\,..\,i]$ such that $p[1\,..\,b]p[j\,..\,k]$ is a prefix of the whole p in constant time.*

PROOF. As in the previous proof, we begin with computing $p[1\,..\,i']$, the longest prefix of p for which $d \leq \frac{i}{2}$ is a period, where d is the period of $p[1\,..\,i]$ (so $i' \geq i$).

We need to choose the smallest α so that shifting p by αd characters results in no mismatches with $p[1\,..\,i]p[j\,..\,k]$, and the shifted p does not end before the right end of $p[1\,..\,i]p[j\,..\,k]$. We begin with shifting $p[1\,..\,i']$ by $\left\lceil \frac{i+|p[j..k]|-m}{d} \right\rceil d$ characters, which is the smallest possible shift, and compute its leftmost mismatch with $p[1\,..\,i]p[j\,..\,k]$, which must be on the right of $p[1\,..\,i]$. We have two cases to consider.

(1) There is no mismatch. If $i' = m$ or $p[i'+1]$ is on the right of $p[j\,..\,k]$, we have the longest long border. Otherwise compute $p[j\,..\,j']$, the longest prefix of $p[j\,..\,k]$ such that d is a period of $p[1\,..\,i]p[j\,..\,j']$. if $j' = k$, there is no such long border. Otherwise we must align $p[1\,..\,i']$ so that $p[i'+1]$ corresponds to $p[j'+1]$. Any other shift results in aligning either $p[i'+1]$ or $p[j'+1]$ with a character continuing the period. Hence we either get that there is no such long border or there is exactly one candidate which can be verified in constant time using one longest common prefix query.
(2) There is a mismatch. Again, call the position in $p[1\,..\,i]p[j\,..\,k]$ at which the mismatch occurs the *obstacle*. Even if we choose a larger shift, $p[1\,..\,i']$ will expect the same character at the obstacle, so we will get a mismatch as well. Thus there is no such long border.

□

By iterating the above lemma we get the following logarithmic complexity. Unfortunately, it is too high to serve as a basis of an optimal linear time algorithm, and we will need additional (and slightly different) ideas. Nevertheless, we state the lemma as a warm-up exercise.

Lemma 4.3. *Given a prefix snippet $p[1\,..\,i]$ and a snippet $p[j\,..\,k]$ we can find the longest border b of $p[1\,..\,i]$ such that $p[1\,..\,b]p[j\,..\,k]$ is a prefix of the whole p in $\mathcal{O}(\log m)$ time.*

Proof. First we apply Lemma 4.2 to check if there is a long border of $p[1\,..\,i]$ which can be extended to get a prefix of p which is a suffix of $p[1\,..\,b]p[j\,..\,k]$. If there is, we are done. Otherwise we would like to replace $p[1\,..\,i]$ with its longest non-long border. Recall that by Lemma 3.9 we can retrieve the longest border of any prefix of p in constant time. Let $p[1\,..\,b]$ be the border of $p[1\,..\,i]$. If $b < \frac{i}{2}$, we have the longest non-long border. Otherwise $d = i - b$ is the period of $p[1\,..\,i]$ and any $i - \alpha d$ is a border as well. We select α to be the largest integer such that $\alpha d < \frac{i}{2}$, i.e., $\alpha = \left\lfloor \frac{i-1}{2d} \right\rfloor$. Observe that $p[1\,..\,i - \alpha d]$ is the shortest long border of $p[1\,..\,i]$. Hence retrieving its longest border gives us the longest non-long border of the whole $p[1\,..\,i]$ in constant time. By iterating this procedure we get the longest border which can be extended by $p[j\,..\,k]$. Each step decreases the current value of i at least twice, hence the complexity is $\mathcal{O}(\log m)$. □

The last result in this section will be heavily used in Chapter 8. It might be seen as an extension of the above lemma with a better running time for some specific cases. While it could be also deduced from the linear time LZW pattern matching algorithm presented in Chapter 5, we prefer to present an explicit (and simpler) separate proof for the sake of completeness.

Lemma 4.4. *Given a prefix snippet s_1 and a snippet s_2 for which we know the corresponding node in the suffix tree, we can compute the longest prefix of p which is a suffix*

of $s_1 s_2$ in time $\mathcal{O}\left(\max\left(1, \log \frac{|s_1|}{|s_2|}\right)\right)$, assuming a linear time and space preprocessing of the suffix tree.

PROOF. We try to find the longest border of $s_1 = p[1..i]$ which can be extended with s_2. If there is none, we use Lemma 3.8 on s_2 to extract the answer. Of course s_1 might happen to have quite a lot of borders, and we do not have enough time to go through each of them separately. We try to apply Lemma 3.3 instead: there are just $\log |s_1|$ groups of borders, and we are going to process each of them in constant time. It is not enough though, we need something faster when $|s_2|$ is relatively big compared to $|s_1|$. The whole method works as follows: as long as $|s_2|$ is smaller than $2|s_1|$, we check if it is possible to extend any of the long borders of s_1. If it is not possible, we replace s_1 with its longest non-long border computed in constant time as shown in the proof of Lemma 4.3. When $|s_2|$ exceeds $2|s_1|$, we look for an occurrence of s_2 in a prefix of p of length $|s_1| + |s_2|$. All such occurrences create one arithmetic progression due to Lemma 3.3, and it is possible to detect which one is preceded by a suffix of s_1 in constant time. More specifically, we show how to implement in constant time the following two primitives. In both cases the method resembles the one from Lemma 4.1.

(1) Computing the longest long border of s_1 which can be extended with s_2 to form a prefix of p, if any. We use Lemma 4.2.
(2) Detecting the rightmost occurrence of s_2 in p preceded by a suffix of s_1, assuming $|s_2| \geq 2|s_1|$. We begin with finding the first and the second occurrence of s_2 in p. Assuming we have the corresponding vertex in the suffix tree available, this takes just constant time after a linear preprocessing. We check those (at most) two occurrences naively. There might be many more of them, though. But if the two first occurrences begin before the $|s_1|$-th character, we know that all other interesting occurrences form one arithmetic progression. Furthermore, the difference between those two first occurrences is equal to the period of s_2 (which is important as we do not have an efficient way of computing the period of any substring of the pattern). We check how far the period extends to the left starting from the right end of s_1 and the first occurrence of s_2 in the whole p. This information gives us a simple arithmetic condition on the possible shift by applying almost the same reasoning as the one from Lemma 4.2.

Each application of the first primitive allows us to decrease $|s_1|$ at least twice, hence after at most $\log \frac{|s_1|}{|s_2|}$ iterations we can apply the second primitive and retrieve the answer. □

Note that the running time from the above lemma stays constant as long as $|s_1|$ is bounded from above by a constant fraction of $|s_2|$.

CHAPTER 5

Lempel-Ziv-Welch compressed pattern matching

1. Overview of the algorithm

Our goal is to detect an occurrence of p in a given Lempel-Ziv-Welch compressed text $t[1\,..\,N]$. At a very high level our idea is to simulate the Knuth-Morris-Pratt algorithm on t. Clearly we cannot afford to process the characters one-by-one and hence try to skip whole codewords. It turns out that working with the codewords directly is somehow rather inconvenient, as they do not have to be anyhow similar to the pattern and hence we cannot hope to preprocess any information which would help us to process whole fragments in constant time. Hence we start with reducing the original problem into a more uniform variant, where we are given a pattern and a collection of strings constructed by concatenating a number of substrings of the pattern one after another. In the next section we show that such problem, which we call pattern matching in a sequence of snippets, can be solved in optimal linear time. Then we observe that pattern matching in LZW compressed text reduces in linear time to such restricted problem. If the alphabet is of constant size, the reduction is rather straightforward. If it does not, we need some additional ideas to keep the running time linear.

2. Pattern matching in a sequence of snippets

In this section we develop an optimal linear time algorithm which detects an occurrence of the pattern in a string constructed by concatenating a number of extended snippets. Given such sequence, we would like to simulate the Knuth-Morris-Pratt algorithm efficiently. To this aim we need Lemma 4.1 and Lemma 4.2 presented in Chapter 4. Using those two procedures as basic building blocks, we can present the first algorithm Naive-pattern-matching. It simulates the Knuth-Morris-Pratt algorithm, adding just two optimizations: we extend the current match by whole snippets, not single letters, and process all long borders of the current match at once. This is not enough to achieve a linear complexity yet: it might happen that we need to spend $\Omega(\log m)$ time at each snippet. Nevertheless, such method will serve as a basis for developing the optimal solution.

While we assume that for all input snippets the information about corresponding vertex in the suffix tree and longest suffix being a prefix of p is already known, during the execution we might create some new snippets. Fortunately, they are either prefix snippets, or snippets of the form $p[\lceil \frac{\ell}{2} \rceil\,..\,\ell]$ which will be called the *half snippets*. Before we analyze the running time of Naive-pattern-matching we need a small technical lemma concerning those fresh snippets. Note that it would be possible to modify the algorithm a little bit so that we create only prefix snippets (or so that not every snippet

Algorithm 1 NAIVE-PATTERN-MATCHING$(s_1, s_2, \ldots, s_n)$

$\ell \leftarrow$ longest prefix of p ending s_1 ▷ **Lemma 3.8**
$k \leftarrow 2$
while $k \leq n$ and $\ell + \sum_{i=k}^{n} |s_i| \geq m$ **do**
  check for an occurrence of p in $p[1\,..\,\ell]s_k$ ▷ **Lemma 4.1**
  if $p[1\,..\,\ell]s_k$ is a prefix of p **then**
    $\ell \leftarrow \ell + |s_k|$
    $k \leftarrow k + 1$
    continue
  end if
  $b \leftarrow$ longest long border of $p[1\,..\,\ell]$ s.t. $p[1\,..\,b]s_k$ is a prefix of p ▷ **Lemma 4.2**
  if b is undefined **then**
    $\ell \leftarrow$ longest prefix of p ending $p[\lceil \frac{\ell}{2} \rceil\,..\,\ell]$ ▷ **Lemma 5.1**
    continue
  end if
  $\ell \leftarrow b + |s_k|$
  $k \leftarrow k + 1$
end while

has to be extended), but we prefer to keep its description more modular and show how to make the fresh snippets extended instead.

Lemma 5.1. *Information for all prefix and half snippets can be computed in linear time.*

PROOF. In case of prefix snippets the computation is trivial. The case of all half snippets is not completely trivial, though. To compute the suffixes we loop through the possible values of ℓ. Assuming we have the longest suffix of $p[\lceil \frac{\ell}{2} \rceil\,..\,\ell]$ which is a prefix $p[1\,..\,i]$, we try to compute such longest suffix of $p[\lceil \frac{\ell+1}{2} \rceil\,..\,\ell+1]$. For that we consider the borders of $p[1\,..\,i]$ looking for the one which can be extended with $p[\ell+1]$. Then we check if the extended border is longer than $p[\lceil \frac{\ell+1}{2} \rceil\,..\,\ell+1]$ and if so, take its border. The total complexity of this procedure is linear because at each step we increase the length of the current prefix by at most 1. Now consider locating the vertices in the suffix tree. Assume that we know the corresponding vertex for a given value of ℓ and need to update the information after increasing ℓ by one. Extending the current word by one letter requires traversing at most one edge in the suffix tree, then we might need to remove the first letter. If the current vertex is explicit, we can use its suffix link. Otherwise we use the suffix link of its deepest explicit ancestor, and then traverse edges down from the vertex found. This traversing might require more than constant time, but can be amortized by noting that the number of explicit ancestors of the current vertex cannot exceed n, and decreases by at most one at for each ℓ. To finish the proof note that looking up the edge can be performed very quickly even when the alphabet is not constant: there are at most two half snippets of a given length and so we can afford to simply iterate through all outgoing edges, then the total complexity of all lookups will be just linear in $|p|$. □

Theorem 5.1. NAIVE-PATTERN-MATCHING *works in time* $\mathcal{O}(n \log m)$.

Algorithm 2 LEVERED-PATTERN-MATCHING$(s_1, s_2, \ldots, s_n)$

1: $\ell \leftarrow$ longest prefix of p ending s_1 ▷ **Lemma 3.8**
2: $k \leftarrow 2$
3: **while** $k \leq n$ and $\ell + \sum_{i=k}^{n} |s_i| \geq m$ **do**
4: choose $t \geq k$ minimizing $|s_k| + |s_{k+1}| + \ldots + |s_{t-1}| - \frac{|s_t|}{2}$
5: **if** $\ell + |s_k| + |s_{k+1}| + \ldots + |s_{t-1}| \leq \frac{|s_t|}{2}$ **then**
6: check for occurrence of p in $p[1\,..\,\ell]s_k s_{k+1} \ldots s_t$ ▷ **Lemma 5.2**
7: $\ell \leftarrow$ longest prefix of p ending $p[1\,..\,\ell]s_k s_{k+1} \ldots s_t$ ▷ **Lemma 5.3**
8: $k \leftarrow t + 1$
9: **else**
10: execute lines 4–16 of NAIVE-PATTERN-MATCHING
11: **end if**
12: **end while**

PROOF. Observe that each iteration of the **while** loop results in either increasing k by 1 or decreasing $\ell \leq m$ at least twice. Thus the total number of iterations is $\mathcal{O}(n \log m)$. In each iteration we spend just a constant time due to the above lemmas. Note that we require that for any snippet we know not only its start and end in p but also the corresponding vertex in the suffix tree (this will be very important for the improved method). While we assume that we get such information for all the input snippets, we might create some new snippets during the execution of the algorithm. Fortunately, the only possible non-input snippets we create are prefix and half snippets, and so we can use Lemma 5.1 to preprocess them. □

To accelerate the above basic method we use the concept of *levers*:

Definition 5.1. We call s_i a lever of a sequence of snippets $s_1 s_2 \ldots s_k$ if $\sum_{j=1}^{i-1} |s_j| \leq \frac{|s_i|}{2}$.

Given such a lever we can eliminate many potential occurrences at once. This is formalized in Lemma 5.2 and Lemma 5.3.

Lemma 5.2. *Given a sequence of extended snippets $s_1 s_2 \ldots s_i$ in which s_i is a lever, we can detect an occurrence of p in $s_1 s_2 \ldots s_i$ in time $\mathcal{O}(i)$.*

PROOF. Let $L = |s_i|$ and $S = \sum_{j=1}^{i-1} |s_j|$. Because $S \leq \frac{L}{2}$ and $L \leq m$, S is at most $\frac{m}{2}$, meaning that any occurrence of p must end in s_i. Note that we can safely replace s_i with the longest suffix $p[k\,..\,m]$ of the pattern which is its prefix, by Lemma 3.8 this requires just constant time. First we check if p is a suffix of the whole sequence in time $\mathcal{O}(i)$. Now, any possible occurrence of p cannot end earlier than after the $m - S \geq \frac{L}{2}$-th character and so in fact we are looking for a long border b of $p[k\,..\,m]$ such that $p[1\,..\,m-b]$ ends $s_1 s_2 \ldots s_{i-1}$. From Lemma 3.3 we know that any such b must be of the form $|p[k\,..\,m]| - \alpha d$, where $d \leq \frac{|p[k..m]|}{2}$ is the period of $p[k\,..\,m]$ calculated in constant time by Lemma 3.9. It turns out that we can restrict the set of possible values of α to just one, which can be then checked naively in time $\mathcal{O}(i)$.

First we compute how far the period of $p[k\,..\,m]$ continues to the left starting from the right end of both $p[1\,..\,k-1]$ and $s_1 s_2 \ldots s_{i-1}$. In order to perform both those

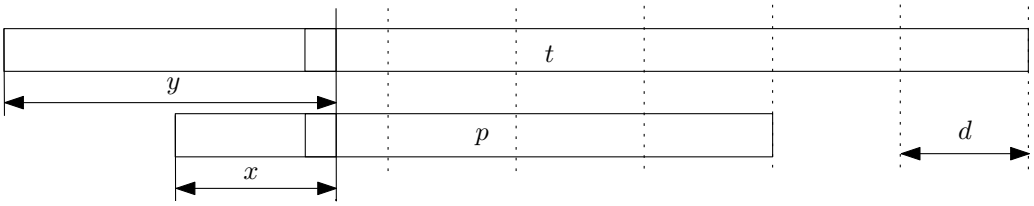


FIGURE 1. Shifting p to get an occurrence in t.

computations efficiently we only have to develop a constant time procedure which, given two snippets $p[i\,..\,j]$ and $p[k\,..\,\ell]$, finds the longest suffix of the former which is also a suffix of some power of the latter. Such procedure works in two steps:

(1) find the longest common suffix of $p[i\,..\,j]$ and $p[k\,..\,\ell]$, return if it is shorter than $|p[k\,..\,\ell]|$,
(2) find and return the longest common suffix of $p[i\,..\,j]$ and $p[i\,..\,j - |p[k\,..\,\ell]|]$.

By repeating this procedure at most i times we can assume that we know how far the period continues in both strings. If d is a period of the whole p, recall that p is not a suffix of the whole $s_1 s_2\,..\,s_i$, and shifting it to the left by any multiple of d cannot result in a match. Otherwise we have found the rightmost position x such that $p[x] \neq p[x+d]$. If d is a period of the whole $t = s_1 s_2\,..\,s_i$, p does not occur there. Otherwise we have found the rightmost position y such that $t[y] \neq t[y+d]$. Now we claim that in order to get a match, we must shift p so that its x-th character is aligned with the y-th character of t, see Figure 1 (recall that we shift p by a multiple of d). Indeed, choosing any smaller shift creates a mismatch concerning $p[x]$ and choosing any bigger shift creates a mismatch concerning $t[y]$. This gives us a simple arithmetic condition on the only possible value of α: if d does not divide $|t| - |p| + x - y$ there is no occurrence, otherwise we check $\alpha = \frac{|t|-|p|+x-y}{d}$. □

Lemma 5.3. *Given a sequence of extended snippets $s_1 s_2 \ldots s_i$ in which s_i is a lever, we can compute the longest prefix of p which is a suffix of $s_1 s_2 \ldots s_i$ in time $\mathcal{O}(i)$.*

PROOF. As in the previous proof, let $L = |s_i|$ and $S = \sum_{j=1}^{i-1} |s_j|$. s_i is an extended snippet so we already know the longest prefix of p which is its suffix. We should check if there is any longer prefix. If so, it corresponds to an occurrence of s_i in $p[1\,..\,S+L]$. First we locate the node corresponding to s_i in the suffix tree built for p. The tree can be preprocessed (in linear time) so that having this node we can compute the first and second occurrence of s_i in $p[1\,..\,S+L]$ (more precisely, we compute the first and second occurrences in the whole p, and check if they are inside $p[1\,..\,S+L]$). If there is none, we terminate. If there is just one, we check naively in time $\mathcal{O}(i)$ the corresponding prefix. If there are two, the situation is more complicated, as in fact there can be many more of them, and we cannot afford to iterate through all possibilities.

Because s_i is a lever, $L \geq 2|S|$ and $L \geq \frac{2}{3}(S+L)$. Thus the overlap of any two occurrences of s_i in $p[1\,..\,S+L]$ is of length at least $\frac{L}{2}$ and so any non-leftmost occurrence corresponds to a long border of s_i. Let d be the period of s_i (note that we do not know d yet), the first occurrence starts at the k-th character of p, and the last occurrence starts

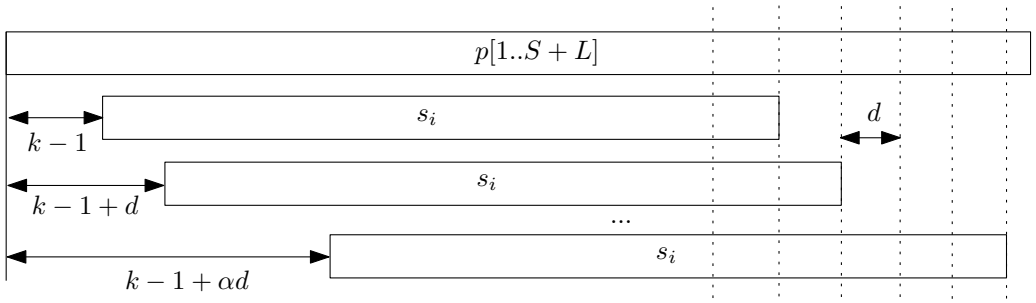


FIGURE 2. All occurrences of s_i in $p[1\,..\,S+L]$.

at the $k+\alpha d$-th character of p. Then it is clear that there are occurrences starting at the $k+\beta d$-th characters, for any $0 \leq \beta \leq \alpha$, see Figure 2. In particular, the second occurrence starts at the $k+d$-th character, so knowing the positions of the first and second occurrence allows us to compute d. Having the value of d and k, we find the longest common prefix of $p[k\,..\,S+L]$ and $p[k+d\,..\,S+L]$, which gives us the value of α (more precisely, if the length is ℓ, $\alpha = \left\lfloor \frac{\ell+d-L}{d} \right\rfloor$). Now we can use the same method as in the previous proof. First compute how far the period of s_i continues to the left starting from the right end of both $p[1\,..\,k-1]$ and $s_1 s_2 \ldots s_{i-1}$. Then consider the following three cases.

