12 Genome compression

A pragmatic problem arising in the analysis of biological sequences is that collections of genomes and especially read sets consisting of material from many species (see Chapter 16) occupy too much space. We shall now consider how to efficiently compress such collections. Observe that we have already considered in Section 10.7.1 how to support pattern searches on top of such compressed genome collections. Another motivation for genome compression comes from the compression distance measures in Section 11.2.5.

We shall explore the powerful *Lempel–Ziv* compression scheme, which is especially good for repetitive genome collections, but is also competitive as a general compressor: popular compressors use variants of this scheme.

In the *Lempel–Ziv parsing* problem we are given a text $T = t_1 t_2 \cdots t_n$ and our goal is to obtain a partitioning $T = T^1 T^2 \cdots T^p$ such that, for all $i \in [1..p]$, T^i is the longest prefix of $t_{|T^1 \cdots T^{i-1}|+1} \cdots t_n$ which has an occurrence starting somewhere in $T^1 \cdots T^{i-1}$. Such strings T^i are called **phrases** of T. Let us denote by L_i the starting position of one of those occurrences (an occurrence of T^i in T starting somewhere in $T^1 \cdots T^{i-1}$). In order to avoid the special case induced by the first occurrences of characters, we assume that the text is prefixed by a virtual string of length at most σ that contains all the characters that will appear in T. Then the string T can be encoded by storing the p pairs of integers $(|T^1|, L_1), (|T^2|, L_2), \ldots, (|T^p|, L_p)$. A naive encoding would use $2p \log n$ bits of space. Decoding the string T can easily be done in time $O(n)$, since, for every i, L_i points to a previously decoded position of the string, so recovering T^i is done just by copying the $|T^i|$ characters starting from that previous position.

In Section 12.2, we shall explore more sophisticated ways to encode the pairs of integers, allowing one to use possibly less space than the naive encoding. We will also explore variations of the Lempel–Ziv compression where the parse $T = T^1 T^2 \cdots T^p$ still has the property that every T^i has an occurrence in T starting in $T^1 \cdots T^{i-1}$, but T^i is not necessarily the longest prefix of $t_{|T^1 \cdots T^{i-1}|+1} \cdots t_n$ with this property.

The motivation for exploring these variations is that they have the potential to achieve better space bounds than achieved by the standard Lempel–Ziv parsing, if the parsing pairs are encoded in a non-naive encoding. Also the techniques to derive these variations include shortest-path computation on DAGs as well as usage of suffix trees, giving an illustrative example of advanced use of the fundamental primitives covered earlier in the book.

We start by showing algorithms for the standard Lempel–Ziv parsing, culminating in a fast and space-efficient algorithm. Those algorithms will make use of the

https://doi.org/10.1017/CBO9781139940023.014 Published online by Cambridge University Press

Knuth–Morris–Pratt (KMP) automaton, the suffix tree, and some advanced tools and primitives based on the BWT index.

To motivate the use of the standard Lempel–Ziv scheme in compressing genome collections, consider a simplistic setting with d length n genomes of individuals from the same species. Assume that these genomes contain s single-nucleotide substitutions with respect to the first one (the reference) in the collection. After concatenating the collection into one sequence of length dn and applying Lempel–Ziv compression, the resulting parse can be encoded in at most $2(n+2s)\log(dn)$ bits, without exploiting any knowledge of the structure of the collection. With the real setting of the mutations containing insertions and deletions as well, the bound remains the same, when redefining s as the size of the new content with respect to the reference.

As mentioned above, this chapter applies heavily a large variety of techniques covered earlier in the book. Hence it offers a means to practice a fluent usage of all those techniques under one common theme.

12.1 Lempel–Ziv parsing

In this section, we focus on the following *Lempel–Ziv parsing* problem. We are given a text $T = t_1 t_2 \cdots t_n$. Our goal is to obtain a greedy partitioning $T = T^1 T^2 \cdots T^p$ such that for all $i \in [1..p]$, T^i is the longest prefix of $t_{|T^1 \cdots T^{i-1}|+1} \cdots t_n$ which has an occurrence in T that starts at a position in $[1..|T^1 \cdots T^{i-1}|]$. The following definition will be used throughout the presentation.

DEFINITION 12.1 *Given a string T and a position i, the* longest previous factor *at position a in T is the longest prefix of $t_a \cdots t_n$ that has an occurrence in T starting at a position in $[1..a-1]$ of T.*

Before studying algorithms to construct the parsing, we derive the following lemmas that characterize the power of Lempel–Ziv parsing.

LEMMA 12.2 *The number of phrases generated by the greedy Lempel–Ziv parsing cannot be more than $2n/\log_\sigma n + \sigma\sqrt{n}$.*

Proof Let us fix a substring length $b = \lceil \log n / 2 \log \sigma \rceil$. The lemma is clearly true whenever $\sigma \geq \sqrt{n}$, so let us assume that $\sigma < \sqrt{n}$. Obviously, there cannot be more than $\sigma^b < \sigma\sqrt{n}$ distinct substrings of length b over an alphabet of size σ. Each such substring could potentially occur once or more in T. Let us now consider the greedy parsing $T^1 \cdots T^p$. Consider Q^i, the prefix of $t_{|T^1 \cdots T^{i-1}|+1} \cdots t_n$ of length b. Then, either Q^i does have an occurrence that starts at some position in $[1..|T^1 \cdots T^{i-1}|]$ or it does not. If it does, then $|T^i| \geq b$. Otherwise $|T^i| < b$, and moreover we have that the first occurrence of Q^i in the whole string T is precisely at position $|T^1 \cdots T^{i-1}| + 1$.

Let us now count the number of phrases of length less than b. Since the number of distinct substrings of length b is upper bounded by $\sigma\sqrt{n}$, we deduce that the number of first occurrences of such substrings is also upper bounded by $\sigma\sqrt{n}$ and thus that the number of phrases of length less than b is also upper bounded by $\sigma\sqrt{n}$. It remains to

https://doi.org/10.1017/CBO9781139940023.014 Published online by Cambridge University Press

bound the total number of phrases of length at least b. Since the total length of such phrases cannot exceed n and their individual length is at least b, we deduce that their number is at most $n/b \leq 2n/\log_\sigma n$ which finishes the proof of the lemma. □

LEMMA 12.3 *The number of phrases generated by the greedy Lempel–Ziv parsing cannot be more than* $3n/\log_\sigma n$.

Proof We consider two cases, either $\sigma < \sqrt[3]{n}$ or $\sigma \geq \sqrt[3]{n}$. In the first case, by Lemma 12.2, we have that the number of phrases is at most $2n/\log_\sigma n + \sigma\sqrt{n} < 2n/\log_\sigma n + n^{5/6} < 3n/\log_\sigma n$. In the second case we have that $\log_\sigma n \leq 3$ and, since the number of phrases cannot exceed the trivial bound n, we have that the number of phrases is at most $n = 3n/3 \leq 3n/\log_\sigma n$. □

With the fact that the two integers to represent each phrase take $O(\log n)$ bits, the lemmas above indicate that a naive encoding of the Lempel–Ziv parsing takes $O(n \log \sigma)$ bits in the worst case. However, the parsing can be much smaller on certain strings. Consider for example the string a^n, for which the size of the parsing is just 1.