(1) If d is a period of $s_1 s_2 \ldots s_i$, for a suffix of length at least $|s_i|$ which is a prefix of p to exists d must be a period of $p[1\,..\,k-1+d]$ as well. Then the longest such suffix corresponds to the biggest $\beta \leq \alpha$ for which $S \geq k-1+\beta d$.
(2) If d is not a period of the whole $s_1 s_2 \ldots s_i$ but is a period of $p[1\,..\,k-1+d]$, any prefix of p of length at least $|s_i|$ which is a suffix of $s_1 s_2 \ldots s_i$ must be completely contained in the periodic suffix of $s_1 s_2 \ldots s_i$. We can select the longest such suffix in constant time.
(3) If d is neither a period of the whole $p[1\,..\,k-1+d]$ nor $s_1 s_2 \ldots s_i$, we get a simple arithmetic condition on the only possible value of α as in the previous lemma. Then we verify this value using $\mathcal{O}(i)$ longest common prefix computations.

□

We are ready to present the improved (and final) algorithm. The idea is that whenever the sequence contains a lever, we can use Lemma 5.2 and Lemma 5.3 to quickly process a bunch of snippets. Otherwise we stick to the same method as in NAIVE-PATTERN-MATCHING. We call the resulting method LEVERED-PATTERN-MATCHING.

While the idea is rather simple, it turns out that the notion of levers captures all inputs on which NAIVE-PATTERN-MATCHING runs in superlinear time. To formalize this claim we will need a few technical lemmas, but before that we show how to execute line 4 efficiently.

Lemma 5.4. *Line 4 of* LEVERED-PATTERN-MATCHING *can be executed in amortized constant time, and $\mathcal{O}(m)$ additional memory.*

PROOF. If we are allowed to use as much as $\Theta(n)$ additional memory, we can preprocess all minima going right to left: the best possible choice of t for a given value of k

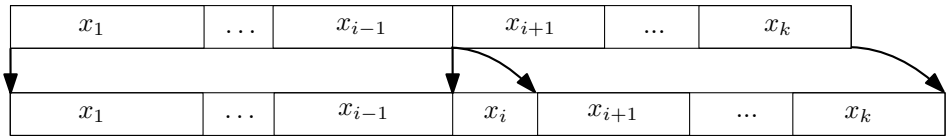


FIGURE 3. Inserting x_i between x_{i-1} and x_{i+1}.

is either the same as for $k+1$, or it is equal k. If we would like to optimize the amount of additional memory used, and avoid the necessity of reading all input snippets when there is a match somewhere near the very beginning, we can use a slightly more complicated method. First note that we are interested only in $t \leq k+m$. For each current value of k we keep an increasing list of *candidates* $k \leq t_1 < t_2 < \ldots < t_c \leq k+m$. Let $f(i) = |s_1| + |s_2| + \ldots + |s_{i-1}| - \frac{|s_i|}{2}$, then t_1 is the position with the minimum value of f in the interval $[k, k+m]$, t_2 is the position with the minimum value of f in the interval $[t_1+1, k+m]$, and so on. Before increasing k by one we need to remove t_1 from the list, if $t_1 = k$, and consider a new candidate $k+1+m$: remove all t_i with $f(t_i) > f(k+1+m)$, and add $k+1+m$ to the list. The amortized cost of this update is clearly constant, and the maximum number of stored candidates $\mathcal{O}(m)$. To find t, simply take the first candidate t_1. □

With each sequence of snippets $s_1 s_2 \ldots s_k$ we associate its potential $\Phi(|s_1|, |s_2|, \ldots, |s_k|)$, which roughly corresponds to the amount of work we still need to perform if we use the concept of levers. Before we use it to bound the running time of LEVERED-PATTERN-MATCHING, we need a few observations.

Definition 5.2. Let $x_1, x_2, \ldots, x_k \leq m$ be a sequence of natural numbers. Consider a segment of length $\sum_{i=1}^{k} x_i$ split into k blocks of length x_i, for $i = 1, 2, \ldots, k$. First mark its suffix of length m. Then, for each i, mark a fragment of length $\frac{x_i}{2}$ ending just before the part corresponding to x_i. Let y_i be the length of the marked suffix in the block corresponding to x_i, and define the potential $\Phi(x_1, x_2, \ldots, x_k)$ as $\sum_{i=1}^{k} 2 + \log \frac{x_i}{y_i}$.

Note than all y_i are strictly positive so the potential is well-defined for any sequence of snippets.

Lemma 5.5. $\Phi(x_1, x_2, \ldots, x_k) \leq 7k$.

PROOF. We apply induction on k. If $k = 1$ the whole segment is marked so the potential is 2 and the claim trivially holds. Let $k > 1$, choose $x_i = \min\{x_1, x_2, \ldots, x_k\}$ and assume that the claim holds for $\Phi(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_k)$. We compute the maximum possible increase in the potential after inserting a block of length x_i between x_{i-1} and x_{i+1}, see Figure 3. There are two reasons the potential might increase.

(1) We create a new block of x_i cells. Note that either $i = k$ and they are all marked, or $i < k$ and $x_{i+1} \geq x_i$ so at least $\frac{x_i}{2}$ of them are marked. In either case, the resulting increase in the potential is at most $2 + \log \frac{x_i}{y_i} \leq 2 + \log 2 = 3$.
(2) We move all blocks corresponding to $x_{i+1}, \ldots, x_k$ further to the right. It might result in unmarking some cells in the blocks corresponding to $x_1, x_2, \ldots, x_{i-1}$. Because we shift all those blocks by x_i to the right, the unmarked blocks are

contained in a segment of such length. Because x_i is smaller than any x_j with $j < i$, any segment of length x_i intersects at most two blocks on the left of x_i. Consider the situation in one such block: the old potential was $2 + \log \frac{x_j}{y_j}$, the new potential is $2 + \log \frac{x_j}{y'_j}$ with $y'_j \geq \max(y_j - x_i, \frac{x_{j+1}}{2}) \geq \max(y_j - x_i, \frac{x_i}{2})$. The increase is at most:

$$\log \frac{x_j}{y'_j} - \log \frac{x_j}{y_j} = \log \frac{y_j}{y'_j} \leq \log \frac{y_j}{\max(y_j - x_i, \frac{x_i}{2})}$$
$$= \begin{cases} \log \frac{y_j}{y_j - x_i} \leq \log \frac{y_j}{y_j - \frac{2}{3}y_j} = \log 3 & \text{if } x_i \leq \frac{2}{3}y_j \\ \log \frac{2y_j}{x_i} \leq \log \frac{2\frac{3}{2}x_i}{x_i} = \log 3 & \text{if } x_i > \frac{2}{3}y_j \end{cases}$$

because there might be two such blocks, the maximum increase is twice as much, $2 \log 3 \leq 4$.

By summing the above cases $\Phi(x_1, x_2, \ldots, x_k) \leq \Phi(x_1, \ldots, x_{i-1}, x_i, \ldots, x_k) + 7 \leq 7k$. □

Lemma 5.6. $\Phi(x_1, x_2, \ldots, x_k) \geq 1 + \Phi(x_1 + x_2, \ldots, x_k)$.

Proof. Let $y \geq y_2$ be the number of marked cells in the block corresponding to $x_1 + x_2$ in $\Phi(x_1 + x_2, x_3, \ldots, x_k)$, see Figure 4. The only change in the potential concerns the first $x_1 + x_2$ cells:

$$\begin{aligned} \Delta &= \Phi(x_1, x_2, x_3, \ldots, x_k) - \Phi(x_1 + x_2, x_3, \ldots, x_k) \\ &= 2 + \log \frac{x_1}{y_1} + \log \frac{x_2}{y_2} - \log \frac{x_1 + x_2}{y} \end{aligned}$$

We have a few cases to consider.

(1) $x_1 \leq \frac{x_2}{2}$. Then $y_1 = x_1$ and the change in the potential is:

$$\begin{aligned} \Delta &\geq 2 + \log \frac{x_2}{y_2} - \log \frac{x_1 + x_2}{y_2} \\ &= 1 + \log \frac{2x_2}{x_1 + x_2} \geq 1 + \log \frac{4}{3} \geq 1 \end{aligned}$$

(2) $x_1 \geq \frac{x_2}{2}$ and $y_1 = \frac{x_2}{2}$. The change in the potential is:

$$\begin{aligned} \Delta &\geq 2 + \log \frac{x_1}{y_1} + \log \frac{x_2}{y_2} - \log \frac{x_1 + x_2}{y_2} \\ &\geq 3 + \log \frac{x_1}{x_2} + \log \frac{x_2}{y_2} - \log \frac{x_1 + x_2}{y_2} \\ &= 3 + \log \frac{x_1}{y_2} - \log \frac{x_1 + x_2}{y_2} = 3 + \log \frac{x_1}{x_1 + x_2} \\ &\geq 3 + \log \frac{1}{3} = 1 + \log \frac{4}{3} \geq 1 \end{aligned}$$

(3) $y_1 > \frac{x_2}{2}$. Then all marked cells in the block corresponding to x_1 are marked either because of some long x_i with $i > 2$ or because they are among the m

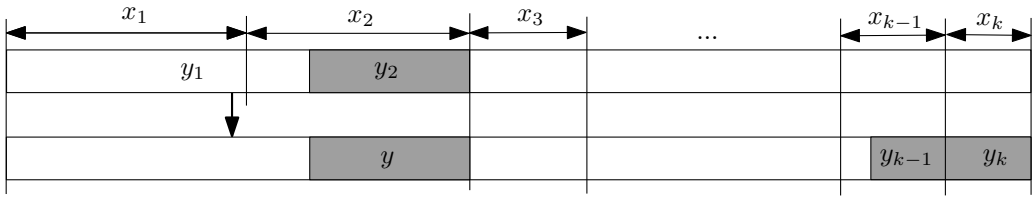


FIGURE 4. Merging blocks corresponding to x_1 and x_2.

rightmost cells, so $y_2 = x_2$ and merging two first blocks does not change the number of marked cell there, $y = y_1 + y_2$. We can bound Δ as follows:

$$\begin{aligned}\Delta &= 2 + \log\frac{x_1}{y_1} - \log\frac{x_1 + x_2}{y_1 + y_2} \\ &= 2 + \log\frac{x_1}{y_1} - \log\frac{x_1 + x_2}{y_1 + x_2} \geq 2\end{aligned}$$

where the last inequality follows from the fact that if $p \geq q$ then $\frac{p}{q} \geq \frac{p+r}{q+r}$.

□

Lemma 5.7. *$\Phi(x_1, x_2, \ldots x_k) \geq \Phi(x'_1, x_2, \ldots, x_k)$ if $x_1 \geq x'_1$.*

PROOF. Decreasing x_1 cannot change any y_i with $i > 1$. Let y'_1 be the number of marked cells in the block corresponding to x'_1. Then $y_1 = y'_1$, and from the definition of the potential we get the claim, or $y'_1 = x'_1$, and $\log\frac{x'_1}{y'_1} = 0 \leq \log\frac{x_1}{y_1}$. □

Theorem 5.2. LEVERED-PATTERN-MATCHING *can be implemented to work in time $\mathcal{O}(n)$ and use $\mathcal{O}(m)$ additional memory.*

PROOF. Consider any execution of the algorithm. In the very beginning we allocate $\Phi(|s_1|, |s_2|, \ldots, |s_n|)$ credits which we then use to pay for all executions of lines 4–10 of LEVERED-PATTERN-MATCHING. We keep the following invariant during the whole execution: we have $\Phi(\ell, |s_k|, |s_{k+1}|, \ldots, |s_n|)$ credits available. Consider a single pass through the body of the **while** loop. From Lemma 5.4 the amortized cost of executing line 4 is constant. Consider the remaining lines.

(1) There is a lever s_t in $p[1\,..\,\ell]s_k s_{k+1} \ldots s_t$. We need as much as $\mathcal{O}(t - k + 1)$ time to process it, but then by $t - k + 1$ applications of Lemma 5.6 the required potential is at most:

$$\begin{aligned}&\Phi(\ell + |s_k| + |s_{k+1}| + \ldots + |s_t|, |s_{t+1}|, \ldots, |s_n|) \\ &\quad \leq \Phi(\ell, |s_k|, |s_{k+1}|, \ldots, |s_n|) - (t - k + 1)\end{aligned}$$

which leaves us with $t - k + 1$ free credits which we can use to pay for the processing time.

(2) There is no lever. Then we execute the corresponding lines from NAIVE-PATTERN-MATCHING which takes just constant time and results in either increasing k by 1 and ℓ by at most $|s_k|$ or decreasing ℓ at least twice. In the first case the required potential is by Lemma 5.6 and Lemma 5.7 at most:

$$\Phi(\ell + |s_k|, |s_{k+1}|, \ldots, |s_n|) \leq \Phi(\ell, |s_k|, \ldots, |s_n|) - 1$$

The second case is slightly more involved: if decreasing ℓ creates a lever, we pay for the pass in the next round. Otherwise we replace $\log \frac{x_1}{y_1}$ with $\log \frac{x_1}{2y_1}$ in the potential so the required amount of credits left is just:

$$\Phi(\frac{\ell}{2}, |s_k|, \ldots, |s_n|) \leq \Phi(\ell, |s_k|, \ldots, |s_n|) - 1$$

which leaves us with one spare credit.

The total amount of allocated credits is by Lemma 5.5 just $\mathcal{O}(n)$ and so is the time complexity. □

3. LZW compression

Recall that a LZW compressed string is a sequence of codewords $t_1 t_2 \ldots t_n$, each codeword being either a single letter, or a previously occurring codeword concatenated with a single letter. First we check if the pattern p occurs inside one of the strings represented by the codewords. Then for each codeword we need to know if the string it represents occurs in the pattern p, and if it does, we need to find the corresponding vertex in the suffix tree. We also need to know its longest prefix which is a suffix of p, and the longest suffix which is a prefix of p (actually, knowing the prefix is necessary only when the string does not occur inside p). All those informations can be found efficiently (in constant time for each codeword) assuming that the alphabet is of constant size.

Lemma 5.8. *If the alphabet is of constant size, we can perform the preprocessing for all codewords in total linear time.*

Proof. The whole set of codewords should be viewed as a trie. For each vertex of this trie we are required to compute:

(1) the longest prefix of the corresponding word which is a suffix of p,
(2) if the corresponding word occurs in p, locate its (implicit or explicit) vertex in the suffix tree,
(3) the longest suffix of the corresponding word which is a prefix of p.

We build an automaton $\mathcal{A}$ recognizing all prefixes of p, i.e., its states set is $\{0, 1, \ldots, |p|\}$ and after reading a word w we are in the state corresponding to the longest prefix of p ending w. Such automaton can be easily constructed in linear time if the alphabet is of constant size. Using the automaton we can compute the longest suffix of each codeword in constant time per codeword.

Then we build the suffix tree for p in linear time [**45**]. For each codeword we will find the corresponding vertex in the suffix tree, if any. Note that such information is actually enough to compute the longest prefix which is a suffix of p for each codeword: first apply Lemma 3.8 to all codewords with corresponding vertex in the suffix tree. Then for all other codewords, simply take the answer computed for its lowest ancestor with a known result. To locate all corresponding vertices in the suffix tree we traverse the trie in a top-bottom fashion. To find the vertex for v, take the vertex found for its parent and traverse at most one edge. This gives a total linear time. □

After applying the above lemma to the set of all codewords, for each maximal sequence of codewords representing substrings of p ($p[i_1 \ldots j_1]p[i_2 \ldots j_2] \ldots p[i_k \ldots j_k]$) we take the longest suffix of the preceeding codeword which is a prefix of p

($p[1\,..\,i]$) and the longest prefix of the succeeding codeword which is a suffix of p ($p[j\,..\,m]$), and run the algorithm from the previous section on the sequence of snippets $p[1\,..\,i]p[i_1\,..\,j_1]\ldots p[i_k\,..\,j_k]p[j\,..\,m]$. Because each such run takes time $\mathcal{O}(k+2)$, and all those values of k sum up to at most n, the total complexity including the preprocessing of the pattern will be $\mathcal{O}(n+m)$. Note that in fact we do not have to process all codewords in the very beginning: we can process the input in an online fashion codword-by-codeword. Computing the information for each single codeword takes constant time, and if there is an occurrence starting at the i-th codeword, running LEVERED-PATTERN-MATCHING requires accessing no more than $i+m$ first codewords. Hence if the first occurrence starts at the n'-th codeword, we can detect it in $\mathcal{O}(n'+m)$ time.

Theorem 5.3. *Pattern matching for LZW compressed strings over a constant size alphabet can be solved in optimal linear time.*

4. Integer alphabets

The assumption of constant size alphabet is not necessary to achieve the claimed linear running time. If the alphabet is of non-constant size but consists of integers which can be sorted in linear time, we can create all necessary snippets sequences in linear time as well.

Lemma 5.9. *If the alphabet consists of integers which can be sorted in linear time, we can perform the preprocessing for all codewords in total linear time, assuming the RAM model of computation.*

PROOF. We begin with constructing the suffix tree in linear time using the assumption of integer alphabet and applying the result of [**15**]. Then we are left with answering the following three questions for each different codeword:

(1) find its longest prefix which is a suffix of p,
(2) check if it occurs in p, and if it does, find the corresponding vertex,
(3) find its longest suffix which is a prefix of p.

We answer those questions for all codewords at once, which requires reading the whole input even if there is an occurrence in the very beginning. Questions of each type are processed separately.

(1) We use Lemma 3.8 and the information found for the second type questions.
(2) We build a trie T containing all the codewords. Then answering the questions reduces to computing the intersection of this trie and the suffix tree S (which is a compressed representation of a trie containing all subwords of p). This can be done in time $\mathcal{O}(|S|+|T|)$ assuming the edges outgoing from each vertex (both in S and T) are sorted according to their labels. We process the codewords in order of their lengths. Assuming that we know the corresponding vertices for all codewords of length ℓ, we consider all codewords wx of length $\ell+1$, and group them according to the vertex corresponding to w (if there is none, wx clearly does not occur in p). In each group the codewords are sorted using x as the key, which can be assured by sorting all codewords in the very beginning. Then we find the corresponding vertices for all codewords in a single group at once, using a left-to-right scan.

(3) Finding such suffix for a single codeword can be performed by running the Knuth-Morris-Pratt algorithm. While the amortized complexity of processing a single letter is constant, we have a lot of different letters that can extend a given codeword, and the amortization argument does not give a linear bound in such case.

Consider the automaton recognizing all prefixes of p from Lemma 5.8, i.e., with the states set $\{0, 1, \ldots, m\}$ and the transitions $\delta(i, a) = \max\{j : p[1 \ldots j] = p[1 \ldots i]a\}$. If the alphabet is of unbounded size the simple linear time construction is no longer possible. Fortunately, the number of nonzero transitions outgoing from a single vertex is at most $2 \log m$, which follows from a well known fact [**12**, Lemma 2.32] that $\pi'^{(2)}(k) < \frac{k}{2}$, where π' is the so-called strong failure function, defined as follows: $\pi'(k)$ is the longest border of $p[1 \ldots k]$ such that $p[k+1] \neq p[\pi'(k)+1]$. In fact much more is known: due to a result of [**44**], the total number of nontrivial transitions is linear in m, regardless of the size of the alphabet. By Lemma 3.1, we can maintain a collections of sets logarithmic size, with amortized constant time update and worst-case constant time lookup, which is enough to construct the automaton in linear time. We create one set for each state. Assuming that we have the sets for all $i' < i$, we consider the i-th state. Its outgoing transitions are exactly the transitions outgoing from the border of $p[1 \ldots i]$ with one exception: $\delta(i, p[i+1]) = i+1$. Hence we can copy the set computed for the border and add (or replace) one element. Then we are able to compute the transition function in constant time per character, which allows us to process all codewords in linear time. Introducing the atomic heaps results in a rather complicated method, and in this particular case it is possible to significantly simplify this structure because the universe is of just polynomial (in m) size. It does not result in changing the claimed time bounds, though.

□

We believe the method of storing the automaton used in the above proof, while simple, might find other applications.

This gives us the final result. Note that in case of integer alphabets, it is not clear how to avoid reading the whole input even in the case when there is an occurrence somewhere in the very beginning.

Theorem 5.4. *Pattern matching for LZW compressed strings over a polynomial size integer alphabet can be solved in optimal linear time assuming the word RAM model.*

CHAPTER 6

Fully Lempel-Ziv-Welch compressed pattern matching

1. Overview of the algorithm

Our goal is to detect an occurrence of a pattern $p[1\,..\,M]$ in a given text $t[1\,..\,N]$, where p and t are described by a Lempel-Ziv-Welch parse of size m and n, respectively. The difficulty here is rather clear: M might be of order m^2, and hence looking at each possible prefix or suffix of the pattern would imply a quadratic (or higher) total complexity. As most efficient uncompressed pattern algorithms are based on a more or less involved preprocessing concerning all prefixes or suffixes, such quadratic behavior seems difficult to avoid. Nevertheless, we can try to use the following reasoning here: either the pattern is really complicated, and then m is very similar to M, hence we can use the linear compressed pattern matching algorithm from Chapter 5, or it is in some sense repetitive, and we can hope to speedup the preprocessing by building on this repetitiveness. In this section we give a high level formalization of this intuition.

We will try to process whole codewords at once. To this aim we need the following technical lemma which allows us to compare large chunks of the text (or the pattern) in a single step. We defer its proof to Section 4 as it is not really necessary to understand the whole idea.

Lemma 6.1. *It is possible to preprocess in linear time a LZW parse of a text over an alphabet consisting of integers which can be sorted in linear time so that given any two codewords we can compute their longest common suffix in constant time.*

As an obvious corollary, given two codewords we can check if the shorter is a suffix of the longer in constant time.

In the very beginning we reverse both the pattern and the text. This is necessary because the above lemma tells how to compute the longest common suffix, and we would actually like to compute the longest common prefix. The only way we will access the characters of both the pattern and the text is either through computing the longest common prefix of two reversed codewords, or retrieving a specified character of a reversed codeword (which can be performed in constant time using level ancestor queries), hence the input can be safely reversed without worrying that it will make working with it more complicated. We call those reversed codewords *blocks*. Note that all suffixes of a block are valid blocks as well.

We start with classifying all possible patterns into two types. Note that this classification depends on both m (size of the compressed pattern) and n (size of the compressed texts) which might seem a little unintuitive.

Definition 6.1. A *kernel* of the pattern is any substring of length $n+m$ such that its border is at most $\frac{n+m}{2}$. A kernel decomposition of the pattern is its periodic prefix with period at most $\frac{n+m}{2}$ followed by a kernel.

Note that the distance between two occurrences of such substring must be at least $\frac{n+m}{2}$, and hence a kernel occurs at most $2m$ times in the pattern and $2n$ times in the text. It might happen that there is no (short) kernel, but in such case the pattern is highly repetitive.

Lemma 6.2. *The pattern either has a kernel decomposition or its period is at most $n+m$. Moreover, those two situations can be distinguished in linear time giving us a decomposition if one exists.*

Proof. We start with decompressing the prefix of length $n+m$. If its period d is at least $\frac{n+m}{2}$, we can return it as a kernel. Otherwise we compute the longest prefix of the pattern and the pattern shifted by d characters (or, in other words, we compute how far the period extends in the whole pattern). This can be performed using at most $2(n+m)$ queries described in Lemma 6.1 (note that being able to check if one block is a prefix of another would be enough to get a linear total complexity here, as we can first identify the longest prefix consisting of whole blocks in both words and then inspect the remaining at most $\min(n,m)$ characters naively). If d is the period of the whole pattern, we are done. Otherwise we identified a substring s of length $n+m$ followed by a character a such that the period of s is at most $d \leq \frac{n+m-1}{2}$ but the period of sa is different (larger). We remove the first character of s and get s'. Let d' be the period of $s'a$. If $d' \geq \frac{n+m}{2}$, $s'a$ is a kernel. Otherwise $d, d' \leq \frac{n+m-1}{2}$ are both periods of s', and hence by Lemma 3.2 they are both multiplies of the period of s'. Let b be the character such that d is a period of $s'b$ (note that $a \neq b$). Because d is a period of $s'b$, $s'[|s'|+1-d] = b$. Similarly, because d' is a period of $s'a$, $s'[|s'|+1-d'] = a$. Hence $s'[|s'|+1-d] \neq s'[|s'|+1-d']$, and because $(|s'|+1-d)-(|s'|+1-d')$ is a multiple of the period of s' we get a contradiction. Note that the prefix before $s'a$ is periodic with period $d \leq \frac{n+m}{2}$ by the construction. □

If the pattern turns out to be repetitive, we try to apply the algorithm from Section 4. The intuition is that while we required a certain preprocessing of the whole pattern, when its period is d it is enough preprocess just the prefix of length $\mathcal{O}(d)$. This intuition is formalized in Section 2. If the pattern has a kernel, we use it to identify $\mathcal{O}(n)$ potential occurrences, which we then manage to verify efficiently. The verification uses a similar idea to the one from Lemma 6.1 but unfortunately it turns out that we need to somehow compress the pattern during the verification as to keep the running time linear. The details of this part are given in Section 3.

2. Detecting occurrence of a periodic pattern

If the pattern is periodic, we would like to somehow use this periodicity so that we do not have to preprocess the whole pattern (i.e., build the suffix tree, LCA structure, compute the borders of all prefixes and suffixes, and so on). It seems reasonable that preprocessing just the first few repetitions of the period should be enough. More precisely, we will decompress a sufficiently long prefix of p and compute some of its occurrences

Algorithm 3 LAZY-LEVERED-PATTERN-MATCHING$(s_1, s_2, \ldots, s_n)$

1: $\ell \leftarrow$ longest prefix of p ending s_1 ▷ **Lemma 3.8**
2: $k \leftarrow 2$
3: **while** $k \leq n$ and $\ell + \sum_{i=k}^{n} |s_i| \geq m$ **do**
4: choose $t \geq k$ minimizing $|s_k| + |s_{k+1}| + \ldots + |s_{t-1}| - \frac{|s_t|}{2}$
5: **if** $\ell + |s_k| + |s_{k+1}| + \ldots + |s_{t-1}| \leq \frac{|s_t|}{2}$ **then**
6: output the first occurrence of p in $p[1\,..\,\ell]s_k s_{k+1} \ldots s_t$, if any ▷ **Lemma 5.2**
7: $\ell \leftarrow$ longest prefix of p ending $p[1\,..\,\ell]s_k s_{k+1} \ldots s_t$ ▷ **Lemma 5.3**
8: $k \leftarrow t + 1$
9: **else**
10: output the first occurrence of p in $p[1\,..\,\ell]s_k$, if any ▷ **Lemma 4.1**
11: **if** p occurs in $p[1\,..\,\ell]s_k$ **then**
12: $\ell \leftarrow$ longest prefix of p ending $p[\lceil \frac{\ell}{2} \rceil\,..\,\ell]$ ▷ **Lemma 5.1**
13: **continue**
14: **end if**
15: execute lines 5–16 of NAIVE-PATTERN-MATCHING
16: **end if**
17: **end while**

inside the text. To compute those occurrences we apply a fairly simple modification of LEVERED-PATTERN-MATCHING called LAZY-LEVERED-PATTERN-MATCHING.

First observe that both Lemma 4.1 and Lemma 5.2 can be modified in a straightforward way so that we get the leftmost occurrence, if any. The original procedure quits as soon it detects that the pattern occurs. We would like it to proceed so that we get more than one occurrence, though. A naive solution would be to simply continue, but then the following situation could happen: both ℓ and $|s_k|$ are very close to m, the pattern occurs both in the very beginning of $p[1\,..\,\ell]s_k$ and somewhere close to the boundary between the two parts, and the longest suffix of the concatenation which is a prefix of the pattern is very short. Then we would detect just the first occurrence, and for some reasons that will be clear in the proof of Lemma 6.6 this is not enough. Hence whenever there is an occurrence in the concatenation, we skip just the first half of $p[1\,..\,\ell]$ and continue.

While LAZY-LEVERED-PATTERN-MATCHING it is not capable of generating all occurrences in some cases, it will always detect a lot of them, in a certain sense. This is formalized in the following lemma.

Lemma 6.3. *If the pattern of length m occurs starting at the i-th character,* LAZY-LEVERED-PATTERN-MATCHING *detects at least one occurrence starting at the j-th character for some $j \in \{i - \frac{m}{2}, i - \frac{m}{2} + 1, \ldots, i\}$.*

PROOF. There are just two places where we can lose a potential occurrence: line 7 and 12. More precisely, it is possible that we output an occurrence and then skip a few others. We would like to prove that the occurrences we skip are quite close to the occurrences we output. We consider the two problematic lines separately.

line 7: s_t is a lever, so $\ell + |s_1| + \ldots + |s_t| \leq \frac{3}{2}m$. Hence the distance between any two occurrences of the pattern inside $p[1\,..\,\ell]s_k s_{k+1} \ldots s_t$ is at most $\frac{m}{2}$. We output the first of them, and so can safely ignore the remaining ones.

line 12: If there is an occurrence, we remove the first half of $p[1\,..\,\ell]$ and might skip some other occurrences starting there. If the first occurrence starts later, we will not skip anything. Otherwise we output the first occurrence starting in $p[1\,..\,\frac{\ell}{2}]$, and if there is any other occurrence starting there, their distance is at most $\frac{\ell}{2} \leq \frac{m}{2}$, hence we can safely ignore the latter.

□

Furthermore, it is easy to see that its running time of Levered-pattern-matching is the same as of Lazy-levered-pattern-matching, i.e., linear.

Lemma 6.4. Lazy-levered-pattern-matching *can be implemented to work in time* $\mathcal{O}(n)$ *and use* $\mathcal{O}(m)$ *additional memory.*

Proof. The proof is almost the same as in Theorem 5.2. The only difference as far as the running time is concerned is line 12. By removing the first half of $p[1\,..\,\ell]$ we either decrease the current potential by 1 or create a lever and thus can amortize the constant time used to locate the first occurrence of the pattern inside $p[1\,..\,\ell]s_k$. □

Note that Lazy-levered-pattern-matching works with a sequence of snippets. By first applying the preprocessing described in Lemma 5.9 we can use it to compute a small set which approximates all occurrences in a compressed text.

Lemma 6.5. Lazy-levered-pattern-matching *can be used to compute a set* S *of* $\mathcal{O}(n)$ *occurrences of an uncompressed pattern of length* $m \geq n$ *in a compressed text such that whenever there is an occurrence starting at the* i*-th character,* S *contains* j *from* $\{i - \frac{m}{2}, i - \frac{m}{2} + 1, \ldots, i\}$.

Proof. By Lemma 5.9 we can reduce compressed pattern matching to pattern matching in a sequence of snippets in linear time. Because $m \geq n$, the preprocessing does not produce any occurrences yet. Then we apply Lazy-levered-pattern-matching. Because its running time is linear by Lemma 6.4, it cannot find more than $\mathcal{O}(n+m)$ occurrences. A closer look at the analysis shows that the number of occurrences produced can be bounded by the potentials of all sequences created during the initial preprocessing phase, which by Lemma 5.5 is at most $\mathcal{O}(n)$. □

Now it turns out that if the pattern is compressed but highly periodic, the occurrences found in linear time by the above lemma applied to a sufficiently long prefix of p are enough to detect an occurrence of the whole pattern.

Lemma 6.6. *Fully compressed pattern matching can be solved in linear time if the pattern is of compressed size* $m \geq n$ *and its period is at most* $\frac{n+m}{2}$*. Furthermore, given a set of* r *potential occurrences we can verify all of them in* $\mathcal{O}(n + m + r)$ *time.*

Proof. We build the shortest prefix $p[1\,..\,\alpha d]$ such that $\alpha d \geq n+m$, where $d \leq \frac{n+m}{2}$ is the period of the whole pattern. Observe that $\alpha d \leq \frac{3}{2}(n+m)$ and hence we can afford to store this prefix in an uncompressed form. By Lemma 6.5 we construct a set S of $\mathcal{O}(n)$ occurrences of $p[1\,..\,\alpha d]$ such that for any other occurrence starting at the i-th character

there exists $j \in S$ such that $0 \leq i - j \leq \frac{\alpha d}{2} \leq \frac{3}{2}(n+m)$. We partition the elements in S according to their remainders modulo d so that $S_r = \{j \in S : j \equiv r \pmod d\}$ and consider each S_r separately. Note that we can easily ensure that its elements are sorted by either applying radix sort to the whole S or observing that LAZY-LEVERED-PATTERN-MATCHING generate the occurrences from left to right.

We split S_r into maximal groups of consecutive elements $x_1 < x_2 < \ldots < x_k$ such that $x_{i+1} \leq x_i + \frac{\alpha d}{2}$, which clearly can be performed in linear time with a single left-to-right sweep. Each such group actually corresponds to a fragment starting at the x_1-th character and ending at the $x_k + \alpha d - 1$ character which is a power of $p[1\,..\,d]$. This is almost enough to detect an occurrence of the whole pattern. If the fragment is sufficiently long, we get an occurrence. In some cases this is not enough to detect the occurrence because we might be required to extend the period to the right as to make sufficient space for the whole pattern. Fortunately, it is impossible to repeat $p[1\,..\,d]$ more than $\frac{3}{2}\alpha$ times starting at the x_k character, as otherwise we would have another $x_{k+1} \in S_r$ which we might have used to extend the group. Hence to compute how far the period extends it would be enough to align $p[1\,..\,\alpha d]p[1\,..\,\frac{\alpha d}{2}]$ starting at the x_k character and compute the first mismatch with the text. We can assume that all suffixes of $p[1\,..\,\alpha d]$ are blocks with just a linear increase in the problem size, and hence we can apply Lemma 6.1 to preprocess the input so that each such alignment can be processed in time proportional to the number of block in the corresponding fragment of the text. To finish the proof, note that any single block in the text will be processed at most twice. Otherwise we would have two groups ending at the x_k-th and $x'_{k'}$-th characters such that $\left|x_k + \alpha d - (x'_{k'} + \alpha d)\right| \leq \frac{\alpha d}{2}$ and that would mean that one of those groups is not maximal. After computing how far the period extends after each group, we only have to check a simple arithmetic conditions to find out if the pattern occurs starting at the corresponding x_1.

To verify a set of r potential occurrences, we construct the groups and compute how far the period extends after each of them as above. Then for each potential occurrence starting at the b_i-th character we lookup the corresponding $S_{b_i \bmod d}$ and find the rightmost group such that $x_1 \leq b_i$. We can verify an occurrence by looking up how far the period extends after the x_k-th character and checking a simple arithmetic condition. To get the claimed time bound, observe that we do not have to perform the lookup separately for each possible occurrence. By first splitting them according to their remainders modulo d and sorting all x_1 and b_i in linear time using radix sort consisting of two passes we get a linear total complexity. □

3. Using kernel to accelerate pattern matching

We start with computing all occurrences of the kernel in both the pattern and the text. Because the kernel is long and aperiodic, there are no more than $2m$ of the former and $2n$ of the latter. The question is if we are able to detect all those occurrences efficiently. It turns out that because the kernel is aperiodic, LAZY-LEVERED-PATTERN-MATCHING can be (again) used for the task. More formally, we have the following lemma.

Lemma 6.7. LAZY-LEVERED-PATTERN-MATCHING *can be used to compute in* $\mathcal{O}(n+m)$ *time all occurrences of an aperiodic pattern of length* $m \geq n$ *in a compressed text.*

Algorithm 4 NAIVE-SCAN(x, y)

```
1: r ← 0
2: while x ≤ |T| and y ≤ |T| do
3:     if t[x] ≠ t[y] then
4:         break
5:     end if
6:     x ← x + 1, y ← y + 1, r ← r + 1
7: end while
8: return r
```

PROOF. By Lemma 6.5 we can construct in linear time a set of occurrences such that any other occurrence is quite close to one of them. But because the pattern is aperiodic, if it occurs at positions i and j with $|i - j| \leq \frac{m}{2}$, then in fact $i = j$. Hence the set contains all occurrences. □

We apply the above lemma to find the occurrences of the kernel in both the pattern and the text. Each occurrence of the kernel in the text gives us a possible candidate for an occurrence of the whole pattern (for example by aligning it with the first occurrence of the kernel in the pattern). Hence we have just a linear number of candidates to verify. Still, the verification is not trivial. An obvious approach would be to repeat a computation similar to the one from Lemma 6.2 for each candidate. This would be too slow, though, as it might turn out that some blocks from the pattern are inspected multiple times. We require a slightly more sophisticated approach.

Using (any) kernel decomposition of the pattern we represent it as $p = p_1p_2p_3$, where the period of p_1 is at most $\frac{n+m}{2}$, and p_2 is a kernel. We start with locating all occurrences of p_2p_3 in the text. It turns out that because p_2 is aperiodic, there cannot be too many of them. Hence we can afford to generate all such occurrences and then verify if any of them is preceded by p_1 as follows:

(1) if $|p_1| \geq n$ then we can directly apply Lemma 6.6,
(2) if $|p_1| < n$ then take the prefix of p_1p_2 consisting of the first $n + m$ letters. Depending on whether this prefix is periodic with the period at most $\frac{n+m}{2}$ or aperiodic, we can apply Lemma 6.6 or Lemma 6.7.

The most involved part is computing all occurrences of p_2p_3. To find them we construct a new string T by concatenating the suffix of the pattern and the text:

$$T = p_2p_3\$t[1\,..\,N]$$

We would like to compute for this new string the values of the prefix function defined in the following way:

$$\text{PREF}[i] = \max\{j : T[k] = T[i + k - 1] \text{ for all } k = 1, 2, \ldots, j\}$$

Of course we cannot afford to compute PREF$[i]$ for all possible $N + M$ values of i. Fortunately, PREF$[i] \geq |p_2|$ iff p_2 occurs in T starting at the i-th character. Because $|p_2| = n + m$ and p_2 is aperiodic, there are no more than $2\frac{N+M}{n+m} \leq n + m$ such values of i. We aim to compute PREF$[i]$ just for those i. First lets take a look at the relatively

Algorithm 5 PREF$(T[1\,..\,|T|])$

```
1:  PREF[1] = 0, s ← 1
2:  for i = 2, 3, ..., |T| do
3:      k ← i − s + 1
4:      r ← s + PREF[s] − 1
5:      if r < i then
6:          PREF[i] = Naive-Scan(i, 1)
7:          if PREF[i] > 0 then
8:              s ← i
9:          end if
10:     else if PREF[k] + k < PREF[s] then
11:         PREF[i] ← PREF[k]
12:     else
13:         x ← Naive-scan(r + 1, r − i + 2)
14:         PREF[i] ← r − i + 1 + x
15:         s ← i
16:     end if
17: end for
18: PREF[1] = |t|
```

well-known algorithm which computes all PREF$[i]$ for all i, which can be found in the classic stringology book by Crochemore and Rytter [**13**]. We state its code for the sake of completeness. Naive-scan(x, y) performs a naive scanning of the input starting at the x-th and y-th characters. PREF uses this procedure in a clever way as to reuse already processed parts of the input and keep the total running time linear. The complexity is linear because the value of $s + \text{PREF}[s]$ cannot decrease nor exceed $|T|$, and whenever it increases we are able to pay for the time spent in Naive-scan using the difference between the new and the old value.

We will transform this algorithm so that it computes only PREF$[i]$ such that the kernel occurs starting at the i-th character. We call such positions i *interesting*. The first problem we encounter is that we need a constant time access to any PREF$[i]$ and cannot afford to allocate a table of length $|T|$. This can be easily overcome.