We will next show our first algorithm to do Lempel–Ziv parsing. The algorithm is simple and uses only an augmented suffix tree. It achieves $O(n \log \sigma)$ time using $O(n \log n)$ bits of space. We then show a slower algorithm that uses just $O(n \log \sigma)$ bits of space. We finally present a more sophisticated algorithm that reuses the two previous ones and achieves $O(n \log \sigma)$ time and bits of space.

12.1.1 Basic algorithm for Lempel–Ziv parsing

Our basic algorithm for Lempel–Ziv parsing uses a suffix tree ST_T built on the text T. Every internal node v in the suffix tree maintains the minimum among the positions in the text of the suffixes under the node. We denote that number by N_v.

The algorithm proceeds as follows. Suppose that the partial parse $T^1T^2 \cdots T^{i-1}$ has been constructed and that we want to determine the next phrase T^i which starts at position $a = |T^1T^2 \cdots T^{i-1}| + 1$ of T.

We match the string $t_a t_{a+1} \cdots t_n$ against the path spelled by a top-down traversal of the suffix tree, finding the deepest internal node v in the suffix tree such that its path is a prefix of $t_a \cdots t_n$ and $N_v < a$. Then, the length of that path is the length of the longest previous factor of position a in T, which is precisely the length of T^i. Moreover, the pointer L_i can be set to N_v, since the latter points to an occurrence of T^i starting at a position in $T[1..a-1]$. We then restart the algorithm from position $a + |T^i|$ in T in order to find the length of T^{i+1} and so on.

Analyzing the time, we can see that traversing the suffix tree top-down takes time $O(|T^i| \log \sigma)$ for every phrase i, and then checking the value of N_v takes constant time. So the time spent for each position of the text is $O(\log \sigma)$ and the total time is $O(n \log \sigma)$. We leave to Exercise 12.3 the derivation of a linear-time algorithm that computes and stores the value N_v for every node v of the suffix tree.

https://doi.org/10.1017/CBO9781139940023.014 Published online by Cambridge University Press

The total space used by the algorithm is $O(n\log n)$ bits, which is needed in order to encode the suffix tree augmented with the values N_v for every node v. We thus have proved the following lemma.

LEMMA 12.4 *The Lempel–Ziv parse of a text $T = t_1 t_2 \cdots t_n$, $t_i \in [1..\sigma]$, can be computed in $O(n \log \sigma)$ time and $O(n \log n)$ bits of space. Moreover the pairs $(|T^1|, L_1), (|T^2|, L_2), \ldots, (|T^p|, L_p)$ are computed within the same time and space bound.*

12.1.2 Space-efficient Lempel–Ziv parsing

We now give a Lempel–Ziv parsing algorithm that uses space $O(n \log \sigma)$ bits, but requires $O(n \log n \log^2 \sigma)$ time; the time will be improved later. We first present the data structures used.

- A succinct suffix array (see Section 9.2.3) based on the Burrows–Wheeler transform $\mathsf{BWT}_{T\#}$ of T. We use a sampling rate $r = \log n$ for the sampled suffix array used in the succinct suffix array.
- An array $R[1..n/\log n]$ such that $R[i]$ stores the minimum of $\mathsf{SA}_{T\#}[(i-1)\log n + 1..i \log n]$.
- A range minimum query data structure (RMQ) on top of R (see Section 3.1), which returns $\min_{i' \in [l..r]} R[i']$ for a given range $[l..r]$.

Decoding an arbitrary position in the suffix array can be done in time $O(r \log \sigma) = O(\log n \log \sigma)$.

By analyzing the space, we can easily see that the succinct suffix array uses space $n \log \sigma (1 + o(1)) + O(n)$ bits and that both R and RMQ use $O(n)$ bits of space. Note that RMQ answers range minimum queries in $O(\log n)$ time. Note also that the succinct suffix array and the arrays R and RMQ can easily be constructed in $O(n \log \sigma)$ time and bits of space, from the BWT index, whose construction we studied in Theorem 9.11.

We are now ready to present the algorithm. Throughout the algorithm we will use Lemma 9.4 to induce the successive values $\mathsf{SA}^{-1}_{T\#}[1], \mathsf{SA}^{-1}_{T\#}[2], \ldots, \mathsf{SA}^{-1}_{T\#}[n]$, spending $O(\log \sigma)$ time per value. We thus assume that the value $\mathsf{SA}^{-1}_{T\#}[j]$ is available when we process the character t_j.

Suppose that we have already done the parsing of the first $i-1$ phrases and that we want to find the length of the phrase T^i. Denote by a the first position of T^i in T, that is, $a = |T^1 T^2 \cdots T^{i-1}| + 1$. Using Lemma 9.4 we find the position j such that $\mathsf{SA}_{T\#}[j] = a$, or equivalently $\mathsf{SA}^{-1}_{T\#}[a] = j$.

We then find the right-most position ℓ such that $\ell < j$ and $\mathsf{SA}_{T\#}[\ell] < a$ and the left-most position r such that $r > j$ and $\mathsf{SA}_{T\#}[\ell] < a$. Assume, for now, that we know ℓ and r. Then we can use the following lemma, whose easy proof is left for Exercise 12.5, to conclude the algorithm.

LEMMA 12.5 *The length of the longest previous factor (recall Definition 12.1) of position a in T is the largest of $\mathtt{lcp}(T[\ell..n], T[a..n])$ and $\mathtt{lcp}(T[r..n], T[a..n])$, where $\mathtt{lcp}(A, B)$ denotes the length of the longest common prefix of two strings A and B, ℓ*

https://doi.org/10.1017/CBO9781139940023.014 Published online by Cambridge University Press

is the largest position such that $\ell < a$ *and* $\mathsf{SA}_{T\#}[\ell] < \mathsf{SA}_{T\#}[a]$*, and* r *is the smallest position such that* $r > a$ *and* $\mathsf{SA}_{T\#}[r] < \mathsf{SA}_{T\#}[a]$.

It remains to show how to find ℓ and r in time $O(\log^2 n \log \sigma)$. We let $b_a = \lceil a/\log n \rceil$. We use a binary search over the interval $R[1..b_a - 1]$ to find the right-most position b_ℓ in R such that $b_\ell < b_a$ and $R[b_\ell] < a$. This binary search takes time $O(\log^2 n)$. It is done using $O(\log n)$ queries on the RMQ. We then similarly do a binary search over $R[b_a + 1..n/\log n]$ to find the left-most position b_r in R such that $b_r > b_a$ and $R[b_r] < a$ in time $O(\log^2 n)$.