Lemma 6.8. *A random access table* PREF *such that* $\text{PREF}[i] > 0$ *iff the kernel occurs starting at the i-th character can be implemented in constant time per operation requiring space and preprocessing time not exceeding the compressed size of T, which is $\mathcal{O}(n+m)$.*

Proof. Observe that any two occurrences of the kernel cannot be too close. More precisely, their distance must be at least $\frac{n+m}{2}$. We split the whole T into disjoint fragments of size $\frac{n+m}{2}$. There are no more than $2(n+m)$ of them and there is at most one occurrence in each of them. Hence we can implement the table by allocating an array of size $2(n+m)$ with each entry storing at most one element. □

We modify line 2 so that it iterates only through i which are interesting. Note that whenever we access some PREF$[j]$ inside, j is either i, s or $k = i - s + 1$. In the first

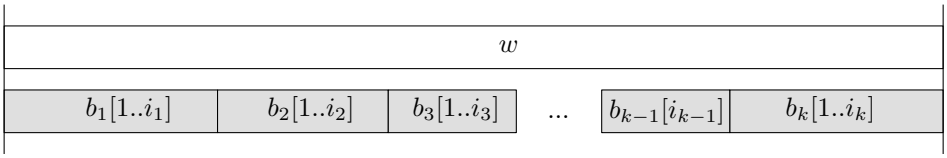


FIGURE 1. A block cover of w.

two cases it is clear that the corresponding positions are interesting so we can access the corresponding value using Lemma 6.8. The third case is not that obvious, though. It might happen that k is not interesting and we will get $\text{PREF}[k] = 0$ instead of the true value. If $r \geq i + |p_2| - 1$ then because p_2 occurs at i, it occurs at k as well, and so k is interesting. Otherwise we cannot access the true value of $\text{PREF}[k]$, so we start a naive scan by calling NAIVE-SCAN$(i + |p_2|, |p_2| + 1)$ (we can start at the $|p_2| + 1$-th character because p_2 occurs at i). After the scanning we set $s \leftarrow i$. Note that because $r < i + |p_2| - 1$, this increases the current value of $s + \text{PREF}[s]$, and we can use the increase to amortize the scanning.

We still have to show how to modify NAIVE-SCAN. Clearly we cannot afford to perform the comparisons character by character. By the increasing $s + \text{PREF}[s]$ argument, any single character from the text is inspected at most once by accessing $T[x]$ (we call it a left side access). It might be inspected multiple times by accessing $T[y]$, though (which we call a right side access). We would like to perform the comparisons block by block using Lemma 6.1. After a single query we skip at least one block. If we skip a block responsible for the left side access, we can clearly afford to pay for the comparison. We need to somehow amortize the situation when we skip a block responsible for the right side access. For this we will iteratively compress the input (this is similar to the idea used in [**24**] with the exception that we work with PREF instead of the failure function). More formally, consider the sequence of blocks describing p_2p_3. First note that no further blocks from T will be responsible for a right side access because of the unique \$ character. Whenever some two neighboring blocks b_1, b_2 from this prefix occur in the same block from the text b', we would like to glue them, i.e., replace by a single block. We cannot be sure that there exists a block corresponding to their concatenation, but because we know where it occurs in b' we can extract (in constant time, by using the level ancestor data structure to preprocess the whole code trie) a block for which the concatenation is a prefix. We will perform such replacement whenever possible. Unfortunately, after such replacement we face a new problem: p_2p_3 is represented as a concatenation of prefixes of blocks instead of whole blocks. Nevertheless, we can still apply Lemma 6.1 to compute the longest common prefix of two prefixes of blocks $b[1..i]$ and $b'[1..i']$ by first computing the longest common prefix of b and b', and decreasing it if it exceeds $\min(i, i')$. More formally, we store a *block cover* of p_2p_3.

Definition 6.2. A block cover of a word w is a sequence of prefixes of blocks $b_1[1\,..\,i_1], b_2[1\,..\,i_2], \ldots, b_k[1\,..\,i_k]$ such that their concatenation is equal to w.

This definition is illustrated on Figure 1. Obviously, the initial partition of p_2p_3 into blocks is a valid block cover. If during the execution of NAIVE-SCAN we find out

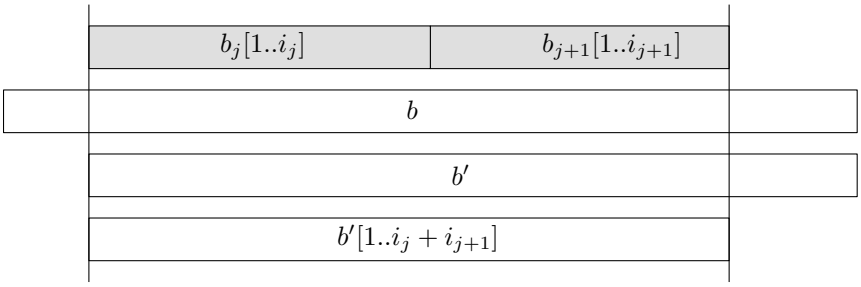


FIGURE 2. Compressing the current block cover.

that two neighboring elements $b_j[1\,..\,i_1], b_{j+1}[1\,..\,i_{j+1}]$ of the current cover occur in some other longer block b, we replace them with the corresponding prefix of b', see Figure 2.

We store all $b_j[1\,..\,i_j]$ on a doubly linked list and update its elements accordingly after each replacement. The final required step is to show how we can quickly access in line 13 the block corresponding to $r - i + 2$. We clearly are allowed to spend just constant time there. We keep a pointer to the block covering the r-th character of the pattern. Whenever we need to access the block covering the $(r - i + 2)$-th character, we simply move the pointer to the left, and whenever the current longest match extends, we move the pointer to the right. We cannot move to the left more time than we move to the right, and the latter can be bounded by the number of blocks in the whole p_1p_2 if we replace the neighboring blocks whenever it is possible.

Such modification of the PREF procedure gives us the following lemma.

Lemma 6.9. *Fully compressed pattern matching can be solved in linear time if we are given the kernel of the pattern.*

4. LZW parse preprocessing

The goal of this section is to prove Lemma 6.1. Recall that we aim to preprocess the codewords trie so that given any two codewords, we can check if the shorter is a suffix of the longer in constant time. It turns out that we can use some existing (while maybe not very known) tools to achieve that. The *suffix tree of a tree* A, where A is a tree with edges labeled with single characters, is defined as the compressed trie containing $s_A(v)\$$ for all $v \in A$, where $s_A(v)$ is the string constructed by concatenating the labels of all edges on the v-to-root path in A, see Figure 3. This has been first used by Kosaraju [**33**], who developed a $\mathcal{O}(|A| \log |A|)$ time construction algorithm, where $|A|$ is the number of nodes of A. The complexity has been then improved by Breslauer [**9**] to just $\mathcal{O}(|A| \log |\Sigma|)$ (which for constant alphabets is linear), and by Shibuya [**43**] to linear for integer alphabets.

We build the suffix tree of the codeword trie T in linear time [**43**]. As a result we also get for any node v of the input trie the node of the suffix tree corresponding to $s_T(v)\$$. Now assume that we would like to compute the longest common suffix of two codewords corresponding to nodes u and v in the input trie. In other words, we would like to compute the longest common prefix of $s_T(u)$ and $s_T(v)$. This can be found in constant time after a linear time preprocessing by retrieving the lowest common ancestor

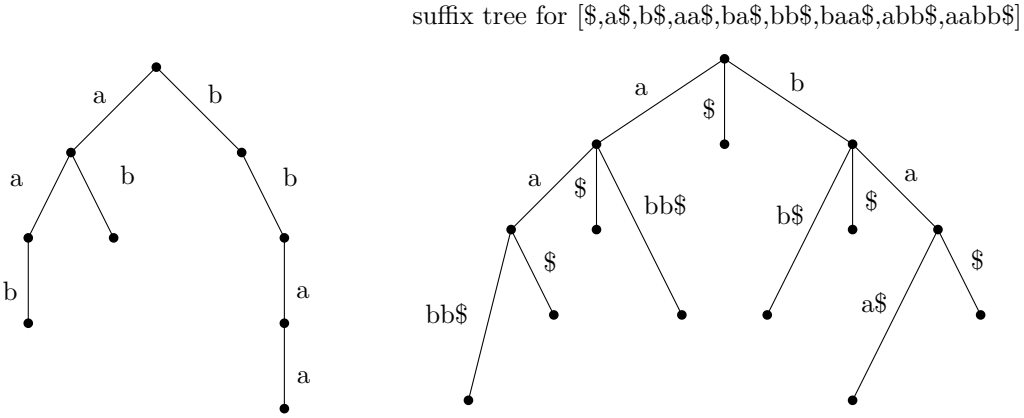


FIGURE 3. Suffix tree built for a trie.

of their corresponding nodes in the suffix tree [**5**]. Hence we get that after a linear time preprocessing we can find the longest common suffix of any two codewords in constant time.

Combined with the previous section, this gives us the final result.

Theorem 6.1. *Fully LZW-compressed pattern matching for strings over a polynomial size integer alphabet can be solved in optimal linear time assuming the word RAM model.*

CHAPTER 7

Multiple Lempel-Ziv-Welch compressed patern matching

1. Overview of the algorithm

Our goal is to detect an occurrence of any of k patterns $p_1, p_2, \ldots, p_k$ in a given Lempel-Ziv-Welch compressed text $t[1\,..\,N]$. Let $M = \sum_{i=1}^{\ell} |p_i|$ be the total size of all patterns. Our final goal is to develop two algorithms working in $\mathcal{O}(n + M^{1+\epsilon})$ and $\mathcal{O}(n \log M + M)$ time. The main tool in both solutions is reducing the problem to simple purely geometrical questions on an integer grid. Those problems, while simple, seem interesting on their own. In particular, by extending one of the ideas we use to guarantee the above running times we are able to improve the bound on the so-called *binary dispatching problem* which we introduce in Appendix A.

Generalizing the notion of snippets introduced in Chapter 4, we say that a snippet is any substring of any pattern. At a high level our approach is the same as in the single pattern case described in Chapter 5: we reduce the original problem to multiple pattern matching in a collection of sequences of snippets. To solve the latter, we try to simulate the Knuth-Morris-Pratt algorithm on each of those sequences. Of course we cannot afford to process the snippets letter-by-letter, and hence must develop efficient procedures operating on whole fragments. To implement those procedures we work with purely geometric formulations of the original questions. In the next section we introduce a few simple structures. Then using those structures we show how to solve multiple pattern matching in a sequence of snippets in Section 3. Then in Section 4 we show how to reduce multiple pattern matching in LZW compressed text to multiple pattern matching in a sequence of snippets. By combining those two sections we get the claimed running times.

2. Basic structures

To prove the main theorem we need to design a few data structures. To simplify the exposition we use the notion of a $\langle f(M), g(M)\rangle$ *structure* meaning that after a $f(M)$ time preprocessing we are able to execute one query in $g(M)$ time. If such structure is *offline*, we are able execute a sequence of t queries in total $f(M)+tg(M)$ time. Similarly, a $\langle f(M), g(M)\rangle$ *dynamic structure* allows updates in $f(M)$ time and queries in $g(M)$ time. It is *persistent* if updating creates a new copy instead of modifying the original data.

We will extensively use the suffix tree S and the suffix array built for the concatenation of all patterns separated by a special character \$ (which does not occur in either the text nor any pattern, and is smaller than any original letter) which we call A:

$$A = p_1\$p_2\$\ldots\$p_{k-1}\$p_k$$

Similarly, S^r is the suffix tree built for the reversed concatenation A^r (for which we also build the suffix array):

$$A^r = p_k^r \$ p_{k-1}^r \$ \dots \$ p_2^r \$ p_1^r$$

Both suffix arrays are enriched with range minimum query structures enabling us to compute the longest common prefix and suffix of any two substrings in constant time.

Given a triple (i, j, k) denoting a substring $p_i[j\,..\,k]$ of the i-th pattern p_i, we would like to retrieve the corresponding (explicit or implicit) node in the suffix tree (or reversed suffix tree) efficiently. $\langle f(M), g(M)\rangle$ *locator* allows $g(M)$ time retrieval after a $f(M)$ time preprocessing.

Lemma 7.1. $\langle \mathcal{O}(M), \mathcal{O}(\log M)\rangle$ *locator exists.*

Proof. $\langle \mathcal{O}(M \log M), \mathcal{O}(\log M)\rangle$ is very simple to implement: for each vertex of the suffix tree we construct a balanced search tree containing all its ancestors sorted according to their depths. Constructing the tree for a vertex requires inserting just one new element into its parent tree (note that most standard balanced binary search trees can be made persistent so that inserting a new number creates a new copy and does not destroy the old one) and so the whole construction takes $\mathcal{O}(M \log M)$ time. This is too much by a factor of $\log M$, though. We use the standard micro-macro tree decomposition to remove it. The suffix tree is partitioned into small subtrees by choosing at most $\frac{M}{\log M}$ macro nodes such that after removing them we get a collection of connected components of at most logarithmic size. Such partition can be easily found in linear time. Then for each macro node we construct a binary search tree containing all its macro ancestors sorted according to their depths. There are just $\frac{M}{\log M}$ macro nodes so the whole preprocessing is linear. To find the ancestor v at depth d we first retrieve the lowest macro ancestor u of v by following at most $\log M$ edges up from v. If none of the traversed vertices is the answer, we find the macro ancestor of u of largest depth not smaller than d using the binary search tree in $\mathcal{O}(\log M)$ time. Then retrieving the answer requires following at most $\log M$ edges up from u. □

To improve the query time in the above lemma we need to replace the balanced search tree. $\langle f(M), g(M)\rangle$ *dynamic dictionary* stores a subset S of $\{0, \dots, M-1\}$ so that we can add or remove elements in $f(M)$ time, and check if a given x belongs to S (and if so, retrieve its associated information) or find its successor and predecessor in $g(M)$ time.

Lemma 7.2. $\langle \mathcal{O}(M^\epsilon), \mathcal{O}(1)\rangle$ *persistent dynamic dictionary exists for any* $\epsilon > 0$.

Proof. Choose an integer $k \geq \frac{1}{\epsilon}$. The idea is to represent the numbers in base $B = M^{\frac{1}{k}}$ and store them in a trie of depth k. At each vertex we maintain a table child$[0\,..\,B-1]$ with the i-th element containing a pointer to the corresponding child, if any. This allows efficient checking if a given x belongs to the current set (we just inspect at most k vertices and at each of them use the table to retrieve the next one in constant time). Note that we do not create a vertex if its corresponding tree is empty. To find the successor (or predecessor) efficiently, we maintain at each vertex two additional tables next$[0\,..\,B-1]$ and prev$[0\,..\,B-1]$ where next$[i]$ is the smallest $j \geq i$ such that child$[j]$ is defined and prev$[i]$ is the largest $j \leq i$ such that child$[j]$ is defined. Using those tables

the running time becomes $\mathcal{O}(k) = \mathcal{O}(1)$. Whenever we add or remove an element, we must recalculate the tables at all vertices from the traversed path. Its length is k and each table is of size B so the updates require $\mathcal{O}(kB) = \mathcal{O}(M^{1+\epsilon})$ time. Note that the whole structure is easily made persistent as after each update we create a new copy of the traversed path and do not modify any other vertices. □

Lemma 7.3. $\langle \mathcal{O}(M^{1+\epsilon}), \mathcal{O}(1) \rangle$ *locator exists for any* $\epsilon > 0$.

PROOF. The idea is the same as in Lemma 7.1: for each vertex of the suffix tree we construct a structure containing all its ancestors sorted according to their depths. Note that the depth are smaller than M so we can apply Lemma 7.2. The total construction time is $\mathcal{O}(M^{1+\epsilon})$ and answering a query reduces to one predecessor lookup. □

We assume the following preprocessing for both the suffix tree and the reversed suffix tree.

Lemma 7.4. *A suffix tree built for a text of length M can be preprocessed in linear time so that given an implicit or explicit vertex v we can retrieve its pre- and post-order numbers (*$\mathrm{pre}(v)$ *and* $\mathrm{post}(v)$*, respectively) in the uncompressed version of the tree in constant time.*

PROOF. We compute in linear time and store the number for each explicit vertex. Then given an implicit vertex v, we retrieve its lowest explicit ancestor v', and compute $\mathrm{pre}(v), \mathrm{post}(v)$ using $\mathrm{pre}(v'), \mathrm{post}(v')$ in constant time. □

3. Multiple pattern matching in a sequence of snippets

A high level description of the algorithm is given in MULTIPLE-PATTERN-MATCHING. prefixer and detector are low-level procedures which will be developed in the next section.

Note that instead of constructing the set P we could call detector(t, p_k) directly but then its implementation would have to be online, and that seems difficult to achieve in the $\langle \mathcal{O}(M), \mathcal{O}(\log M) \rangle$ variant.

Algorithm 6 MULTIPLE-PATTERN-MATCHING$(p_1, p_2, \ldots, p_n)$

```
1: P ← ∅
2: t ← p_1
3: for k = 2, 3, ..., n do
4:     add (t, p_k) to P
5:     t ← prefixer(t, p_k)
6: end for
7: for all (s_1, s_2) ∈ P do
8:     detector(s_1, s_2)
9: end for
```

A $\langle f(M), g(M) \rangle$ *prefixer* is a data structure which preprocesses the collection of patterns in $f(M)$ time so that given any two snippets we can compute the longest suffix of their concatenation which is a prefix of some pattern in $g(M)$ time.

Lemma 7.5. $\langle \mathcal{O}(M), \mathcal{O}(\log M) \rangle$ *prefixer exists.*

PROOF. Let the two snippets be s_1 and s_2. First note that using Lemma 3.4 we can compute the longest common prefix of $s_2^r s_1^r$ and a given suffix of A^r in constant time. Hence we can apply binary search to find the (lexicographically) largest suffix of A^r which either begins with $s_2^r s_1^r$ or is (lexicographically) smaller in $\mathcal{O}(\log M)$ time. Given this suffix $A^r[i\,..\,|A^r|]$ we compute $\ell = |\mathrm{LCP}(|s_s^r s_1^r|, A^r[i\,..\,|A^r|])|$ and apply Lemma 7.1 to retrieve the ancestor v of $A^r[i\,..\,|A^r|]$ at depth ℓ in $\mathcal{O}(\log M)$ time. The longest prefix we are looking for corresponds to an ancestor u of v which has at least one outgoing edge starting with \$. Observe that such u must be explicit as there are no \$ characters on the root-to-v path. This means that we can apply a simple linear time preprocessing to compute such u for each possible explicit v in linear time. Then given a (possibly implicit) v we use the preprocessing to compute the u corresponding to the longest prefix in constant time, giving a $\mathcal{O}(\log M)$ total query time. □

Lemma 7.6. *$\langle \mathcal{O}(M^{1+\epsilon}), \mathcal{O}(1) \rangle$ prefixer exists for any $\epsilon > 0$.*

PROOF. For each pattern p_i we consider all possibilities to cut it into two parts $p_i = p_i[1\,..\,j]p_i[j+1\,..\,|p_i|]$. For each cut we locate vertex u corresponding to $p_i[1\,..\,j]$ in the reversed suffix tree and v to $p_i[j+1\,..\,|p_i|]$ in the suffix tree. By Lemma 7.3 it takes constant time and by Lemma 7.4 we can then compute $\mathrm{pre}(u)$, $\mathrm{pre}(v)$ and $\mathrm{post}(v)$. Then we add a horizontal segment $\{\mathrm{pre}(u)\} \times [\mathrm{pre}(v), \mathrm{post}(v)]$ with weight j to the collection. Now consider a query concerning two snippets s_1 and s_2. First locate the vertex u corresponding to s_1 in the reversed suffix tree and v to s_2 in the suffix tree. Then construct a vertical segment $[\mathrm{pre}(u), \mathrm{post}(u)] \times \{\mathrm{pre}(v)\}$ and observe that the query reduces to finding the heaviest horizontal segment in the collection it intersects (if there is none, we retrieve the lowest ancestor of v which has an outgoing edge starting with \$, which can be precomputed in linear time), see Figure 1. Additionally, the horizontal segments are either disjoint or contained in each other. In the latter case, weight of the longer segment is larger than weight of the shorter. To this end we prove that there exists a $\langle \mathcal{O}(M^{1+\epsilon}), \mathcal{O}(1) \rangle$ structure for computing the heaviest horizontal segment intersected by a given vertical segment in such collection on a $M^2 \times M^2$ grid.

We sweep the grid from left to right maintaining a structure describing currently active horizontal segments. The structure is based on the idea from Lemma 7.3 with $k \geq \frac{2}{\epsilon}$. Each leaf corresponds to a different y coordinate and stores all active horizontal segments with this coordinate on a stack, with the most recently encountered segment on the top (because weights of intersecting segments are monotone with their lengths, it is also the heaviest segment). Each inner vertex stores a table $\mathrm{heaviest}[0\,..\,M^{\frac{2}{k}}]$ with the i-th element containing the maximum weight in the subtree corresponding to the i-th leaf, if any. Additionally, a range minimum query structure RMQ(heaviest) is stored so that given any two indices i, j we can compute the maximum $\mathrm{heaviest}[k]$ over all $k \in \{i, i+1, \ldots, j\}$ in constant time. Adding or removing an active segment requires locating the corresponding stack and either pushing a new element or removing the topmost element. Then we must update the tables at all ancestors of the corresponding leaf, which by Lemma 3.5 takes $\mathcal{O}(kM^{\frac{2}{k}}) = \mathcal{O}(M^{\epsilon})$ time. Given a query, we first locate the appropriate version of the structure. Then we traverse the trie and find the heaviest intersected segment by asking at most $2k$ range minimum queries. □

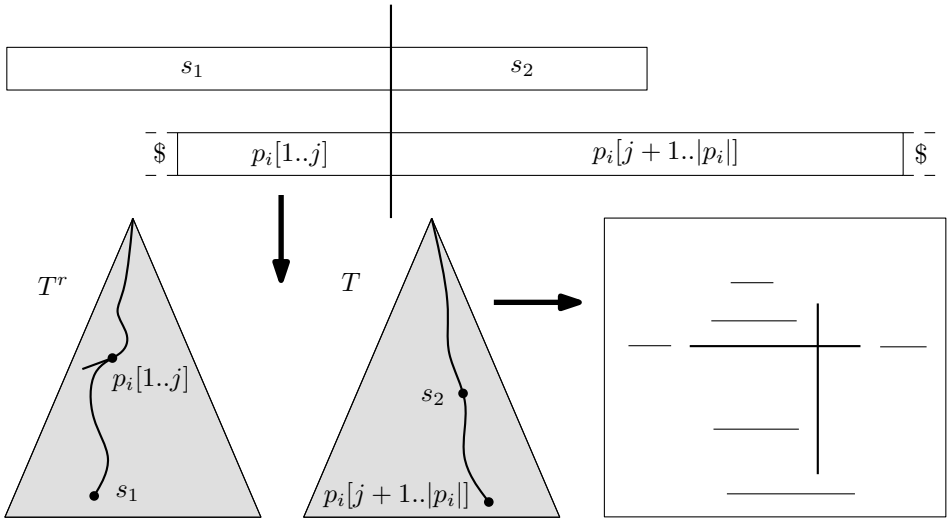


FIGURE 1. Reducing prefixer queries to segments intersection.

A $\langle f(M), g(M)\rangle$ *detector* is a data structure which preprocesses the collection of patterns in $f(M)$ time so that given any two snippets we can detect an occurrence of a pattern in their concatenation in $g(M)$ time. Both implementation that we are going to develop are based on the same idea of reducing the problem to a purely geometric question on a $M \times M$ grid, similar to the one from Lemma 7.6. For each pattern p_i we consider all possibilities to cut it into two parts $p_i = p_i[1\,..\,j]p_i[j+1\,..\,|p_i|]$. For each cut we locate in constant time vertex u corresponding to $p_i[1\,..\,j]$ in the reversed suffix tree and v to $p_i[j+1\,..\,|p_i|]$ in the suffix tree. If both u and v are explicit vertices, we add a rectangle $[\mathrm{pre}(u), \mathrm{post}(u)] \times [\mathrm{pre}(v), \mathrm{post}(v)]$ to the collection. Then given two snippets s_1 and s_2 detecting an occurrence in their concatenation reduces in constant time to retrieving any rectangle containing $(\mathrm{pre}(u), \mathrm{pre}(v))$ where u is the vertex corresponding to s_1 in the reversed suffix tree and v to s_2 in the suffix tree, see Figure 2. Note that the x and y projections of any two rectangles in the collection are either disjoint or contained in each other. Assuming no pattern occurs in another, no two rectangles are contained in each other. We call a collection with such two properties *valid*.

Lemma 7.7. $\langle \mathcal{O}(M), \mathcal{O}(\log M)\rangle$ *offline detector exists.*

PROOF. Recall that in the offline version we are given all queries in an advance. We sweep the grid from left to right maintaining a structure describing the currently intersected rectangles. At a high level the structure is just a full binary tree on M leaves corresponding to different y coordinates (and each inner vertex corresponding to an continuous interval of y coordinates). If we aim to achieve logarithmic time of both update and query, the implementation is rather straightforward. We want to achieve constant time update, though. Say that we encounter a new rectangle and need to insert an interval $[y_1, y_2]$ with $y_1 < y_2$ into the structure. We compute the lowest common ancestor v of the leaves corresponding to y_1 and y_2 in the tree (as the tree is full there exists a simple arithmetic formula for that) and call v *responsible* for $[y_1, y_2]$.

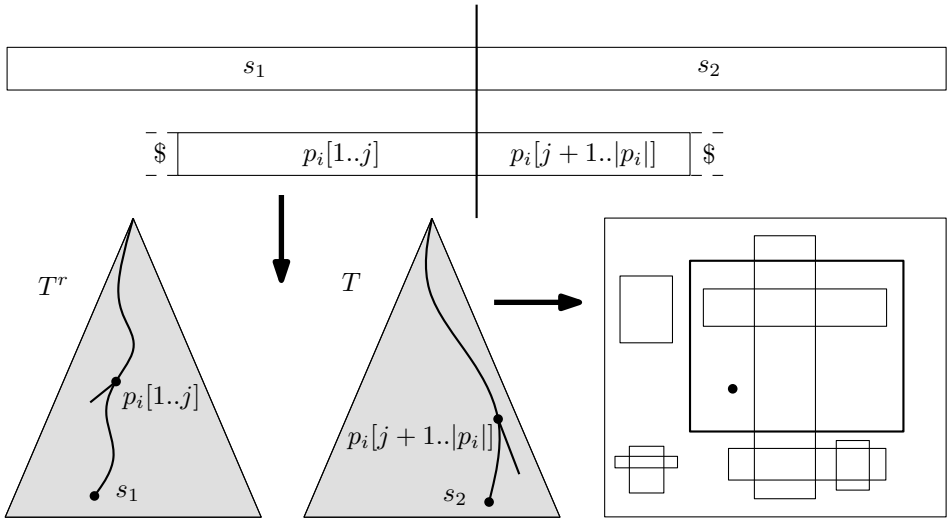


FIGURE 2. Reducing detector queries to rectangles retrieval in a valid collection.

v corresponds to an interval $\big[\alpha 2^\ell, (\alpha+2)2^\ell\big)$ such that $y_1 \in \big[\alpha 2^\ell, (\alpha+1)2^\ell\big)$ and $y_2 \in \big[(\alpha+1)2^\ell, (\alpha+2)2^\ell\big)$. For each inner vertex we store its *interval stack*. To insert $[y_1, y_2]$ we simply push it on the interval stack of the responsible vertex. Note that because the collection is valid, all intervals $I_1, I_2, \ldots, I_k$ stored on the same interval stack at a given moment are nested, i.e., $I_1 \subseteq I_2 \subseteq \ldots \subseteq I_k$. To remove an interval we locate the responsible vertex and pop the topmost element from its interval stack. The only nontrivial part is detecting an interval containing a given point x. First traverse the path starting at the corresponding leaf. This gives us a sequence of $\log M$ interval stacks. Observe that for a fixed interval stack it is enough to check if its top element (if any) contains x, hence $\mathcal{O}(\log M)$ query time follows. □

The idea if responsible vertices from the above lemma, while very simple, allows us to improve the result of Alstrup *et al.*, who in turn improved an algorithm given by Ferragina and Muthikrishnan [**18**]. They worked with the following bridge color formulation: given a tree T on n vertices, and a collection of m *bridges*, which are simply pairs of vertices (u, v), preprocess them so that given any two vertices u' and v' we can find the lowest bridge (u, v) such that u' belongs to the subtree of u and v' belongs to the subtree of v, where (u, v) is lower than (u', v') if u is a descendant of u' and v is a descendant of v'. In case there is no unique lowest bridge, we have to signal ambiguity. It is known that a $\mathcal{O}(\log m)$ query time is possible after constructing a structure of size $\mathcal{O}(m \log m)$ [**18**], and that the space can be improved to linear while retaining the same query bound [**2, 40**]. Unfortunately, the construction time of the latter structure is as big as expected $\mathcal{O}(m(\log\log m)^2)$, and the algorithm is quite involved. In Appendix A we show how to improve it to linear.

Lemma 7.8. *$\big\langle \mathcal{O}(M^{1+\epsilon}), \mathcal{O}(1) \big\rangle$ detector exists for any $\epsilon > 0$.*

PROOF. We sweep the grid from left to right maintaining a structure describing currently intersected rectangles. The structure should allow adding or removing intervals from $\{0, 1, \ldots, M-1\}$ and retrieving any interval containing a specified point. A straightforward use of the idea from Lemma 7.2 allows an efficient implementation of those operations in $\mathcal{O}(M^\epsilon)$ and $\mathcal{O}(1)$ time, respectively. □

By plugging either Lemma 7.5 and Lemma 7.7 or Lemma 7.6 and Lemma 7.8 into MULTIPLE-PATTERN-MATCHING we get the main theorem.

Theorem 7.1. *Multiple pattern matching in a sequence of n snippets can be performed in $\mathcal{O}(n \log M + M)$ or $\mathcal{O}(n + M^{1+\epsilon})$ time, where M is the combined size of all patterns.*

4. LZW compression

We are given a sequence of blocks, each block being either a single letter, or a previously defined block concatenated with a single letter. For each block we would like to check if the corresponding word occurs in any of the patterns, and if not, we would like to find its longest suffix (prefix) which is a prefix (suffix) of any of the patterns. First we consider all blocks at once and for each of them compute its longest prefix which occurs in some p_i.

Lemma 7.9. *Given a LZW compressed text we can compute for all blocks the corresponding snippet (if any) and the longest prefix which is a suffix of some pattern in total linear time.*

PROOF. The idea is the same as in the single pattern case from Lemma 5.9: intersect the suffix tree and the trie defined by all blocks at once. □

To compute the longest suffix which is a prefix of some pattern, we would like to use the Aho-Corasick automaton built for all $p_1, p_2, \ldots, p_k$, which is a standard multiple pattern matching tool [**1**]. Recall that its state set consists of all unique prefixes $p_i[1 \ldots j]$ organized in a trie. Additionally, each v stores the so-called *failure link* failure(v), which points to the longest proper suffix of the corresponding word which occurs in the trie as well. If the alphabet is of constant time, we can afford to build and store the full transition function of such automaton. If the alphabet is of non-constant size, without losing the generality we can assume that it consists of integers $\{0, 1, \ldots, M-1\}$. In such case it is not clear if we can afford to store the full transition function. Nevertheless, storing the trie and all failure links are enough to navigate in amortized constant time per letter. This is not enough for our purposes, though, as we need a worst case bound. We start with building the trie and computing the failure links. This is trivial to perform in linear time after constructing the reversed suffix tree: each state is a (implicit or explicit) node of the tree with an outgoing edge starting with \$. Its failure link is simply the lowest ancestor corresponding to such node as well. The depending on the preprocessing allowed we get two time bounds.

Lemma 7.10. *Given a LZW compressed text we can compute for all blocks the longest suffix which is a prefix of some pattern in total time $\mathcal{O}(n + M^{1+\epsilon})$ for any $\epsilon > 0$.*

PROOF. At each vertex we create a $\langle \mathcal{O}(M^{1+\epsilon}), \mathcal{O}(1) \rangle$ persistent dynamic dictionary. To create the dictionary for v we take the dictionary stored at failure(v) and update it

by inserting all edges outgoing from v. There are at most M updates to all dictionaries, each of them taking $\mathcal{O}(M^\epsilon)$ time, and then any query is answered in constant time, resulting in the claimed bound. □

Lemma 7.11. *Given a LZW compressed text we can compute for all blocks the longest suffix which is a prefix of some pattern in total time* $\mathcal{O}(n \log M + M)$.

Proof. For a vertex v consider the sequence of its ancestors failure(v), failure$^2(v)$, failure$^3(v), \ldots$. To retrieve the transition $\delta(v, c)$ we should find the first vertex in this sequence having an outgoing edge starting with c. For each different character c we build a separate structure $S(c)$ containing all intervals $[\text{pre}(v), \text{post}(v)]$ for v having an outgoing edge starting with c, where pre(v) and post(v) are the pre- and post-order numbers of v in a tree defined by the failure links (i.e., failure(v) is the parent of v there). Then to to calculate $\delta(v, c)$ we should locate the smallest interval containing pre(v) in $S(c)$. By implementing $S(v)$ as a balanced search tree we get the claimed bound. □

By adding Lemma 7.9 and either Lemma 7.11 or Lemma 7.10 to Theorem 7.1 we get the claimed total running time of the whole solution.

Theorem 7.2. *Multiple pattern matching in LZW compressed texts can be performed in* $\mathcal{O}(n \log M + M)$ *or* $\mathcal{O}(n + M^{1+\epsilon})$ *time, where* M *is the combined size of all patterns and* n *size of the compressed representation.*

CHAPTER 8

Lempel-Ziv compressed pattern matching

1. Overview of the algorithm

Our goal is to detect an occurrence of p in a given Lempel-Ziv compressed text $t[1\,..\,N]$. The Lempel-Ziv representation is quite difficult to work with efficiently, even for a such seemingly simple task as extracting a single letter. The starting point of our algorithm is thus transforming the input into a *straight-line program*, which is a context-free grammar with each nonterminal generating exactly one string. For that we use the method of Charikar *et al.* [**10**] to construct a SLP of size $\mathcal{O}(n \log \frac{N}{n})$ with additional property that all productions are *balanced*, meaning that the right sides are of the form XY with $\frac{\alpha}{1-\alpha} \leq \frac{|X|}{|Y|} \leq \frac{1-\alpha}{\alpha}$ for some constant α, where $|X|$ is the length of the (unique) string generated by X. Note that Rytter gave a much simpler and cleaner algorithm [**42**] with the same size guarantee, using the so-called AVL grammars, but we need the grammar to be balanced. We also need to add a small modification to allow self-referential LZ.

After transforming the text into a balanced SLP, for each nonterminal we try to check if the string it represents occurs inside p, and if so, compute the position of (any) its occurrence. If it does not, we would like to compute the longest prefix (suffix) of this string which is a suffix (prefix) of p. At first glance this might seem like a different problem that the one we promised to solve: instead of locating an occurrence of the pattern in the text, we retrieve the positions of fragments of the text in the pattern. Nevertheless, solving it efficiently gives us enough information to answer the original question due to a constant time procedure which detects an occurrence of p in a concatenation of two its substrings.

The first (simple) algorithm for processing a balanced SLP we develop requires as much as $\mathcal{O}(\log m)$ time per query, which results in $\mathcal{O}(n \log \frac{N}{n} \log m + m)$ total complexity. This is clearly not enough to beat [**17**] on all possible inputs. Hence instead of performing the computation for each nonterminal separately, we try to process them in $\mathcal{O}(\log N)$ groups corresponding to the (truncated) logarithm of their length. Using the fact that the grammar is balanced, we are then able to achieve $\mathcal{O}(n \log \frac{N}{n} + m \log m)$ time. Because of some technical difficulties, in order to decrease this complexity we cannot really afford to check if the represented string occurs in p for each nonterminal exactly, though. Nevertheless, we can compute some approximation of this information, and by using a tailored variant of binary search applied to all nonterminals in a single group at once, we manage to process the whole grammar in time proportional to its size while adding just $\mathcal{O}(m)$ to the running time.

2. Constructing balanced grammar

Recall that a LZ parse is a sequence of triples $(start_i, len_i, next_i)$ for $i = 1, 2, \ldots, n$. In the not self-referential variant considered in [**10**], we require that $start_i + len_i - 1 \leq \sum_{j<i} len_j$ so that each triple refers only to the prefix generated so far. Although such assumption is made by some LZ-based compressors, [**17**] deals with the compressed pattern matching problem in its full generality, allowing self-references. Thus for the sake of completeness we need to construct a balanced grammar from a potentially self-referential LZ parse. It turns out that a small modification of a known method is enough for this task.

Lemma 8.1 (see Theorem 1 of [**10**])**.** *Given a (potentially self-referential) LZ parse of size n, we can build a α-balanced SLP of size $\mathcal{O}(n \log \frac{N}{n})$ describing the same string of length N, for any constant $0 < \alpha \leq 1 - \frac{\sqrt{2}}{2}$. Running time of the construction is proportional to the size of the output.*

Proof. At a very high level, the idea of [**10**] is to process the parse from left-to-right. When processing a triple $(start_i, len_i, next_i)$, we already have an α-balanced SLP describing the prefix of the whole text corresponding to the previously encountered triples. Because the grammar is balanced, we can define $t[start_i \,..\, start_i + len_i - 1]$ by introducing a relatively small number of new nonterminals (with small actually meaning small in the amortized sense). Now if we allow the parse to be self-referential, it might happen that $t[start_i \,..\, start_i + len_i - 1]$ sticks out from the right end of $t[1 \,..\, \sum_{j=1}^{i-1} len_j]$. In such case we do as follows: let $L = \sum_{j=1}^{i-1} len_j$, and split the fragment corresponding to the current triple into three parts. First we have $t[start_i \,..\, L]$, then some repetitions of the same fragment, and then $t[start_i \,..\, start_i - 1 + len_i \bmod (L - start_i + 1)]$ followed by a single letter $next_i$. After defining a nonterminal deriving $t[start_i \,..\, L]$, we can define a nonterminal deriving the repetitions at the expense of introducing at most $2 \log len_i$ new nonterminals. Then we define a nonterminal deriving $t[start_i \,..\, start_i - 1 + len_i \bmod (L - start_i + 1)]next_i$. The only change in the analysis of this method is that we might end up adding $\sum_{i=1}^{n} \log len_i$ new nonterminals, which by the concavity of log is at most $\mathcal{O}(n \log \frac{N}{n})$, and thus does not change the asymptotic upper bound. Note that the authors of [**10**] were not concerned with the computational complexity of their algorithm. Nevertheless, it is easy to see that the only place which cannot be amortized by the number of new nonterminals is finding the corresponding place at the so-called active symbols list and traversing the grammar top-down in order to find the appropriate nonterminal. The former can be implemented by storing the active list in a balanced search tree, adding $\mathcal{O}(n \log n)$ to the time. The latter adds just $\mathcal{O}(n \log N)$ to the whole running time. Hence we can implement the whole method in $\mathcal{O}(n \log N)$. In order to decrease this complexity to just $\mathcal{O}(n \log \frac{N}{n})$, we cut the string into n parts of roughly the same size. Note that this requires that our computational model allows constant time integer division.

Note that the algorithm in [**10**] contains one special case: if the compression ratio is at most $2e$, the trivial grammar is returned. We do the same. □

As a result we get a context-free grammar in which all nonterminals derive exactly one string, and right sides of all productions are of the form XY with $\frac{\alpha}{1-\alpha} \leq \frac{|X|}{|Y|} \leq \frac{1-\alpha}{\alpha}$.

The exact value of α is not important, we only need the fact that both $\frac{|X|}{|Y|}$ and $\frac{|Y|}{|X|}$ are bounded from above. For the sake of concreteness we assume $\alpha = 0.25$. We also need to compute $|X|$ for each nonterminal X, and to group the nonterminals according to the (rounded down) logarithm of their length, with the base of the logarithm to be chosen later. Note that taking logarithms of large numbers (i.e., substantially longer than $\log n$ bits) is not necessarily a constant time operations in our model. We can use the fact that the grammar is balanced here: if $X \to YZ$, then $\log_b |X| \leq \beta + \max(\log_b |Y|, \log_b |Z|)$ for some constant β depending only on α and b, and the logarithms can be computed for all nonterminals in a bottom-up fashion using just linear time.

3. Processing balanced grammar

While the final goal of this section is a $\mathcal{O}(n \log \frac{N}{n} + m)$ time algorithm, we start with a simple $\mathcal{O}(n \log \frac{N}{n} \log m + m)$ time solution, which then is modified to take just $\mathcal{O}(n \log \frac{N}{n} + m \log m)$, and finally $\mathcal{O}(n \log \frac{N}{n} + m)$ time.

For each nonterminal X we would like to check if the string it represents occurs inside p. If it does not, we would like to compute $\text{prefix}(X)$ and $\text{suffix}(X)$, the longest prefix (suffix) which is a suffix (prefix) of the whole p. Given such information for all possible nonterminals, we can easily detect an occurrence.

Lemma 8.2. *If p occurs in a string represented by a SLP then there exists a production $X \to YZ$ such that p occurs in* $\text{suffix}(Y)\,\text{prefix}(Z)$.