In the next step, we extract the regions $\mathsf{SA}_{T\#}[(b_a - 1)\log n + 1..b_a \log n]$, $\mathsf{SA}_{T\#}[(b_\ell - 1)\log n + 1..b_i \log n]$, and $\mathsf{SA}_{T\#}[(b_r - 1)\log n + 1..b_r \log n]$ using the succinct suffix array. The extraction of each suffix array position takes time $O(\log n \log \sigma)$. The extraction of all the $3\log n$ positions takes time $O(\log^2 n \log \sigma)$. By scanning $\mathsf{SA}_{T\#}[a + 1..b_a \log n]$ left-to-right we can determine the smallest position r such that $a < r \leq b_a \log n$ and $\mathsf{SA}_{T\#}[r] < a$ if it exists. Similarly, by scanning $\mathsf{SA}_{T\#}[b_a \log n..a - 1]$ in right-to-left order, we can determine the largest position ℓ such that $(b_a - 1)\log n + 1 \leq \ell < a$ and $\mathsf{SA}[\ell] < a$ if it exists. If ℓ is not in the interval $[(b_a - 1)\log n + 1..a - 1]$, then we scan $\mathsf{SA}_{T\#}[(b_\ell - 1)\log n + 1..b_\ell \log n]$ right-to-left and ℓ will be the largest position in $[(b_\ell - 1)\log n + 1..b_\ell \log n]$ such that $\mathsf{SA}_{T\#}[\ell] < a$. Similarly, if r is not in the interval $[a + 1..b_a \log n]$, then the value of r will be the smallest position in $\mathsf{SA}_{T\#}[(b_r - 1)\log n + 1..b_r \log n]$ such that $\mathsf{SA}_{T\#}[r] < a$. The final step is to determine $l_1 = \mathtt{lcp}(T[\mathsf{SA}_{T\#}[\ell]..n], T[a..n])$ and $l_2 = \mathtt{lcp}(T[\mathsf{SA}_{T\#}[r]..n], T[a..n])$. This can be done in time $O((l_1 + l_2)\log \sigma)$ by using Lemma 9.5, which allows one to extract the sequence of characters in T that start from positions $\mathsf{SA}_{T\#}[\ell]$ and $\mathsf{SA}_{T\#}[r]$, one by one in $O(\log \sigma)$ time per character. By Lemma 12.5, the length of T^i is precisely the maximum of l_1 and l_2, and moreover $L_i = \ell$ if $l_1 > l_2$, and $L_i = r$, otherwise.

The determination of L_i is done in time $O(|T^i| \log \sigma)$. Overall, we spend $O(|T^i| \log \sigma + \log^2 n \log \sigma)$ time to find the factor T^i. The terms $|T^i| \log \sigma$ sum up to $O(n \log \sigma)$. The terms $\log^2 n \log \sigma$ sum up to $O(n \log n \log^2 \sigma)$, since we have $O(n/\log_\sigma n)$ such terms.

Together with Theorem 9.11, we thus have proved the following lemma.

LEMMA 12.6 *The Lempel–Ziv parse* $T = T^1 T^2 \cdots T^p$ *of a text* $T = t_1 t_2 \cdots t_n$, $t_i \in [1..\sigma]$ *can be computed in* $O(n \log n \log^2 \sigma)$ *time and* $O(n \log \sigma)$ *bits of space. Moreover, the pairs* $(|T^1|, L_1), (|T^2|, L_2), \ldots, (|T^p|, L_p)$ *are computed within the same time and space bounds and the determination of every phrase* T^i *is done in time* $O(|T^i| \log \sigma + \log^2 n \log \sigma)$.

*12.1.3 Space- and time-efficient Lempel–Ziv parsing

Now we are ready to prove the main theorem of Section 12.1.

THEOREM 12.7 *The Lempel–Ziv parse of a text* $T = t_1 t_2 \cdots t_n$, $t_i \in [1..\sigma]$ *can be computed in* $O(n \log \sigma)$ *time and* $O(n \log \sigma)$ *bits of space, given the Burrows–Wheeler transform of* T *and of the reverse of* T*. Moreover the pairs* $(|T^1|, L_1), (|T^2|, L_2), \ldots, (|T^p|, L_p)$ *can be computed within the same time and space bound.*

https://doi.org/10.1017/CBO9781139940023.014 Published online by Cambridge University Press

The algorithm to match the claim of the theorem consists of several cases, where we exploit the earlier lemmas as well as some new constructions. In addition to all the structures shown in Section 12.1.2 (the succinct suffix array, the vector R, and the RMQ), we require the BWT index on the reverse of the text T (that is, $\mathsf{BWT}_{\underline{T\#}}$).

We also make use of a bitvector V of length n bits, indexed to answer rank and select queries in constant time. This takes $n + o(n)$ bits of space: see Section 3.2. The overall space for all the required structures is $O(n\log\sigma)$ bits.

In what follows, we give a high-level overview of the algorithm, dividing it into several cases, which will then be detailed in the forthcoming subsections. In addition to the global data structures listed above, the algorithm uses some local data structures related to the block division defined next and to the cases into which the algorithm is divided.

We start by partitioning the text into log$n$ equally sized blocks of size $B = \lceil n/\log n \rceil$ characters, except possibly for the last one, which might be of smaller size.

We process the blocks left-to-right. When we are processing the Ith block, we say that we are in *phase* I of the algorithm. In phase I, we first build the augmented suffix tree shown in Section 12.1.1 on block I. We then scan block I in left-to-right order and determine the lengths of the phrases that start inside the block. Suppose that the partial parse $T^1T^2\cdots T^{i-1}$ has already been obtained, and that the starting position of the next phrase in the parse is $a = |T^1T^2\cdots T^{i-1}| + 1 \in [(I-1)B + 1..I\cdot B]$. The length of T^i will be the length of the longest previous factor at position a in T.

A phrase can be allocated to one of the following categories according to the position of the phrase and the position of its previous occurrence.

(1) The phrase does not cross a block boundary, and its previous occurrence is within the same block.
(2) The phrase does not cross a block boundary, and its previous occurrence starts in a previous block. This case can be decomposed into two cases:
    (a) the previous occurrence ends in a previous block;
    (b) the previous occurrence ends in the current block.
(3) The phrase crosses a block boundary.

The cases are illustrated in Figure 12.1. In order to find the phrases from different categories, the algorithm considers two separate cases.

In the first case, the algorithm searches for the longest prefix of $t_a\cdots t_{I\cdot B-1}$ which has an occurrence starting in $T[(I-1)B + 1..a-1]$. In the second case, the algorithm searches for the longest prefix of $t_a\cdots t_{I\cdot B-1}$ that has an occurrence starting inside $T[1..(I-1)B]$ and returns a starting position of such a prefix. Then the length of the phrase T^i will be the larger of those two lengths. We have a special case with the last phrase of the block. We now show how to find the length for the first case, and then for the second case, except in the case of the last phrase in the block, which will be described at the end, since it is common to both cases. Finally, we conclude by putting the pieces together. The space and time analyses are summarized at the end of each subsection.

https://doi.org/10.1017/CBO9781139940023.014 Published online by Cambridge University Press

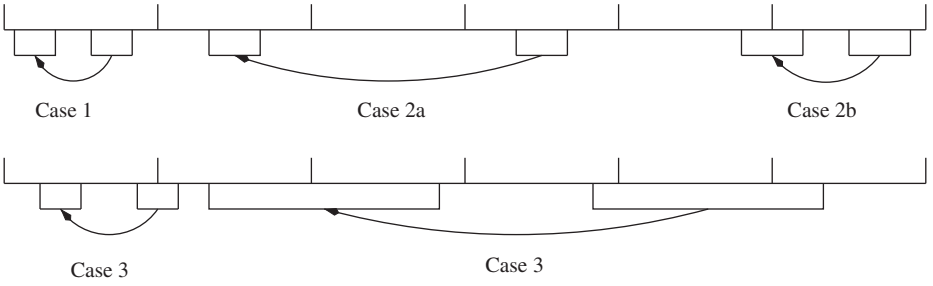

Figure 12.1 Illustration of the different cases of the algorithm in Section 12.1.3. Case 1 is a phrase whose previous occurrence is fully contained in the same block and the phrase does not cross a block boundary. Case 2a is a phrase that does not cross a block boundary and whose previous occurrence occurs before the current block. Case 2b is the same as Case 2a, except that the previous occurrence ends inside the current block. Finally, Case 3 is for a phrase that crosses a block boundary.