Proof. Consider the leftmost occurrence of p. Take the starting symbol $X = S$ and its production $X \to YZ$. If the leftmost occurrence is completely inside Y or Z, repeat with X replaced with Y or Z. Otherwise the occurrence crosses the boundary between Y and Z, in other words there is a prefix snippet $p[1\,..\,i]$ ending Y and a suffix snippet $p[i+1\,..\,m]$ starting Z. Then $|\text{suffix}(Y)| \geq i$ and $|\text{prefix}(Z)| \geq m - i$, and p occurs in $\text{suffix}(Y)\,\text{prefix}(Z)$. □

Lemma 8.3. *Given a (potentially self-referential) Lempel-Ziv parse of size n describing a text $t[1\,..\,N]$ and a pattern $p[1\,..\,m]$, we can detect an occurrence of p inside t deterministically in time $\mathcal{O}(n \log \frac{N}{n} \log m + m)$.*

Proof. By Lemma 8.1 and Lemma 8.2, we only have to compute for each nonterminal X its corresponding snippet (if any) and both $\text{prefix}(X)$ and $\text{suffix}(X)$. We process the productions in a bottom-up order. Assume that we have the information concerning Y and Z available and would like to process $X \to YZ$. If both Y and Z correspond to substrings of p, we can apply binary search in the suffix array to check if their concatenation does as well in $\mathcal{O}(\log m)$ steps, each step consisting of two applications of Lemma 3.4 used to compare the concatenation with a suffix of p. To compute $\text{prefix}(X)$ and $\text{suffix}(X)$ in $\mathcal{O}(\log m)$ time we could use Lemma 4.4. There is one difficulty here, though: we need to know the corresponding node in the suffix tree. This is exactly the same problem that we encountered in Chapter 7. Recall that we developed the so-called $\langle \mathcal{O}(m), \mathcal{O}(\log m) \rangle$ locator in Lemma 7.1 which after a linear time preprocessing allowed retrieving the (implicit or explicit) node corresponding to a given snippet in $\mathcal{O}(\log m)$ time. We use the same method here. □

We would like to remove the $\log m$ factor from the above complexity. It seems that the main difficulty here is that we need to implement a procedure for detecting if a concatenation of two substrings of p occurs in p as well, and in order to get the claimed running time we would need to answer such queries in constant time after a linear (or close to linear) preprocessing. We overcome this obstacle by choosing to work with an approximation of this information instead and using the fact that the grammar we are working with is balanced.

Definition 8.1. A cover of a nonterminal X is pair of snippets $p[i\,..\,i+2^k-1]$ and $p[j\,..\,j+2^k-1]$ such that $2^k < |X| \leq 2^{k+1}$, $p[i\,..\,i+2^k-1]$ is a prefix of the string represented by X, and $p[j\,..\,j+2^k-1]$ is a suffix of the string represented by X. We call k the order of X's cover.

We try to find the cover of each nonterminal X. If there is none, we know that the string it represents does not occur inside p. In such case we compute $\text{prefix}(X)$ and $\text{suffix}(X)$. More precisely, we either:

(1) compute the cover, in such case the string represented by X might or might not occur in p,
(2) do not compute the cover, in such case the string represented by X does not occur in p.

As the lemma below shows, it is possible to extract $\text{prefix}(X)$ and $\text{suffix}(X)$ from the cover of X using Lemma 4.4 in constant time, and the information about $\text{prefix}(X)$ and $\text{suffix}(X)$ for each nonterminal X is enough to detect an occurrence.

Lemma 8.4. *Given the covers of Y and Z, we can compute* $\text{prefix}(X)$ *and* $\text{suffix}(X)$ *in constant time as long as $\frac{|Y|}{|Z|}$ and $\frac{|Z|}{|Y|}$ are bounded from above by a constant and for any snippet $p[i\,..\,i+2^k-1]$ we can retrieve the corresponding node in the suffix tree in constant time. To compute* $\text{prefix}(X)$ *we can use* $\text{prefix}(Z)$ *instead of the cover of Z, and to compute* $\text{suffix}(X)$ *we can use* $\text{suffix}(Y)$ *instead of the cover of Y.*

Proof. It is enough to consider $\text{prefix}(X)$. The idea is to use a few application of Lemma 4.4 with carefully chosen arguments, see Figure 1. More specifically, let a, b and c, d be the covers of Y and Z, respectively. First we locate the vertex corresponding to d in the suffix tree, then:

(1) apply Lemma 3.8 to compute $\text{prefix}(d)$ if we have the cover of Z, otherwise take the known $\text{prefix}(Z)$ and go to (3),
(2) apply Lemma 4.4 to c and $\text{prefix}(d)$ without the first $|c|+|d|-|Z|$ letters to get prefix_1,
(3) apply Lemma 4.4 to b and prefix_1 to get prefix_2,
(4) apply Lemma 4.4 to a and prefix_2 without the first $|a|+|b|-|Y|$ letters to get the desired answer prefix_3.

Note that whenever we apply the lemma to two words u and v, $|v|$ is a power of 2 and so we can use Lemma 8.7 to locate its corresponding node in constant time. Also, it holds that $|u| \geq \frac{\min(|Y|,|Z|)}{2}$ and $|v| \leq |Y|+|Z|$ and so the running time is bounded by:

$$\max\left(1, \log\frac{|v|}{|u|}\right) \leq \max\left(1, \log\left(\frac{|Y|+|Z|}{\min(|Y|,|Z|)}\right)\right) = \log\left(1+\frac{\max(|Y|,|Z|)}{\min(|Y|,|Z|)}\right)$$

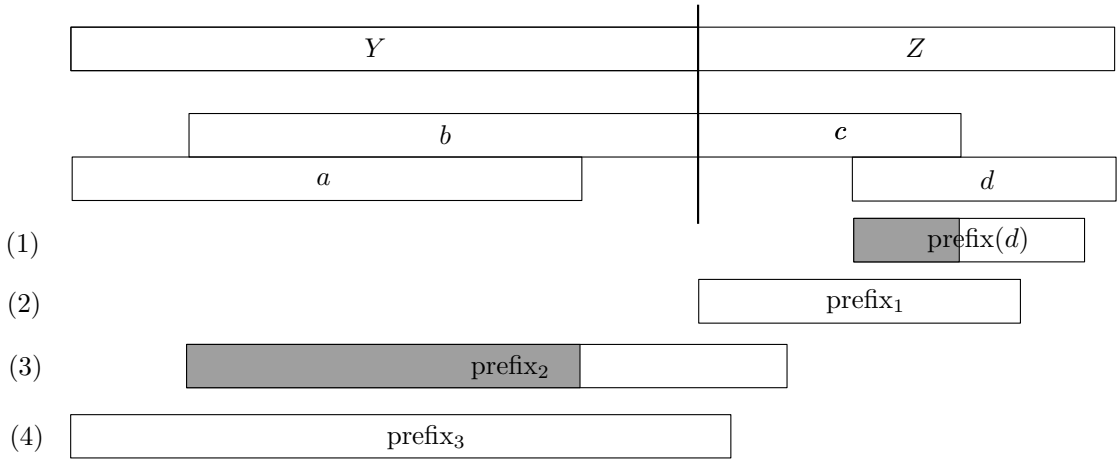


FIGURE 1. Computing prefix(X) given the covers of Y and Z.

which is $\mathcal{O}(1)$. □

To find the covers we process the nonterminals in groups. Nonterminals in the ℓ-th group $\mathcal{G}_\ell = \{X_1, X_2, \dots X_s\}$ are chosen so that $(\frac{4}{3})^\ell < |X_i| \leq (\frac{4}{3})^{\ell+1}$. The groups are disjoint so $\sum_\ell |\mathcal{G}_\ell| = \mathcal{O}(n \log \frac{N}{n})$. Furthermore, the partition can be constructed in linear time. We start with computing the covers of nonterminals in $\mathcal{G}_1$ naively. Then we assume that all nonterminal in $\mathcal{G}_{\ell-1}$ are already processed, and we consider $\mathcal{G}_\ell$. Because the grammar is 0.25-balanced, if $X_i \to Y_i Z_i$ then $|Y_i|, |Z_i| \leq \frac{3}{4}|X_i|$, and Y_i, Z_i belong to already processed $\mathcal{G}_{\ell'}$ with $\ell - 5 \leq \ell' < \ell$. If for some Y_i or Z_i we do not have the corresponding cover, neither must have the corresponding X_i, so we use Lemma 4.4 to calculate prefix(X_i), suffix(X_i), and remove X_i from $\mathcal{G}_\ell$. For all remaining X_i we are left with the following task: given the covers of Y_i and Z_i, compute the cover of X_i, or detect that the represented string does not occur in p and so we do not need to compute the cover. Note that the known covers are of order k with $k_{min} = \lfloor \ell \log \frac{4}{3} \rfloor - 3 \leq k \leq \lceil \ell \log \frac{4}{3} \rceil = k_{max}$.

We reduce computing covers to a sequence of batched queries of the form: given a sequence of pairs of snippets $p[i \,..\, i + 2^{k_1} - 1]$, $p[j \,..\, j + 2^{k_2} - 1]$ does their concatenation occur in p, and if so, what is the corresponding snippet? We call this merging the pair. For each ℓ we will require solving a constant number of such problems with $k_{min} \leq k_1, k_2 \leq k_{max}$, each containing $\mathcal{O}(|\mathcal{G}_\ell|)$ queries. We call this problem BATCHED-POWERS-MERGE. Before we develop an efficient solution for such question, lets see how it can be used to compute covers.

Lemma 8.5. *Computing covers of the nonterminals in any $\mathcal{G}_\ell$ can be reduced in linear time to a constant number of calls to* BATCHED-POWERS-MERGE*, with the number of pairs in each call bounded by $|\mathcal{G}_\ell|$.*

PROOF. Recall that for each given pair of snippets we have their covers available, and the orders of those covers are from $\{k, k+1, \dots, k+4\}$. Consider the situation for a single pair, see Figure 2. Let a, b be the cover of the first snippet and c, d the cover of the

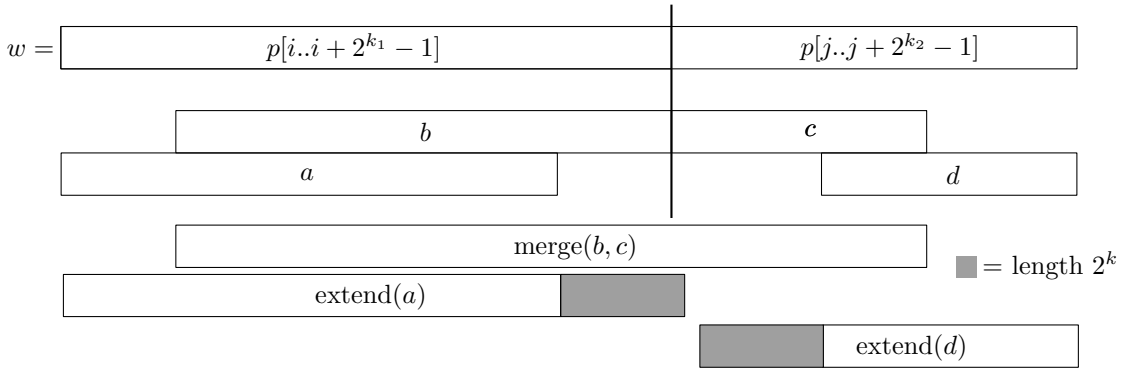


FIGURE 2. Computing cover of a pair of snippets.

second snippet. First we merge b and c to get merge(b, c). Then we extend a to the right and d to the left by merging with the corresponding fragments of merge(b, c) of length 2^k, and call the results extend(a) and extend(d). Then we would like iteratively extend both a and d with fragments of such length as long as it does not result in sticking out of the considered word w. To do that, we need to have the snippets corresponding to those fragments available. Consider the situation for a: first we extract the snippets from merge(b, c), then from extend(d). We claim that we are always able to perform such extraction: if the next 2^k characters fall outside merge(b, c), the distance to the left boundary of d does not exceed 2^k and thus we can use extend(d). If during this extending procedure the merging fails, the pair does not represent a substring of p. Otherwise we get the snippet corresponding to the prefix and suffix of w of lengths $|w| - |w| \bmod 2^k$, which allows us to extract the prefix and suffix of length $2^{k'}$ where $2^{k'} < |w| \leq 2^{k'+1}$, because $k \leq k'$.

To finish the proof, note that for a single pair we need a constant number of merges. Thus we can do the merging in parallel for all pairs in a constant number of calls to BATCHED-POWERS-MERGE. □

Now we only have to develop the algorithm for BATCHED-POWERS-MERGE. A simple solution would be to do a binary search in the suffix array built for p for each pair separately: we can compare $p[i\,..\,i + 2^{k_1} - 1]p[j\,..\,j + 2^{k_2} - 1]$ with any suffix of p in constant time using at most two longest common prefix queries so a single search takes $\mathcal{O}(\log m)$ time, which gets us back to the bounds from Theorem 8.3. In order to get a better running time we aim to exploit the fact that we are given many pairs at once. First observe that we can order all concatenations from a single problem efficiently.

Lemma 8.6. *Given $\mathcal{O}(|\mathcal{G}_\ell|)$ pairs of words of the form $p[i\,..\,i+2^{k_1}-1]$, $p[j\,..\,j+2^{k_2}-1]$ with $k_{min} \leq k_1, k_2 \leq k_{max}$ we can lexicographically sort their concatenations in time $\mathcal{O}(|\mathcal{G}_\ell| + m^c)$ if $|k_{max} - k_{min}| \in \mathcal{O}(1)$.*

PROOF. We split the words to be sorted into a constant number of chunks of length $2^{k_{min}}$. Then we would like to assign numbers to those chunks so that $\mathrm{nr}(p[i\,..\,i+2^{k_{min}}-1]) < \mathrm{nr}(p[j\,..\,j+2^{k_{min}}-1])$ iff $p[i\,..\,i+2^{k_{min}}-1]) <_{lex} p[j\,..\,j+2^{k_{min}}-1])$. To compute all $\mathrm{nr}(p[i\,..\,i + 2^{k_{min}} - 1])$ we retrieve the positions of $p[i\,..\,m]$ in the suffix array. Then

we sort the resulting list of $\mathcal{O}(|\mathcal{G}_\ell|)$ integers using radix sort, i.e., by $\frac{1}{\epsilon}$ rounds of counting sort. The time required by this sorting is linear plus $\mathcal{O}(m^\epsilon)$. After sorting we scan the list and identify different suffixes with the same prefix of length $2^{k_{min}}$, such suffixes belong to continuous blocks whose boundaries can be identified using longest prefix queries. Then the original task reduces to sorting a list of constant length vectors consisting of integers not exceeding m, which can be done efficiently using radix sort. □

We apply the above lemma to all calls to Batched-powers-merge corresponding to nonempty $\mathcal{G}_\ell$. If $(\frac{4}{3})^\ell > m$ then clearly the corresponding $\mathcal{G}_\ell$ is empty, so the total running time of this part is just $\mathcal{O}(m^\epsilon \log m + \sum_\ell |\mathcal{G}_\ell|) = \mathcal{O}(m + n \log \frac{N}{n})$. Now that the queries from a single call to Batched-powers-merge are sorted, instead of performing a separate binary search for each of them we can scan the queries and the suffix array at once, resulting in a $\mathcal{O}(|\mathcal{G}_\ell| + m)$ running time for each different ℓ. As after a $\mathcal{O}(m \log m)$ preprocessing we can compute the corresponding node in the suffix tree for all snippets $p[i \mathinner{..} i + 2^k - 1]$ used in Lemma 8.4, this gives us the following total running time.

Theorem 8.1. *Given a (potentially self-referential) Lempel-Ziv parse of size n describing a text $t[1 \mathinner{..} N]$ and a pattern $p[1 \mathinner{..} m]$, we can detect an occurrence of p inside t deterministically in time $\mathcal{O}(n \log \frac{N}{n} + m \log m)$.*

This is still not enough to improve [**17**] on all possible inputs. We would like to replace $m \log m$ by m in the above complexity by focusing on improving the running time of Batched-powers-merge. At a high level the idea is to consider the queries in a single call in sorted order, and for each of them perform a binary search starting where the lexicographically previous pair was found. This might be still too slow though. To accelerate the search we develop a constant time procedure for locating the fragment of the suffix array corresponding to all occurrences of any $p[i \mathinner{..} i + 2^k - 1]$. This procedure will be also used in Lemma 8.4.

Lemma 8.7. *The pattern p can be processed in linear time so that given any $p[i \mathinner{..} i + 2^k - 1]$ we can compute its first and last occurrence in the suffix array of p in constant time.*

Proof. It is enough to show that the suffix tree S built for p can be preprocessed in linear time so that we can locate the (implicit or explicit) vertex corresponding to any fragment which is a power of 2 in constant time. For that we should locate an ancestor of a given leaf which is at specified depth 2^k. This can be reduced to the so-called weighted ancestor queries: given a node-weighted tree, with the weights nondecreasing on any root-to-leaf path, preprocess it to find the predecessor of a given weight among the ancestors of v efficiently. Unfortunately, all known solutions for this problem [**16, 32**] give non-constant query time. We wish to improve this time by abusing the fact that only ancestors at depths 2^k are sought. First note that such ancestor is not necessarily an explicit vertex. We start with considering all edges of S. For each such edge e, we compute the smallest k such that e contains an implicit vertex at depth 2^k (there might be none), and split the edge to make it explicit. We call all original vertices at depths being powers of 2, and all new vertices, marked. For each vertex v we would like to compute the depths of all its marked ancestors, see Figure 3. This can be done in linear time by a single top-bottom transversal, and the information can be stored in a single $\Theta(\log |p|)$-bit

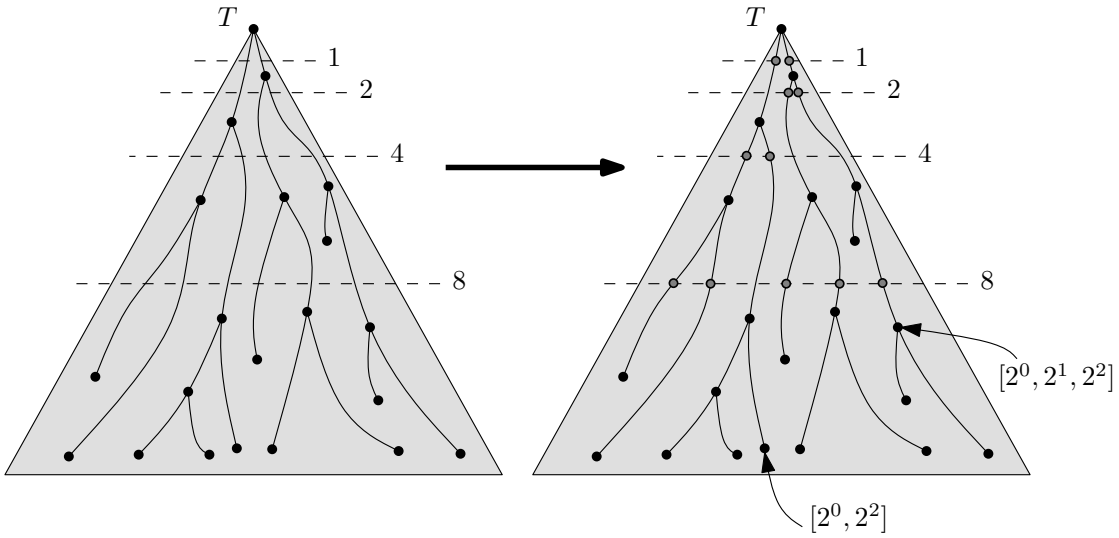


FIGURE 3. Marking vertices at depths 2^k in the suffix tree.

word. More precisely, for each vertex v we construct a single word marked(v) with the k-th bit set iff v has a marked ancestor at depth 2^k. Then we construct $S' = \text{compress}(S)$ containing only the leaves and marked vertices of S by collapsing all maximal fragments of S without such vertices, and build the level ancestor data structure for S' allowing us to find the k-th ancestor of any vertex in constant time. Now given i and k we first locate the leaf v corresponding to $p[i\,..\,|p|]$ in S, then take a look at its bitvector marked(v). We can compute in constant time $t = \{k' > k : k' \in \text{marked}(v)\}$ and retrieve the t-th ancestor of v in S'. Going back to S we get a node with the same (lexicographically) smallest and largest suffix in its subtree as the node corresponding to $p[i\,..\,i + 2^k - 1]$.

While the structure of [**6**] does give constant time answers, we can use a significantly simpler solution building on the fact that the depth of S' is just $\log m$. First we use the standard micro-macro tree decomposition, which gives us a top fragment containing just $\frac{m}{\log m}$ leaves, and a collection of small trees on at most $\log m$ leaves. Note that in this particular case, the total number of vertices cannot be much larger than the number of leaves: the original tree contains no vertices with outdegree 1, then we introduced at most one such vertex at each edge, and then we collapsed some parts of the tree. For each node in the top tree we store all $\log m$ answers explicitly. For each small tree we do as follows: number the vertices in a depth-first order and for each node v compute two bitvectors ancestor(v) and marked(v). ancestor(v) has the i-th bit set iff i is an ancestor of v. marked(v) has the k-th bit set iff v has an ancestor which is at depth 2^k in the original tree. To find the ancestor at depth 2^k of a given vertex v, we consider two cases.