Finding occurrences inside the block

Case 1 in Figure 12.1 of finding the longest prefix of $t_a \cdots t_n$ which has an occurrence starting in $T[(I-1)B+1..a-1]$ is easily done using the structure shown in Section 12.1.1 and then using the algorithm described in Section 12.1.2 to determine the length of the longest prefix $t_a \cdots t_n$ which has an occurrence starting in $T[(I-1)B+1..a-1]$ and the starting position of one of those occurrences. The space used by the structure is $O((n/\log n)\log n) = O(n)$ bits and the time is $O((n/\log n)\log \sigma)$.

Finding occurrences before the block

We now consider Case 2 in Figure 12.1, that is, we show how to search for the longest prefix of $t_a \cdots t_n$ which has an occurrence starting in $T[1..(I-1)B]$. We first show how to find the longest prefix of $t_a \cdots t_n$ that has an occurrence fully contained in $T[1..(I-1)B]$ (Case 2a). Let such a longest prefix be of length m. We then show how to find the length of the longest prefix of $t_a \cdots t_n$ which has an occurrence starting in $T[1..(I-1)B]$ and ending at some position in $T[(I-1)B+1..n]$, *but only if that length exceeds* m (Case 2b).

As mentioned before, the algorithm uses a bitvector $V[1..n]$. Initially, all the bits are set to zero. Recall that the bitvector is augmented so as to support rank and select operations in constant time. We assume throughout the algorithm that we can get successively for every position $j \in [1..n]$ the number $\mathsf{SA}_{T\#}^{-1}[j]$ which is the lexicographic order of the prefix $T[1..j]$. This number can be computed using Lemma 9.4: this takes $O(\log \sigma)$ time. Before processing block I, we assume that the bits in V that represent the lexicographic order of all prefixes that end at positions in $[1..I \cdot B]$ in T have all been set to one. We later show how these bits are updated.

We start by doing successive backward steps on $T[a], T[a+1], T[a+2], \ldots, T[I \cdot B]$ on $\mathsf{BWT}_{T\#}$. After each backward step on a character $T[a+k]$, we check whether the interval $V[s_k..e_k]$ contains at least one bit set to one, where $[s_k..e_k]$ is the interval returned by the backward step on character $a+k$. Note that all backward steps are successful, since we are testing only substrings that appear in the text. If there is a bit set to one, we continue

https://doi.org/10.1017/CBO9781139940023.014 Published online by Cambridge University Press

taking backward steps and doing the test again. We stop exactly at the moment where the test fails, because of the following observation.

OBSERVATION 12.8 *Range* $[s_k..e_k]$ *in bitvector* V *contains a bit set to one if and only if the substring* $T[a..a+k]$ *has already appeared in* $T[1..(I-1)B]$.

Proof If $V[p] = 1$ for some $p \in [s_k..e_k]$, then $V[p] = 1$ implies that the prefix of lexicographic rank p ends somewhere in $T[1..(I-1)B]$ and $p \in [s_k..e_k]$ implies that the prefix has $T[a..a+k]$ as a suffix and thus the factor $T[a..a+k]$ occurs inside $T[1..(I-1)B]$.

If $V[p] = 0$ for all $p \in [s_k..e_k]$, then we conclude that none of the prefixes that end in $T[1..(I-1)B]$ can have $T[a..a+k]$ as a suffix and thus that $T[a..a+k]$ does not occur in $T[1..(I-1)B]$. □

Whenever we stop at some position $a + k \leq I \cdot B - 1$, we deduce that the longest prefix of $T[a..I \cdot B]$ that occurs in $T[1..(I-1)B]$ is $T[a..a+k-1]$. Moreover one possible ending position of such a factor is given by $\mathsf{SA}_{\underline{T\#}}[p]$, where p is any position in $[s_{k-1}, e_{k-1}]$ such that $V[p] = 1$. Such a position can be found by rank and select on bitvector V.

We can now set $m = k$ for the k at which the test fails. Now that we have determined the length m of the longest prefix of $T[a..n]$ which has an occurrence fully contained in previous blocks, that is, Case 2a of Figure 12.1, it remains to handle Case 2b of Figure 12.1, where we have an occurrence that ends in the current block. We are interested in such an occurrence only if its length is at least m.

We make use of the Knuth–Morris–Pratt (KMP) algorithm described in Example 2.1 in Section 2.1. Suppose that we have found a factor $T[a..a+m-1]$ of length m such that the length of its previous occurrence in the previous blocks is m. We then build the KMP automaton on $T[a..a+2m-1]$: this takes time $O(m)$. We then consider the window $T[(I-1)B+1-m..(I-1)B+2m-1]$. We use the automaton to find the longest prefix of $T[a..a+2m-1]$ that occurs inside $T[(I-1)B+1-m..(I-1)B+2m-1]$ and that starts inside $T[(I-1)B+1-m..(I-1)B]$. Both the construction of the KMP automaton and the matching of the text fragment $T[(I-1)B+1-m..(I-1)B+2m-1]$ take time $O(m)$. If the length of the found prefix of $T[a..a+2m-1]$ is less than $2m$, we are done, since we could not find an occurrence of $T[a..a+2m-1]$ that starts inside $T[(I-1)B+1-m..(I-1)B]$. Otherwise, if the length of the found prefix is exactly $2m$, we build the KMP automaton on $T[a..a+4m-1]$, scan the text fragment $T[(I-1)B+1-m..(I-1)B+4m-1]$, and use the automaton to determine the longest prefix of $T[a..a+4m-1]$ that occurs inside $T[(I-1)B+1-m..(I-1)B+4m-1]$ and starts inside $T[(I-1)B+1-m..(I-1)B]$. We continue in this way as long as $m' = 2^k m$ for $k = 2, 3, \ldots$ and $m' \leq \log^3 n$. Note that the total time spent to compute m' is $O(2^k m) = O(m')$, since we are spending time $O(2^{k'} m)$ at every value of k' increasing from 1 to k.

Now, if $m' < \log^3 n$, $T[a..a+m']$ will be the longest factor whose current occurrence starts at position a and whose previous occurrence starts inside $[1..(I-1)B]$. Otherwise, we use the algorithm described in Section 12.1.2 to determine the length of the phrase starting at position a in T. This takes time $O(m' \log \sigma + \log^2 n \log \sigma)$.

https://doi.org/10.1017/CBO9781139940023.014 Published online by Cambridge University Press

Handling the last phrase in the block

It remains to handle the case of the last phrase of the block (Case 3 in Figure 12.1). During the course of the algorithm, whenever we detect that $t_a \cdots t_{l \cdot B}$ occurs before or inside the block, we deduce that the phrase T^i starting at position a is actually the last one in the block. In such a case we use directly the algorithm described in Section 12.1.2 to determine the length of T^i in time $O(|T^i| \log \sigma + \log^2 n \log \sigma)$. Since we have only one last phrase per block, the total additional time incurred by the application of the algorithm is $O(\log n \log^2 n \log \sigma) = O(\log^3 n \log \sigma)$.

Postprocessing the block

At the end of the processing of block I, we set the bits that represent the lexicographic order of all prefixes that end at positions in $[(I-1) \cdot B + 1..I \cdot B]$ in T. That is, for every $j \in [(I-1) \cdot B + 1..I \cdot B]$, we set position $V[\mathsf{SA}^{-1}_{\underline{T\#}}[j]]$ to one.

At the end, we update the bitvector V such that the positions that correspond to the ranks in $\mathsf{BWT}_{\underline{T\#}}$ of the prefixes of $T[1..B]$ are set to 1. We can now scan the bitvector V and update the auxiliary rank support structures. The time taken to update the auxiliary structures is $O(n/\log n)$, by taking advantage of bit-parallelism (see Exercise 3.8).

Concluding the proof

The length of each phrase will be the maximum of the lengths found from the two first cases and the special case. It can easily be seen that the total time spent for all phrases is $O(n \log \sigma)$. In all cases, the algorithms will also return a pointer to a previous occurrence, except in Case 2a, in which only a position in a suffix array is returned. We can then use Algorithm 9.2 to transform the suffix array positions into text positions in $O(n \log \sigma + p) = O(n \log \sigma)$ time. This concludes the proof of Theorem 12.7.

*12.2 Bit-optimal Lempel–Ziv compression

In practice it is desirable to engineer a given compression scheme C to encode its input string $S \in [1..\sigma]$ *with the minimum possible number of bits*: we call such a version of C *bit-optimal*. Minimizing the size of the output makes $C(S)$ a better approximation of the Kolmogorov complexity of S, and it might improve the accuracy of the compression-based measure of dissimilarity introduced in Section 11.2.5. Here we focus on making the Lempel–Ziv compression scheme introduced in Section 12.1 bit-optimal. Conceptually, the construction presented here relies on the careful interplay of a number of data structures and algorithms borrowed from different parts of the book, including the shortest paths in a DAG, the construction and traversal of suffix trees, the construction of suffix arrays and LCP arrays, and range-minimum queries.

We denote by LZ the Lempel–Ziv compression scheme and by $\mathsf{LZ}(S)$ its output, that is, a sequence of pairs (d, ℓ), where $d > 0$ is the distance, from the current position i in S, of a substring to be copied from a previous position $i - d$, and ℓ is the length of such a copied substring. Note that we use distances here, rather than absolute positions in S, because we focus on making the output as small as possible. We call both the

https://doi.org/10.1017/CBO9781139940023.014 Published online by Cambridge University Press

copied substring represented by a pair (d, ℓ), and the pair (d, ℓ) itself a *parse*. We call any representation of S as a sequence of phrases a *parse*. Recall that, at every step, LZ copies the *longest substring* that occurs before the current position i: this greedy choice, called the *longest previous factor* (see Definition 12.1), is guaranteed to minimize the number of *phrases* over all possible parses of S. If pairs are encoded with integers of fixed length, then the greedy choice of LZ minimizes also the number of *bits* in the output, over all possible parses. However, if pairs are encoded with a *variable-length encoder*, LZ is no longer guaranteed to minimize the number of bits in the output.

DEFINITION 12.9 *A* variable-length integer encoder *on a range* $[1..n]$ *is a function* $f : [1..n] \mapsto \{0,1\}^+$ *such that* $x \leq y \iff |f(x)| \leq |f(y)|$ *for all* x *and* y *in* $[1..n]$.

Equal-length codes, as well as the gamma and delta codes described in Section 8.1, are variable-length encoders. An encoder f partitions its domain $[1..n]$ into contiguous intervals $\overbrace{b_1}, \overbrace{b_2}, \ldots, \overbrace{b_m}$, such that all integers that belong to interval $\overbrace{b_i}$ are mapped to binary strings of length exactly b_i, and such that two integers that belong to intervals $\overbrace{b_i}$ and $\overbrace{b_j}$ with $i \neq j$ are mapped to binary strings of length $b_i \neq b_j$. Clearly $m \in O(\log n)$ in practice, since n can be encoded in $O(\log n)$ bits. In some cases of practical interest, m can even be constant.

The greedy scheme of LZ can easily be optimized to take variable-length encoders into account: see Insight 12.1. Moreover, specific non-greedy variants of LZ can work well in practice: see Insights 12.2 and 12.3. In this section, we consider the set of *all possible* LZ *parses of* S, which can be represented as a directed acyclic graph.

Insight 12.1 Bit-optimal greedy Lempel–Ziv scheme

A natural way of reducing the number of bits in the output of LZ, without altering its greedy nature, consists in choosing the smallest possible value of d for every pair (d, ℓ) in the parse. This amounts to selecting *the right-most occurrence of each phrase*, a problem that can be solved using the suffix tree of S and its LZ parse, as follows.

Let $\mathsf{ST}_S = (V, W, E)$, where V is the set of internal nodes of ST_S, W is the set of its leaves, and E is the set of its edges. Recall from Section 12.1 that every phrase in the parse corresponds to a node of ST: we say that such a node is *marked*, and we denote the set of all marked nodes of ST by $V' \subseteq V$. A marked node can be associated with more than one phrase, and with at most σ phrases. At most m nodes of ST are marked, where m is the number of phrases in the parse. Let $\mathsf{ST}' = (V', W, E')$ be the *contraction* of ST induced by its leaves and by its marked nodes. In other words, ST' is a tree whose leaves are exactly the leaves of ST, whose internal nodes are the marked nodes of ST, and in which the parent of a marked node (or of a leaf) in ST' is its lowest marked ancestor in ST. Given the LZ parse of S, marking ST takes $O(|S| \log \sigma)$ time, and ST' can be built with an $O(|S|)$-time top-down traversal of ST.

With ST' at hand, we can scan S from left to right, maintaining for every node $v \in V'$ the largest identifier of a leaf that has been met during the scan, and that

https://doi.org/10.1017/CBO9781139940023.014 Published online by Cambridge University Press

is *directly attached to* v in ST'. Specifically, for every position i in S, we go to the leaf w with identifier i in ST', and we set the value of $\texttt{parent}(v)$ to i. If i corresponds to the starting position of a phrase, we first jump in constant time to the node $v \in V'$ associated with that phrase, and we compute the maximum value over all the internal nodes in the subtree of ST' rooted at v. The overall process takes $O(|S|+\sigma \cdot \sum_{v \in V'} f(v))$ time, where $f(v)$ is the number of internal nodes in the subtree of ST' rooted at v. Discarding multiplicities (which are already taken into account by the σ multiplicative factor), a node $v \in V'$ is touched exactly once by each one of its ancestors in ST'. Since v has at most $|\ell(v)|$ ancestors in ST', it follows that $\sum_{v \in V'} f(v) \leq \sum_{v \in V'} |\ell(v)| \leq |S|$, thus the algorithm takes $O(\sigma |S|)$ time overall.