(1) v belongs to the top tree. Then we have the answer available.
(2) v belongs to some small tree. We first check in constant time if the small tree contains any ancestors of v which are at depth $2^{k'}$ with $k' \leq k$ in the original tree (this can be done by checking the lowest bit set in marked(v)). If there are no such ancestors, we can use the precomputed answers stored for the

parent (in the top tree) of the root. Otherwise we compute in constant time $t = \{k' > k : k' \in \text{marked}(v)\}$ and find the t-th bit set in ancestor(v). Then we retrieve the node corresponding to this depth-first number.

□

Observe that the above lemma can be used to give an optimal solution for a slight relaxation of the *substring fingerprints* problem considered in [**16**]. This problem is defined as follows: given a string p, preprocess it to compute any *substring hash* $h_p(p[i\,..\,j])$ efficiently. We require that:

(1) $h_p(p[i\,..\,j]) \in [1, \mathcal{O}(|p|^2)]$ so that the values can be operated on efficiently,
(2) $h_p(p[i\,..\,j]) = h_p(p[k\,..\,l])$ iff $p[i\,..\,j] = p[k\,..\,l]$.

If we allow the range of h_p to be slightly larger, say $\mathcal{O}(|p|^3)$, a direct application of the above lemma allows us to evaluate the fingerprints in constant time after a linear preprocessing. While such range is not optimal, it allows constant time evaluation which should be enough to replace the results of [**16**].

Lemma 8.8. *Substring fingerprints of size $\mathcal{O}(|p|^3)$ can be computed in constant time after a linear time preprocessing.*

Proof. First we apply the preprocessing from Lemma 8.7 to p. We also store $\lfloor \log x \rfloor$ for any $1 \le x \le |p|$. Then given a query $p[i\,..\,j]$ we compute $k = \lfloor \log(j - i + 1) \rfloor$ and using constant time level ancestors queries we locate the lowest existing ancestors of both $p[i\,..\,i + 2^k - 1]$ and $p[j - 2^k + 1\,..\,j]$ in the suffix tree. Then $h_p(p[i\,..\,j])$ is a triple containing $j - i + 1$ and those two ancestors. □

Algorithm 7 Two-way-binary-search(a, b, w)

```
x ← a, y ← b
k ← 1
while 2^k ≤ b − a do
    if w <_lex p[SA[a + 2^k]] then
        y ← a + 2^k
        break
    end if
    if p[SA[b − 2^k]] <_lex w then
        x ← b − 2^k
        break
    end if
    k ← k + 1
end while
r ← binary search for w in p[SA[x]..|p|], p[SA[x + 1]..|p|], ..., p[SA[y]..|p|]
return r
```

Now getting back to the original question, the input to Batched-power-merge is a sequence of pairs of snippets $w_1, w_2, \ldots, w_{|\mathcal{G}_\ell|}$. By Lemma 8.6 we can consider them in a sorted order. For each such pair $w = p[i\,..\,i + 2^{k_1} - 1]p[j\,..\,j + 2^{k_2} - 1]$, we first look up the fragment of the suffix array corresponding to its prefix $p[i\,..\,i + 2^{k_{min}} - 1]$ using

Lemma 8.7. Then we apply binary search in this fragment, with the exception that if the previous binary search was in this fragment as well, we start from the position it finished, not the beginning of the fragment. Additionally, the binary search is performed from the beginning and the end of the interval at the same time, see TWO-WAY-BINARY-SEARCH. If the initial interval is $[a, b]$ and the position we are after is r, such modified search uses just $\mathcal{O}(\log \min(r-a+1, b-r+1))$ applications of Lemma 3.4 instead of $\mathcal{O}(\log(b-a+1))$ time, which is important.

While a single binary search might require a non-constant time, we will show that their amortized complexity is constant. To analyze the whole sequence of those searches, we keep a partition of the whole $[1, |p|]$ into a number of disjoint intervals. Doing a single search splits at most one interval into two parts at the position of the first occurrence. If the first occurrence is exactly at an already existing boundary, there is no split, otherwise we say that those two smaller intervals have been created in phase k_{min} (recall that k_{min} linearly depends on ℓ), and intervals created in phase k_{min} are kept in a list $I_{k_{min}}$. We do not want to split an interval more than once and hence each call to BATCHED-POWERS-MERGE starts with finding for each w_i its corresponding interval in $I_{k_{min}}$. After processing all concatenations, we add the new intervals to $I_{k_{min}}$ and prune it to contain the intervals which are minimal under inclusion. Scanning and pruning $I_{k_{min}}$ takes linear time in its size, and we will show that this size is small.

Algorithm 8 BATCHED-POWERS-MERGE$(w_1, w_2, \ldots, w_{|\mathcal{G}_\ell|})$

```
1: sort all w_i                                                    ▷ Lemma 8.6
2: scan I_{k_min} to find the intervals containing w_i
3: L ← ∅
4: r_0 ← 1
5: for i ← 1 to |G_ℓ| do
6:     [a, b] ← the interval corresponding to w_i[1 .. 2^{k_min}] in SA   ▷ Lemma 8.7
7:     choose [c, d] ∈ I_{k_min} containing the first occurrence of w_i in SA
8:     if [c, d] is defined then
9:         a ← max(a, c)
10:        b ← min(b, d)
11:    end if
12:    a ← max(r_{i−1}, a)
13:    r_i ← TWO-WAY-BINARY-SEARCH(a, b, w_i)
14:    add [a, r_i] and [r_i, b] to L
15: end for
16: sort L and merge it with I_{k_min}, removing non-minimal intervals
17: return all answers r_i
```

Lemma 8.9. *All $\mathcal{O}(\log m)$ calls to* BATCHED-POWERS-MERGE *run in total time $\mathcal{O}(m + \sum_\ell |\mathcal{G}_\ell|)$.*

PROOF. First note that the sorting in line 16 can be performed in time $\mathcal{O}(m^\epsilon + |I_{k_{min}}| + |\mathcal{G}_\ell|)$ using radix sort. Line 1 takes time $\mathcal{O}(m^\epsilon + |\mathcal{G}_\ell|)$ due to Lemma 8.6, and line 2 requires $\mathcal{O}(|I_{k_{min}}| + |\mathcal{G}_\ell|)$. All executions of line 7 take time $\mathcal{O}(|I_{k_{min}}|)$ because the

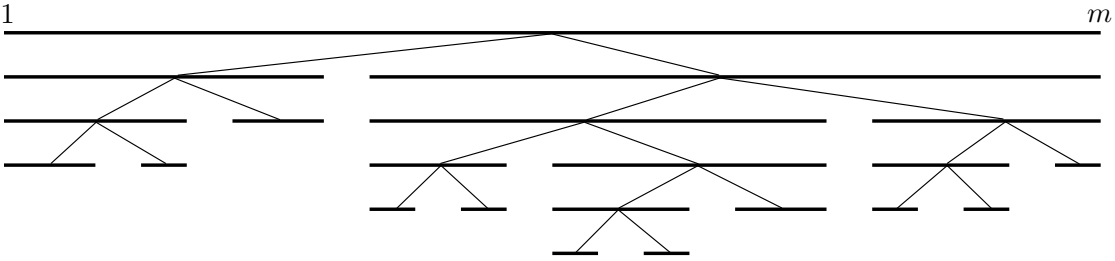


FIGURE 4. Interpreting the intervals as a tree.

words w_i are already sorted. For the time being assume that the binary search in line 13 is for free. Then the total complexity becomes $\mathcal{O}(\sum_i m^\epsilon + \left|I^{(i)}_{k_{min}}\right| + |\mathcal{G}_\ell|)$ where $\left|I^{(i)}_{k_{min}}\right|$ is the size of $I_{k_{min}}$ just before the i-th call to BATCHED-POWERS-MERGE. There is a constant number of those calls for each value of $1 \le \ell \le m$, and each k_{min} corresponds to at most constant number of different continuous values of ℓ, thus the sum is in fact $\mathcal{O}(m + \sum_\ell |\mathcal{G}_\ell|)$.

To finish the proof we have to bound the time taken by all binary searches. For that to happen we will view the intervals as vertices of a tree. Whenever performing a binary search splits an interval into two, we add a left and right child to the corresponding leaf v, see Figure 4. The *rank* $\mathrm{rank}(v)$ of a vertex v is the rounded logarithm of its *weight*, which is the length of the corresponding interval. Then the cost of line 13 is simply $\mathcal{O}(1+\min(\mathrm{rank}(\mathrm{left}(v)), \mathrm{rank}(\mathrm{right}(v))))$ where $\mathrm{left}(v)$ and $\mathrm{right}(v)$ are the left and right child of v, respectively. Hence we should bound the sum $\sum_v \min(\mathrm{rank}(\mathrm{left}(v)), \mathrm{rank}(\mathrm{right}(v)))$, where v is a non-leaf. We say that a vertex is *charged* when its weight does not exceed the weight of its brother. Now we claim that there are at most $\frac{m}{2^k}$ charged vertices of rank k: assume that there are u and v such that u is an ancestor of v, both are charged and of rank k, then weight of v plus weight of its brother is at least twice as large as the weight of v alone, thus the rank of their parent is larger than the rank of v, contradiction. So all charged vertices of the same rank correspond to disjoint intervals, and there cannot be more than $\frac{m}{2^k}$ disjoint intervals of length at least 2^k on a segment of length m. Bounding the sum gives the claim:

$$\sum_v \min(\mathrm{rank}(\mathrm{left}(v)), \mathrm{rank}(\mathrm{right}(v))) \le \sum_{k\ge 0}^{\log m} k\frac{m}{2^k} \le m\sum_{k\ge 0}^{\infty}\frac{k}{2^k} = 2m$$

□

Theorem 8.2. *Given a 0.25-balanced SLP of size $\mathcal{O}(n \log \frac{N}{n})$ and a pattern $p[1\,..\,m]$, we can detect an occurrence of p in the represented text in time $\mathcal{O}(n \log \frac{N}{n} + m)$.*

PROOF. By Lemma 8.5 and Lemma 8.9 we compute the covers of all nonterminals which represent subwords of p in time $\mathcal{O}(n \log \frac{N}{n} + m)$. For the remaining nonterminals X we use Lemma 8.4 to compute $\mathrm{prefix}(X)$ and $\mathrm{suffix}(X)$ in total linear time considering the nonterminals in bottom-up order. Then due to Lemma 8.2 if there is an occurrence of p, there is an occurrence in $\mathrm{prefix}(Y)\,\mathrm{suffix}(Z)$ for some production $X \to YZ$. We consider

every nonterminal X, either lookup the already computed prefix(Y) and suffix(Z) or compute them using the known covers and Lemma 8.4, and use Lemma 4.1 to detect a possible occurrence. □

By using Lemma 8.1 and Theorem 8.2 we get the final result.

Theorem 8.3. *Given a (potentially self-referential) Lempel-Ziv parse of size n describing a text $t[1..N]$ and a pattern $p[1..m]$, we can detect an occurrence of p inside t deterministically in time $\mathcal{O}(n \log \frac{N}{n} + m)$.*

4. Conclusions

Recall that in order to guarantee a $\mathcal{O}(n \log \frac{N}{n} + m)$ running time, it was necessary to use integer division in the proof of Lemma 8.1. This was the only such place, though. If we assume that integer division is not allowed, and the only operations on the integers $start_i, len_i$ appearing in the input triples are addition, subtraction, multiplication and comparing with 0 (which are the only operations used by the $\mathcal{O}(n \log N + m)$ version of our algorithm), we can prove a matching lower bound by looking at the corresponding algebraic computation trees.

An algebraic computation trees model programs which at each step perform either an arithmetic operation and store its result or a branching operation given an input consisting of n numbers $x_1, x_2, \ldots, x_n$. More precisely, such tree has two types of internal nodes:

(1) computation nodes v of degree 1 labeled by instructions of the form $f_v = a \circ b$, where $\circ \in \{+, -, \times, \div\}$ and a, b are either real constants, x_i or f_u for u being an ancestor of v,
(2) branching nodes v of degree 3 labeled by x_i or f_u for u being an ancestor of v and with the outgoing edges labeled by $-, 0, +$.

It is easy to see that one can construct an algebraic computation tree of depth $\mathcal{O}(n \log N + m)$ which correctly recognizes if the pattern occurs in a LZ compressed text. It turns out that no better solution exists. The intuition is that without integer division the only way to figure out the parity of a number is to iteratively subtract powers of 2, and computing parities of a collection of numbers can be reduced to LZ compressed pattern matching. A straightforward application of some known tools [**36, 47**] can be used to formalize this intuition. First we state the main theorem of Yao [**47**] which we are going to apply.

Definition 8.2. A set $W \subseteq \mathcal{R}^n$ is said to be scale-invariant if $\vec{x} \in W$ implies $\lambda \vec{x} \in W$ for all $\lambda > 0$.

Definition 8.3. A set $W \subseteq \mathcal{R}^n$ is said to be rationally dispersed if for every $\vec{x} \in W$ and $\epsilon > 0$ there exists a rational point $\vec{z}$ such that $||\vec{z} - \vec{x}|| < \epsilon$ and $\vec{x} \in W$ iff $\vec{z} \in W$.

Theorem 8.4 (Theorem 1 of [**47**])**.** *Let $W \subseteq \mathcal{R}^n$ be scale-invariant and rationally dispersed. Then the depth of any algebraic computation tree which correctly recognizes integer points from W is at least $c_1(\log \hat{\beta}(W) - 1) - c_2 n$ for some constants c_1, c_2, where $\hat{\beta}(W)$ is the number of connected components of W which are not of measure 0.*

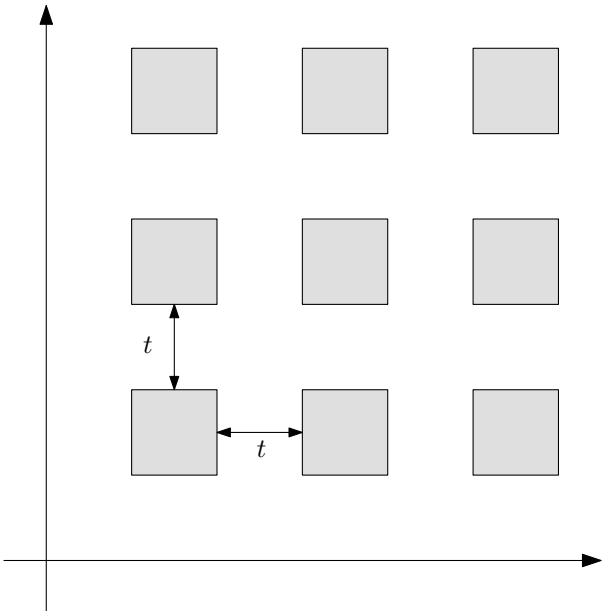


FIGURE 5. Set W for $n = 2$.

To prove a lower bound consider the following set $W \subseteq \mathcal{R}^{n+1}$:

$W = \{t, x_1, x_2, \ldots, x_n : x_i = (2\alpha_i+1)t+\beta_i$ with $0 \leq \beta_i < t, \alpha_i = 0, 1, \ldots, N-1$ for all $i\}$

See Figure 5 for a small example with $n = 2$. Such set is clearly scale-invariant and rationally dispersed. Hence any algebraic computation tree which correctly recognizes integer points from W is $c_1(\log \hat{\beta}(W) - 1) - c_2 n = c_1 \log N^n - c_2 n = \Omega(n \log N)$.

Now observe that one can construct a self-referential LZ of constant size deriving $(1^t 0^t)^N$. Hence one can also construct a LZ of size $\mathcal{O}(n)$ deriving $(1^t 0^t)^N b_1 1 \ldots b_n 1$ where $b_i = \left\lfloor \frac{x_i}{t} \right\rfloor \bmod 2$. This string does not contain 11 as a substring iff all x_i are of the form $x_i = (2\alpha_i + 1)t + \beta_i$ and the lower bound on the depth of any algebraic computation tree which correctly recognizes LZ compressed text which does not contain 11 as a substring follows.

APPENDIX A

Bridge color problem

1. Introduction

The motivation for the bridge color problem comes from object-oriented programming languages, where we have a hierarchy of classes with uniquely defined parents. We also have a collection of m functions accepting a constant number of arguments, with each argument having a specified class as an ancestor in the hierarchy. Then given a function call, we should (efficiently) determine the most specific implementation based on the actual types of the arguments. Such problem was considered by Ferragina *et al.* [**18**], whose methods imply that for the special case of the binary dispatching (where all function are binary) we can achieve $\mathcal{O}(\log\log m)$ query time after a $\mathcal{O}(m^{1+\epsilon})$ preprocessing or $\mathcal{O}(\log m)$ query after a $\mathcal{O}(m\log m)$ preprocessing. Then Eppstein and Muthukrishnan [**14**] improved the former by showing that, for example, $\mathcal{O}(1)$ query time is possible after a $\mathcal{O}(m^{1+\epsilon})$ preprocessing. Finally, Alstrup *et al.* [**2**] decreased the preprocessing space to $\mathcal{O}(m)$ while retaining logarithmic query time. Unfortunately, their preprocessing time was not linear but (expected) $\mathcal{O}(m(\log\log m)^2)$. Another structure with the same bounds was given by Poon and Kwok [**40**]. In this section we give a simpler yet more effective solution with (deterministic) linear time and space preprocessing and the same logarithmic query time.

We work with the following formulation of the problem: we are given a tree T on n vertices and a collection of m *bridges*, which are simply pairs of vertices (u, v). We say that a bridge (u, v) is lower than (u', v') if u' is a descendant of u and v' is a descendant of v. We want to preprocess the input so that given two vertices u' and v' we can detect the lowest bridge (u, v) such that u' is a descendant of u and v' is a descendant of v, see Figure 1. If there is no such unique lowest bridge, we need to signal ambiguity. We aim to develop a $\mathcal{O}(n + m)$ time preprocessing which allows $\mathcal{O}(\log m)$ time queries.

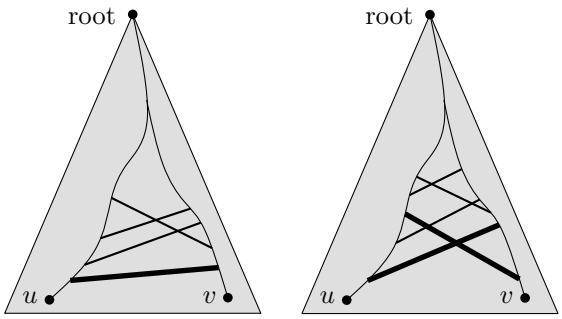


FIGURE 1. Unique lowest bridge for u and v and an ambiguous situation.

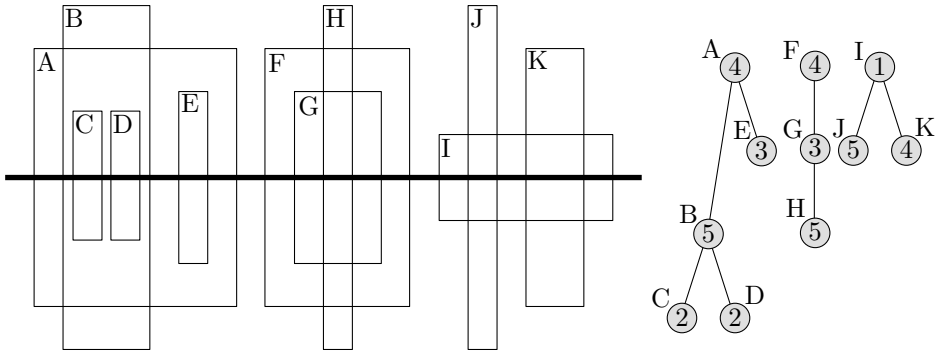


FIGURE 2. Rectangles on the same stack and their version tree.

2. Improved algorithm

First note that the above problem reduces in a natural way (by computing the pre- and post-order numbers) to retrieving the smallest rectangle containing a given point on a $n \times n$ grid (or detecting there is no such unique smallest rectangle). From now on we will work with this simple geometric formulation. The starting point of our structure is the idea from Lemma 7.7. Its goal was (more or less) to solve a variant of the bridge color problem. Of course the situation there was substantially easier (and different): we had additional constraints on the structure of the rectangles, and we had to find any rectangle containing a given point. Now the collection of rectangles is not necessarily valid. Nevertheless, it is *almost valid*, meaning that the x and y projections of any two rectangles are either disjoint or contained in each other. Moreover, by normalizing the coordinates we can assume that $n \leq 2m$.