Insight 12.2 Practical non-greedy variants of the Lempel–Ziv scheme

Let W_i be the *longest previous factor* that starts at a position i in string S. In other words, W_i is the longest substring that starts at position i in S, and that occurs also at a position $i' < i$ in S. Let $\mathsf{LPF}[i] = |W_i|$, and let $D[i] = i - i'$, where i' is the right-most occurrence of W_i before i. Recall that the bit-optimal version of LZ described in Insight 12.1 outputs the following sequence of pairs: $(D[i], \mathsf{LPF}[i]), (D[i + \mathsf{LPF}[i]], \mathsf{LPF}[i + \mathsf{LPF}[i]]), \ldots$. Rather than encoding at every position i a representation of its longest previous factor, we could set a domain-specific *lookahead threshold* $\tau \geq 1$, find the position $i^* = \text{argmax}\{\mathsf{LPF}[j] : j \in [i..i + \tau]\}$, and output substring $S[i..i^* - 1]$ explicitly, followed by pair $(D[i^*], \mathsf{LPF}[i^*])$. We could then repeat the process from position $i^* + \mathsf{LPF}[i^*]$.

Using a *dynamic threshold* might be beneficial in some applications. Given a position i in S, we say that the *maximizer of suffix* $S[i..|S|]$ is the smallest position $i^* \geq i$ such that $\mathsf{LPF}[j] < \mathsf{LPF}[i^*]$ for all $j \in [i..i^* - 1]$ and for all $j \in [i^* + 1..i^* + \mathsf{LPF}[i^*] - 1]$. Assume that i^* is initially set to i. We can check whether there is a position $j \in [i^* + 1..i^* + \mathsf{LPF}[i^*] - 1]$ such that $\mathsf{LPF}[j] > \mathsf{LPF}[i^*]$: we call such j a *local maximizer*. If a local maximizer exists, we reset i^* to j and repeat the process, until we find no local maximizer. The resulting value of i^* is the maximizer of the suffix $S[i..|S|]$: we thus output the pair $(D[i^*], \mathsf{LPF}[i^*])$, and we encode the substring $S[i..i^* - 1]$ using a sequence of nonoverlapping local maximizers and of plain substrings of S.

Insight 12.3 Practical relative Lempel–Ziv scheme

In practice we often want to compress a string S with respect to a *reference string* T, for example to implement the conditional compression measure $C(S|T)$ described in Section 11.2.5, or to archive a large set of genomes from similar species (see also Section 10.7.1). In the latter case, we typically store a concatenation T of reference genomes in plain text, and we encode the LZ factorization of every other genome S with respect to T. Note that the LPF vector described in Insight 12.2 becomes in this

https://doi.org/10.1017/CBO9781139940023.014 Published online by Cambridge University Press

case the matching statistics vector $\mathsf{MS}_{S,T}$ of Section 11.2.3, and the array D encodes *positions in* T rather than distances. The space taken by the D component of any LZ parse of S becomes thus a large portion of the output.

However, if genomes S and T are evolutionarily related, the sequence of D values output by any LZ parse should contain a *long increasing subsequence*, which corresponds to long substrings that preserve their order in S and T. We could thus reduce the size of the compressed file by computing the longest increasing subsequence of D values, using for example the algorithm in Example 2.2, and by encoding *differences* between consecutive D values in such increasing subsequence.

The fixed or dynamic lookahead strategies described in Insight 12.2 are also useful for compressing pairs of similar genomes, since they allow one to encode SNPs, point mutations, and small insertions in plain text, and to use positions and lengths to encode long shared fragments.

DEFINITION 12.10 *Given a string S, its* parse graph *is a DAG $G = (V, E)$ with $|S| + 1$ vertices, such that the first $|S|$ such vertices, $v_1, v_2, \ldots, v_{|S|}$, are in one-to-one correspondence with the positions of S. Set E contains arc (v_i, v_{i+1}) for all $i \in [1..|S|]$, and every such arc is labeled by the pair $(0, 1)$. For every vertex $v_i \in V$ and for every string $S[i..j]$ with $j > i + 1$ that starts also at a position $i' < i$ in S, there is an arc (v_i, v_{j+1}) with label (d_e, ℓ_e), where $\ell_e = j - i + 1$ is the length of the arc, $d_e = i - i^*$ is its copy distance from i, and i^* is the starting position of the* right-most *occurrence of $S[i..j]$ before i. G has a unique source $s = v_1$ and a unique sink $t = v_{|S|+1}$.*

Note that G encodes all possible LZ parses of S as *paths* from s to t. Note also that the set of arcs E is *closed*, in the sense that the existence of an arc $e = (v_i, v_{j+1})$ implies the existence of all possible *nested arcs* whose start and end vertices belong to the range $[i..j + 1]$: see Figure 12.2(b). Formally, we can express this as follows.

Arc closure. $(v_i, v_{j+1}) \in E \Rightarrow (v_x, v_y) \in E$ for all $x < y$ such that $x \in [i..j - 1]$ and $y \in [i + 2..j + 1]$.

In other words, given an arc $(v_i, v_{j+1}) \in E$, the subgraph of G induced by vertices $v_i, v_{i+1}, \ldots, v_j, v_{j+1}$ is an orientation of a complete undirected graph. Recall from Exercise 4.9 that such a graph is called a *tournament*. Since there can be at most λ arcs starting from the same vertex of G, where $\lambda \in O(|S|)$ is the length of the longest repeated substring of S, we have that $|E| \in O(|S|^2)$.

Consider now two variable-length encoders f and g used to represent copy distances and lengths, respectively. We denote by ϕ the number of intervals induced by f on the range $[1..|S|]$, and we denote by γ the number of intervals induced by g on the range $[1..\tau + \lambda]$, where $\tau = \overline{|g(\sigma)|}$. Assume that we assign a cost $c(e) = |f(d_e)| + |g(\tau + \ell_e)|$ to every arc e of length greater than one, and a cost $c(e) = |g(S[i])|$ to every arc e of length one. Note that we are reserving the integers $[1..\sigma]$ to encode the *label* of arcs of

https://doi.org/10.1017/CBO9781139940023.014 Published online by Cambridge University Press

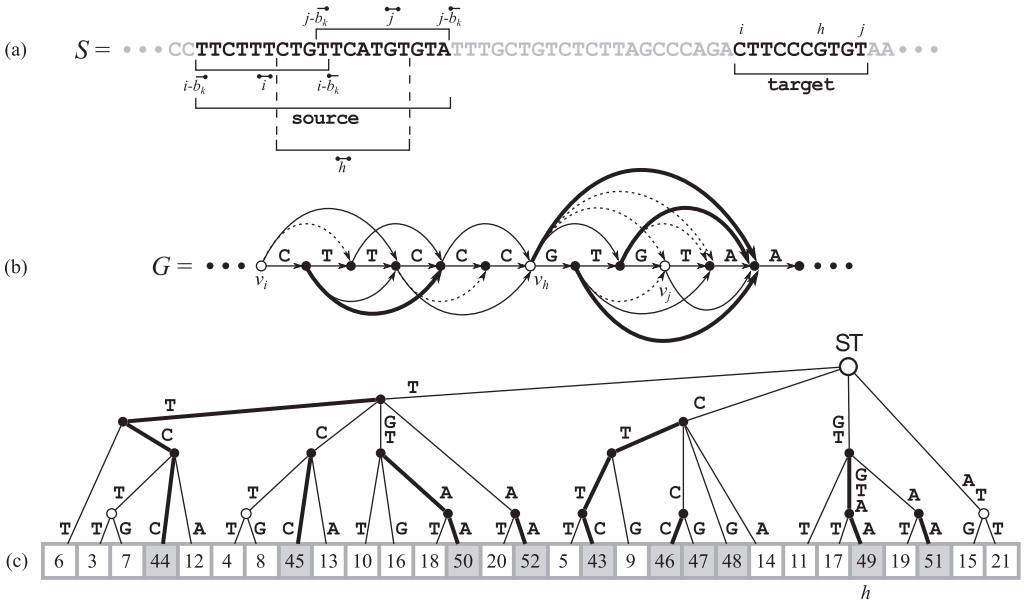

Figure 12.2 Building distance-maximal arcs: a sub-iteration of iteration b_k. In this example, the distances that are encoded in exactly b_k bits are in the range [31..40]. (a) The relationship between the intervals `target` and `source` in S. (b) The portion of the graph G that corresponds to `target`. To reduce clutter, arc labels are not shown, and characters are attached to arcs of length one. Dashed arcs are not distance-maximal, solid arcs are distance-maximal, and bold arcs are distance-maximal with a distance encoded in exactly b_k bits. (c) The compact trie ST of the suffixes that start inside `source` ∪ `target`. For every edge that connects an internal node to a leaf, only the first character of its label is shown. The children of every node are sorted according to the order `T` < `C` < `G` < `A`, which is the order in which the characters in $[1..\sigma]$ first appear when `source` is scanned from left to right. Numbers at leaves indicate their corresponding positions in S. Gray leaves are positions in `target`. The bold path that starts from every gray leaf v (if any) is the set of internal nodes u of ST such that the smallest gray leaf in the subtree rooted at u is v itself.

length one, and we are mapping lengths greater than one to the interval $[\tau + 1..\tau + \lambda]$. We use the shorthand $c(v_i, v_j)$ to mean $c(e)$ where $e = (v_i, v_j)$.

The cost function obeys the following monotonicity property, whose proof is left to Exercise 12.7.

> **Monotonicity of cost.** $c(v_i, v_j) \leq c(v_i, v_{j'})$ for all $j < j'$. Similarly, $c(v_{i'}, v_j) \leq c(v_i, v_j)$ for all $i < i'$.

We are interested in arcs for which $c(v_i, v_j)$ is *strictly smaller* than $c(v_i, v_{j+1})$.

DEFINITION 12.11 *Consider arcs $e = (v_i, v_j)$ and $e' = (v_i, v_{j+1})$. We say that e is* maximal *if $c(e) < c(e')$. In particular, e is* distance-maximal *if $|f(d_e)| < |f(d_{e'})|$, and e is* length-maximal *if $|g(\ell_e)| < |g(\ell_{e'})|$.*

See Figure 12.2(b) for an example of distance-maximal arcs. Note that the cost we assigned to arcs of length one makes them both distance-maximal and length-maximal.

https://doi.org/10.1017/CBO9781139940023.014 Published online by Cambridge University Press

It turns out that maximal arcs suffice to compute the smallest number of bits in the output over all LZ parses of S that use encoders f and g.

LEMMA 12.12 *There is a shortest path from v_1 to $v_{|S|+1}$ that consists only of maximal arcs.*

The easy proof of this lemma is left to Exercise 12.8. We denote by $G' = (V, E')$ the subgraph of G induced by maximal arcs. The following fact is immediately evident from Lemma 12.12, and its proof is developed in Exercise 12.9.

THEOREM 12.13 *Given G', we can compute an LZ parse of S that minimizes the number of bits over all possible LZ parses of S, in $O\big(|S| \cdot (\phi + \gamma)\big)$ time.*

Note that $\phi + \gamma$ can be significantly smaller than $O(|V| + |E|)$ in practice. We are thus left with the task of building G' from S, *without representing G explicitly*.

Before proceeding, let us make the problem a little bit easier. Note that all the arcs that start from the same vertex can be partitioned into equivalence classes $b_{i_1}, b_{i_2}, \ldots, b_{i_k}$ according to the number of bits required to encode their distances using f, where $k \leq \phi$. Once the distance-maximal arc of every equivalence class is known, it is easy to compute all the length-maximal arcs inside each class b_{i_p}, *in constant time per length-maximal arc*: see Exercise 12.10. Thus, in what follows we focus exclusively on building the set of distance-maximal arcs.

*12.2.1 Building distance-maximal arcs

Given a position i in S and a number of bits b, we denote by $\overrightarrow{i_b} = [i - \overrightarrow{b}..i - \overleftarrow{b}]$ the range of all positions in S smaller than i that are addressable from i using exactly b bits (we omit the subscript whenever b is clear from the context). We organize the construction of distance-maximal arcs into ϕ iterations, one iteration for each interval $\overrightarrow{b_k}$ induced by f on the range of distances $[1..|S|]$, in the increasing order $\overrightarrow{b_1}, \overrightarrow{b_2}, \ldots, \overrightarrow{b_\phi}$. Every iteration k is itself organized into $\lceil |S|/|\overrightarrow{b_k}| \rceil$ sub-iterations: in each sub-iteration we examine a distinct interval $\texttt{target} = [i..j]$ of S spanning $|\overrightarrow{b_k}|$ consecutive positions, and we compute all the distance-maximal arcs that start from vertices $\{v_h \in V : h \in \texttt{target}\}$ and whose copy distance is encoded in exactly b_k bits. We will engineer each sub-iteration of iteration k to take $O\left(|\overrightarrow{b_k}| \log |S|\right)$ time and $O(|\overrightarrow{b_k}|)$ space, and we will use global auxiliary data structures that take $O(|S|)$ space. Combined with Theorem 12.13, this approach will provide the key result of this section.

THEOREM 12.14 *The bit-optimal LZ parse of a string S on alphabet $[1..\sigma]$ can be computed in $O(\phi|S| \log |S| + \gamma|S|)$ time and in $O(|S|)$ space.*

Performing all the ϕ iterations in parallel takes just $O\left(\sum_{k=1}^{\phi} |\overrightarrow{b_k}|\right) = O(|S|)$ temporary space in addition to the input and to the auxiliary data structures, and it allows one to build G' and to run the single-source shortest-path algorithm *on the fly, without storing the entire G' in memory*.