We sweep the grid from left to right while maintaining a structure describing currently intersected rectangles. The structure is, as in the proof of Lemma 7.7, a full binary tree on n leaves corresponding to different y coordinates. To process an interval $[y_1, y_2]$ with $y_1 < y_2$, we locate the lowest common ancestor v of the leaves corresponding to y_1 and y_2 and call it responsible for $[y_1, y_2]$. Each inner vertex stores a stack containing all intervals it is currently responsible for. To insert a new interval we push it onto its responsible vertex stack. To remove an interval, locate the responsible vertex and observe that (because the collection is almost valid) the interval we want to remove is its top element, and we can simply pop it.

Fix an inner vertex and consider all ℓ rectangles it was responsible for, see Figure 2. Note that the intersection of all their y projections is nonempty, and hence any two of those projections are contained in each other. By a simple linear time transformation we can assume that all their start and end points are different, and their sorted list is $x_1 < x_2 < \ldots < x_\ell$ (of course we cannot assume that sorting a single list can be performed in linear time, but note that we can sort lists of all inner vertices at once, and because their elements are small integers we can apply counting sort). We define the *version tree* as follows: the parent of a rectangle $[x_1, x_2] \times [y_1, y_2]$ is the rectangle $[x_1', x_2'] \times [y_1', y_2']$ such that $[x_1, x_2] \subseteq [x_1', x_2']$ and $[x_1', x_2']$ is the smallest possible. Because any two x projections are either disjoint or contained in each other, and all x_i are

different, this is a valid definition. We may assume that the result is indeed a tree (not a forest) by adding one artificial rectangle. Each vertex of this tree is labeled with an integer denoting the height $y_2 - y_1$ of the corresponding rectangle. Consider the sorted list of all different x coordinates. For each pair of consecutive integers $x_i < x_{i+1}$ on this list we would like to find the vertex of the version tree such that its ancestors are exactly the elements of the stack at time $t \in (x_i, x_{i+1})$ (we call it the *tail* at time t). This can be precomputed in a straightforward way during the sweep.

Consider a query concerning a point (x, y). First we locate the leaf v corresponding to x. Any rectangle containing (x, y) belongs to the stack of one of its ancestors at time x. More specifically, it must be an ancestor of the tail at time x of one of those $\log m$ stacks. Hence we should start with locating all those $\log m$ tails efficiently.

Lemma A.1. *Given a time t we can retrieve its tail at every ancestor of the leaf corresponding to t in total $\mathcal{O}(\log m)$ time after a linear time and space preprocessing.*

Proof. A straightforward application of the fractional cascading technique of Chazelle [**11**]. Recall that this technique allows linear time and space preprocessing of a constant-degree graph with (sorted) lists of elements associated to the vertices so that given a path we can perform binary search for the same value in all lists corresponding to its vertices in time $\mathcal{O}(\log m + p)$, where m is the total size of all lists and p is the length of the path. In our case $p = \log m$ and the claimed running time follows. □

Each version tree will be carefully preprocessed as to implement two operations. Let $\mathrm{path}(v)$ be the set of all ancestor weights of a given vertex v. The first operation is very simple: given v we would like to find the vertex corresponding to $\max \mathrm{path}(v)$. This can be trivially preprocessed in linear time and space. The second operation is more involved: given v and x we would like to find the vertex corresponding to the successor of x in $\mathrm{path}(v)$. Before we show how to implement it efficiently, we formulate some auxiliary lemmas.

Lemma A.2. *A collection of sets of points $S(i)$ on a $n \times n$ grid such that $|S(i)| \leq \log^2 n$ for all i can be preprocessed in $\mathcal{O}(n + \sum_i |S_i|)$ time so that given i and (x, y) we can retrieve the point corresponding to $\min\{y' : (x', y') \in S(i) \wedge x' \leq x \wedge y' \geq y\}$ in constant time.*

Proof. The idea is to recursively build a collection of smaller structures allowing performing the same operation but on subsets of the original set of point.

Assume we have a set of $m \leq \log^2 n$ points (x_i, y_i). By Lemma 3.1 we can sort the points according to their y coordinates in $\mathcal{O}(m)$ time. Then we split the original set into blocks of size roughly $\sqrt{m}$ by choosing $\sqrt{m}$ evenly spaced elements $(x_{i\sqrt{m}}, y_{i\sqrt{m}})$ in this sorted sequence and call the i-th block B_i. We build a smaller structure for each B_i. Additionally, let x'_i be the smallest x coordinate in the i-th block and create a new set of $\sqrt{m}$ points of the form (i, x'_i) which we call *representatives*. We build a smaller structure for this new set. If m is very small, say $m \leq \sqrt{\log n}$, we switch to a different method: we can first normalize the coordinates of the points so that they are at most $\sqrt{\log m}$ and encode the whole set in a single machine word going row-by-row.

Observe that the total size of all structures built for a single $S(i)$ is just $\mathcal{O}(S(i))$. Furthermore, they allow us to answer a single query in constant time as follows. First

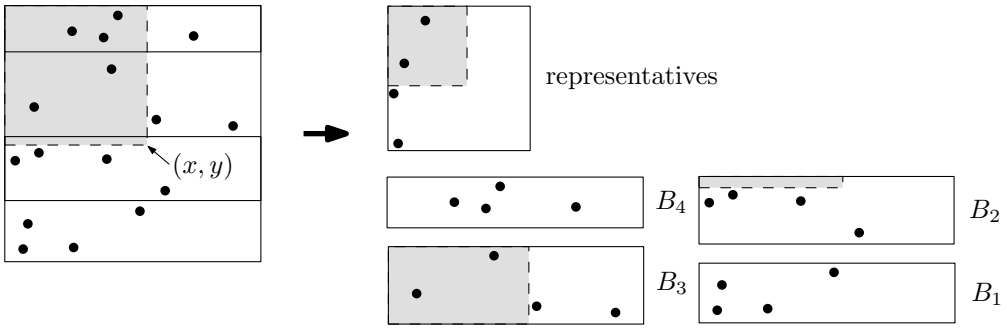


FIGURE 3. Reducing the original query to queries in smaller structures.

locate the block y belongs to and query the corresponding smaller structure. If this smaller structure contains a point (x', y') with $x' \leq x$ and $y' \geq y$, we are done. Otherwise we use the smaller structure built for the representatives to locate the lowermost block containing such point and query its corresponding smaller structure, see Figure 3. If the size of $S(i)$ is small and we have the whole set encoded in a single machine word, we can answer a query by first masking out all bits corresponding to points with too big x or too small y coordinates and then finding the lowest bit set to 1. The total running time is constant because we will inspect just a constant number of structures for a single query. □

Lemma A.3. *A node weighted weighted tree on n vertices with the weights from $\{1, 2, \ldots, n\}$ can be preprocessed in linear time and space so that given v and x we can find the vertex corresponding to the successor of x in* path(v) *in $\mathcal{O}(\log n)$ time.*

PROOF. Let weight(v), depth(v) and size(v) be the weight, the depth, and the subtree size of a given vertex v, respectively. We start with finding a heavy path decomposition of the tree. Each vertex chooses an edge leading to a child with the largest corresponding size(v). Removing all non-chosen edges leaves us with a collection of paths. We define the *path tree* by creating one vertex for each such path, and choosing the parent of a path p by looking up the parent of its highest vertex head(p) in the original tree and retrieving the corresponding path. It is easy to see that the depth of any path tree is just $\log n$. We say that a path p is above a vertex v if the path v belongs to is a descendant of p. The *path depth* is the depth of a path in the path tree.

Consider a single path. By preprocessing all paths at once we can construct a sorted list of all weights weight$(v_1) \leq$ weight$(v_2) \leq \ldots \leq$ weight(v_ℓ) on this path. We choose $\frac{\ell}{\log^2 n}$ evenly spaced weights weight$(v_{\alpha \log^2 n})$ and call the corresponding vertices *important*. Note that the total number of all important vertices is at most $\frac{n}{\log^2 n}$. We would like to implement the following operation efficiently: given v and x, for each path above v find the successor of x among the chosen weights of the path.

For each path p we build a binary search tree containing all important vertices located on all paths corresponding to the ancestors of p in the path tree. The vertices are sorted according to their weight. By using any persistent balanced search trees we can build the structures in total linear time and space. Furthermore, the total number of new

nodes created as a result of all inserts will be just $\mathcal{O}(\frac{n}{\log n})$. For the sake of concreteness, assume that we use trees in which the original elements are stored only in the leaves. To facilitate efficient query processing, at each vertex v we store an additional *small structure* mapping a path depth to the smallest weight stored at the subtree of v and originating from an important vertex with such path depth. This structure consists of an array of size $\log n$ and one word with the i-th bit set if and only if the i-th entry in the array is defined. The additional structures are of just $\mathcal{O}(\log n)$ size, hence we can afford to build one for each new node in total linear time and space. Now given a query concerning a vertex v, we locate its path and the corresponding binary search tree. Then we find the successor of x in this tree with a single $\mathcal{O}(\log n)$ time transversal. We claim that the small structures stored at all right brothers of the visited vertices give us enough information to locate the successors of x among the chosen weights of all paths above v. This is fairly obvious if we consider a single such path. The tricky part is to extract the information for all of them in $\mathcal{O}(\log n)$ time. We go through the vertices in a bottom-up order. In the very beginning we do not have the successor on any path. We iteratively consider the right brother of the next visited vertex: its small structure gives us the successor for every path for which the corresponding entry is defined and which is yet unknown. By storing the currently unknown set in a single word, we can compute this intersection in constant time, and then extract the successors in constant time per path.

Now we can assume that we know the successor of x among the chosen weights in every path p above v. Consider a single path. For each chosen weight $\text{weight}(v_{\alpha \log^2 n})$ we construct a small set of all pairs

$$\Big(\text{depth}(v_{\alpha \log^2 n - \Delta}), -\text{weight}(v_{\alpha \log^2 n - \Delta})\Big) \qquad \text{for all } \Delta = 0, 1, \ldots, \log^2 n$$

which we call the α-th group. We apply Lemma A.2 to each such group. Then to compute the successor of x among the weights in a prefix of p we first retrieve the structure corresponding to the group of the known chosen weight $v_{\alpha \log^2 n}$. If this structure contains $(\text{depth}(v_i), \text{weight}(v_i))$ with $\text{depth}(v_i) \leq \text{depth}(v)$ and $\text{weight}(v_i) \geq x$ then we are done. Otherwise the situation is more complicated. We choose from each group the highest vertex and call those vertices *representatives*. We would like to implement the following operation efficiently: given v and x, for each path above v find the successor of x among the weights of representatives which are the ancestors of v. Because the total number of representatives is just $\frac{n}{\log^2 n}$, we can use here the same method of building a set of persistent binary search trees with smallest structures associated to each vertex as we did in the previous paragraph. Then for each path we take the representative which is the successor of x and retrieve the structure corresponding to its group. By querying it we can extract the successor of x among all weights in a prefix of p containing the ancestors of v in constant time. □

Now we are ready to prove the main lemma in this section.

Lemma A.4. *A set of m rectangles on a $n \times n$ grid can be preprocessed in linear time and space so that given a point (x, y) we can find the rectangle $[x_1, x_2] \times [y_1, y_2]$ containing (x, y) with the smallest height $y_2 - y_1$ in $\mathcal{O}(\log m)$ time.*

PROOF. First apply Lemma A.1 to locate the tail at time x in every ancestor of the leaf corresponding to x. Then select v to be the lowest of those ancestors such that the maximum on the corresponding tail-to-root path is sufficiently big to contain y in the corresponding interval. There are just $\log m$ ancestors and extracting each maximum requires constant time. Now we claim that the smallest rectangle can be found by looking at the version tree of v only. Assume otherwise, i.e., there is v' such that v' is an ancestor of v and the smallest rectangle belongs to the version tree of v'. But then the y intervals of both rectangles intersect, and hence the interval of the rectangle at v is smaller.

To finish the proof, we apply Lemma A.3 to each version tree. Then given the tail at v we first compute the smallest possible height of a rectangle stored in this tree which guarantees containing y inside. This can be performed in $\mathcal{O}(\log m)$ time if we store all y intervals sorted according to their lengths. Then we compute the successor of this smallest possible height on the tail-to-root path. It corresponds to the smallest height rectangle. □

Theorem A.1. *Bridge color problem can be solved in $\mathcal{O}(\log m)$ time after a linear time and space preprocessing.*

PROOF. First we transform the input so that no two bridges share an endpoint. This can be ensured by increasing the number of vertices to at most $n' = n + 2m$ by repeating the following procedure: given a group of bridges (u, v_1), (u, v_2) ,. . ., (u, v_k) with the same endpoint u, sort them so that $\text{depth}(v_i) < \text{depth}(v_{i+1})$ for all $i = 1, 2, \ldots, k-1$. Then replace u by a path $u_1 \to u_2 \to \ldots \to u_k$ and create k bridges (u_i, v_i) for $i = 1, 2, \ldots, k$. We interpret the resulting set of bridges as a collection of rectangles on a $n' \times n'$ grid.

By Lemma A.4 we can find the rectangle $[x_1, x_2] \times [y_1, y_2]$ containing (x, y) with the smallest height $y_2 - y_1$ in $\mathcal{O}(\log m)$ time. By swapping all x and y coordinates, we can also find the rectangle with the smallest width $x_2 - x_1$. Note that because no two bridges share an endpoint, those two rectangles are uniquely defined. If they are different, we signal ambiguity. Otherwise this rectangle is the unique lowest bridge. □

Bibliography

[1] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18:333–340, June 1975.

[2] S. Alstrup, G. S. Brodal, I. L. Gørtz, and T. Rauhe. Time and space efficient multi-method dispatching. In M. Penttonen and E. M. Schmidt, editors, *SWAT*, volume 2368 of *Lecture Notes in Computer Science*, pages 20–29. Springer, 2002.

[3] A. Amir and G. Benson. Efficient two-dimensional compressed matching. In *Data Compression Conference*, pages 279–288, 1992.

[4] A. Amir, G. Benson, and M. Farach. Let sleeping files lie: pattern matching in Z-compressed files. In *SODA '94: Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 705–714, Philadelphia, PA, USA, 1994. Society for Industrial and Applied Mathematics.

[5] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *LATIN '00: Proceedings of the 4th Latin American Symposium on Theoretical Informatics*, pages 88–94, London, UK, 2000. Springer-Verlag.

[6] M. A. Bender and M. Farach-Colton. The level ancestor problem simplified. *Theor. Comput. Sci.*, 321(1):5–12, 2004.

[7] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.

[8] D. Breslauer. Saving comparisons in the Crochemore-Perrin string matching algorithm. In *In Proc. of 1st European Symp. on Algorithms*, pages 61–72, 1995.

[9] D. Breslauer. The suffix tree of a tree and minimizing sequential transducers. In *Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching*, CPM '96, pages 116–129, London, UK, 1996. Springer-Verlag.

[10] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Rasala, A. Sahai, and a. shelat. Approximating the smallest grammar: Kolmogorov complexity in natural models. In *STOC '02: Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 792–801, New York, NY, USA, 2002. ACM.

[11] B. Chazelle and L. J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1(2):133–162, 1986.

[12] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007.

[13] M. Crochemore and W. Rytter. *Jewels of stringology*. World Scientific, 2002.

[14] D. Eppstein and S. Muthukrishnan. Internet packet filter management and rectangle geometry. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, SODA '01, pages 827–835, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

[15] M. Farach. Optimal suffix tree construction with large alphabets. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, pages 137–, Washington, DC, USA, 1997. IEEE Computer Society.

[16] M. Farach and S. Muthukrishnan. Perfect hashing for strings: Formalization and algorithms. In *Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching*, CPM '96, pages 130–140, London, UK, 1996. Springer-Verlag.

[17] M. Farach and M. Thorup. String matching in Lempel-Ziv compressed strings. In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, STOC '95, pages 703–712, New York, NY, USA, 1995. ACM.

[18] P. Ferragina, S. Muthukrishnan, and M. de Berg. Multi-method dispatching: a geometric approach with applications to string matching problems. In *Proceedings of the thirty-first annual ACM symposium on Theory of computing*, STOC '99, pages 483–491, New York, NY, USA, 1999. ACM.

[19] M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. Syst. Sci.*, 48(3):533–551, 1994.

[20] Z. Galil. String matching in real time. *J. ACM*, 28(1):134–149, 1981.

[21] Z. Galil and J. Seiferas. Time-space-optimal string matching (preliminary report). In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, STOC '81, pages 106–113, New York, NY, USA, 1981. ACM.

[22] P. Gawrychowski. Optimal pattern matching in LZW compressed strings. In *SODA*, pages 362–372, 2011.

[23] P. Gawrychowski. Pattern matching in Lempel-Ziv compressed strings: fast, simple, and deterministic. In *ESA*, 2011, to appear.

[24] L. Gąsieniec, M. Karpinski, W. Plandowski, and W. Rytter. Efficient algorithms for Lempel-Ziv encoding (extended abstract). In R. G. Karlsson and A. Lingas, editors, *SWAT*, volume 1097 of *Lecture Notes in Computer Science*, pages 392–403. Springer, 1996.

[25] L. Gąsieniec and W. Rytter. Almost optimal fully LZW-compressed pattern matching. In *Data Compression Conference*, pages 316–325, 1999.

[26] J. Iacono and O. Özkan. Mergeable dictionaries. In *Proceedings of the 37th international colloquium conference on Automata, languages and programming*, ICALP'10, pages 164–175, Berlin, Heidelberg, 2010. Springer-Verlag.

[27] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006.

[28] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987.

[29] M. Karpinski, W. Rytter, and A. Shinohara. Pattern-matching for strings with short descriptions. In Z. Galil and E. Ukkonen, editors, *Combinatorial Pattern Matching*, volume 937 of *Lecture Notes in Computer Science*, pages 205–214. Springer Berlin / Heidelberg, 1995.

[30] T. Kida, M. Takeda, A. Shinohara, M. Miyazaki, and S. Arikawa. Multiple pattern matching in LZW compressed text. In *Data Compression Conference*, pages 103–112, 1998.

[31] D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[32] T. Kopelowitz and M. Lewenstein. Dynamic weighted ancestors. In *SODA '07: Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 565–574, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[33] S. Kosaraju. Efficient tree pattern matching. *Foundations of Computer Science, Annual IEEE Symposium on*, 0:178–183, 1989.

[34] S. R. Kosaraju. Pattern matching in compressed texts. In *Proceedings of the 15th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 349–362, London, UK, 1995. Springer-Verlag.

[35] Y. Lifshits. Processing compressed texts: A tractability border. In B. Ma and K. Zhang, editors, *CPM*, volume 4580 of *Lecture Notes in Computer Science*, pages 228–240. Springer, 2007.

[36] A. Lubiw and A. Rácz. A lower bound for the integer element distinctiveness problem. *Inf. Comput.*, 94(1):83–92, September 1991.

[37] M. Miyazaki, A. Shinohara, and M. Takeda. An improved pattern matching algorithm for strings in terms of straight-line programs. In A. Apostolico and J. Hein, editors, *CPM*, volume 1264 of *Lecture Notes in Computer Science*, pages 1–11. Springer, 1997.

[38] J. H. Morris, Jr. and V. R. Pratt. A linear pattern-matching algorithm. Technical Report 40, University of California, Berkeley, 1970.

[39] G. Navarro and M. Raffinot. A general practical approach to pattern matching over Ziv-Lempel compressed text. In M. Crochemore and M. Paterson, editors, *CPM*, volume 1645 of *Lecture Notes in Computer Science*, pages 14–36. Springer, 1999.

[40] C. K. Poon and A. Kwok. Space optimal packet classification for 2-d conflict-free filters. In *ISPAN*, pages 260–265, 2004.

[41] M. Rodeh, V. R. Pratt, and S. Even. Linear algorithm for data compression via string matching. *J. ACM*, 28:16–24, January 1981.

[42] W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1-3):211–222, 2003.

[43] T. Shibuya. Constructing the suffix tree of a tree with a large alphabet. In *Proceedings of the 10th International Symposium on Algorithms and Computation*, ISAAC '99, pages 225–236, London, UK, 1999. Springer-Verlag.

[44] I. Simon. String matching algorithms and automata. In *Proceedings of the Colloquium in Honor of Arto Salomaa on Results and Trends in Theoretical Computer Science*, pages 386–395, London, UK, 1994. Springer-Verlag.

[45] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[46] T. A. Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, 1984.

[47] A. C. C. Yao. Lower bounds for algebraic computation trees with integer inputs. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 308–313, Washington, DC, USA, 1989. IEEE Computer Society.

[48] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23:337–343, 1977.