Consider the interval $\texttt{source} = [i - \overrightarrow{b_k}..j - \overleftarrow{b_k}] = \overrightarrow{i} \cup \overrightarrow{j}$ of size $2|\overrightarrow{b_k}| - 1$ (see Figure 12.2(a)), and note that source and target might intersect. All arcs that

https://doi.org/10.1017/CBO9781139940023.014 Published online by Cambridge University Press

start from a vertex v_h with $h \in$ `target` and whose distance is encoded in exactly b_k bits correspond to substrings of S that start both inside `target` and inside `source`. Moreover, such substrings cannot start *between* `source` and `target`, that is between position $j - \overbullet{b_k} + 1$ and position $i - 1$, otherwise the distance of the corresponding arc that starts from v_h could be represented in strictly less than b_k bits. Distance-maximal arcs enjoy one more property: their length depends on the range of positions of S that are addressable using their distance.

Length of distance-maximal arcs. If arc $e = (v_h, v_{h+\ell_e+1})$ is distance-maximal with copy distance d_e, then its length ℓ_e equals $\max\{\mathsf{LCP}(h, h') : h' \in \overbullet{h_b},\ b = |f(d_e)|\}$, where $\mathsf{LCP}(h, h')$ is the length of the longest common prefix between suffix $S[h..|S|]\#$ and suffix $S[h'..|S|]\#$.

The easy proof of this property is left to Exercise 12.11. Consider thus the trie ST that contains all suffixes $S[h..|S|]\#$ for $h \in$ `source` $\cup$ `target`, and in which unary paths have been compacted into single edges whose labels contain more than one character (see Figure 12.2(c)). In other words, ST is the subgraph of the suffix tree of S induced by the suffixes that start inside `source` $\cup$ `target`. Let ST_u denote the subtree of ST rooted at node u. Given a position $h \in$ `target`, there is an easy algorithm to detect whether there is a distance-maximal arc that starts from vertex v_h and whose distance is encoded in b_k bits.

LEMMA 12.15 *Let h be a position in* `target`*, let v be the leaf of* ST *associated with position h, and let x be the length of the longest distance-maximal arc that starts from v_h and that has been detected in a previous iteration b_q with $q < k$. Assume that we can answer the following queries for any internal node u of* ST*:*

- `allSmaller`$(u, \overbullet{h})$*, return true if all leaves of* ST_u *are smaller than* $\overbullet{h}$*;*
- `oneLarger`$(u, \overbullet{h})$*, return true if* ST_u *contains a leaf whose position is larger than* $\overbullet{h}$*.*

Then, we can compute the distance-maximal arc from v_h whose distance is encoded in b_k bits, or decide that no such distance-maximal arc exists, in $O(|\overbullet{b_k}|)$ time.

Proof Consider node $u =$ `parent`(v): if ST_u contains a leaf that corresponds to a position of S greater than $\overbullet{h}$, then no arc of G that starts from v_h can possibly encode its distance with b_k bits, because the longest substring that starts both at h and inside $\overbullet{h}$ has a distance that can be encoded with strictly fewer than b_k bits.

Conversely, if all leaves in ST_u correspond to positions in S that are smaller than $\overbullet{h}$, then the distance d_e of arc $e = (v_h, v_{h+|\ell(u)|+1})$ requires either strictly more or strictly fewer than b_k bits to be encoded. Note that we cannot yet determine which of these two cases applies, because substring $\ell(u)$ might start inside the interval $[j - \overbullet{b_k} + 1..i - 1]$, and these positions are not in ST. The same observation holds for all substrings of S that end in the middle of the edge $(\texttt{parent}(u), u)$ in ST. However, it might be that the distance of arc $e' = (v_h, v_{h+|\ell(\texttt{parent}(u))|+1})$ is encoded in b_k bits and that e' is distance-maximal: we thus repeat the algorithm from `parent`(u).

https://doi.org/10.1017/CBO9781139940023.014 Published online by Cambridge University Press

Finally, if ST_u does contain a leaf $p \in \overrightarrow{h}$ (where p might itself be in `target`), then $\ell(u)$ is the longest substring that starts both at h and at a position inside $\overrightarrow{h}$: it follows that arc $e = (v_h, v_{h+|\ell(u)|+1})$ does exist in G, and by the property of the length of distance-maximal arcs, it *might* be distance-maximal. If its copy distance d_e is $h-p$, and thus if $|f(d_e)|$ equals b_k (see for example arc $(v_h, v_{h+5}) = (v_{49}, v_{54})$ in Figures 12.2(b) and (c)), then e is indeed distance-maximal. Clearly d_e might be smaller than $h - p$ if substring $\ell(u)$ occurs *after* the last position of `source` (for example, for arc $(v_i, v_{i+3}) = (v_{43}, v_{46})$ in Figures 12.2(b) and (c)), but in this case the maximality of e has already been decided in a previous iteration b_q with $q < k$. We thus need to compare the length of $\ell(u)$ with the length x of the longest distance-maximal arc in E' that starts from v_h. □

Supporting the queries required by Lemma 12.15 turns out to be particularly easy.

LEMMA 12.16 *After an $O(|\overrightarrow{b_k}|)$-time preprocessing of ST, we can answer in constant time queries* `allSmaller`$(u, \overrightarrow{h})$ *and* `oneLarger`$(u, \overrightarrow{h})$ *for any internal node u of ST and for any $h \in$* `target`.

Proof Since $|\overrightarrow{h}| = |\overrightarrow{b_k}|$, we know that $\overrightarrow{h}$ overlaps either $\overrightarrow{i}$ or $\overrightarrow{j}$, or both. We can thus store in every internal node u of ST the following values:

- `largestI`, the largest position in $\overrightarrow{i}$ associated with a leaf in ST_u;
- `largestJ`, the largest position in $\overrightarrow{j}$ associated with a leaf in ST_u;
- `minTarget`, the smallest position in `target` associated with a leaf in ST_u.

Note that some of these values might be undefined: in this case, we assume that they are set to zero. Given $\overrightarrow{h}$, we can implement the two queries as follows:

$$\texttt{allSmaller}(u, \overrightarrow{h}) = \begin{cases} \texttt{false} & \text{if } h = i, \\ (u.\texttt{largestI} < \overrightarrow{h}) \wedge (u.\texttt{largestJ} = 0) & \text{if } h \neq i, \end{cases}$$

$$\texttt{oneLarger}(u, \overrightarrow{h}) = \begin{cases} \texttt{false} & \text{if } h = j, \\ (u.\texttt{largestJ} > \overrightarrow{h}) \vee (u.\texttt{minTarget} \in [\overrightarrow{h} + 1..h - 1]) & \text{if } h \neq i. \end{cases}$$

Computing $u.\texttt{largestI}$, $u.\texttt{largestJ}$ and $u.\texttt{minTarget}$ for every internal node u of ST takes $O(|\mathsf{ST}|) = O(|\overrightarrow{b_k}|)$ time. □

We would like to run the algorithm in Lemma 12.15 for all $h \in$ `target`. Surprisingly, it turns out that this does not increase the asymptotic running time of Lemma 12.15.

LEMMA 12.17 *Running the algorithm in Lemma 12.15 on all leaves that correspond to positions in* `target` *takes $O(|\overrightarrow{b_k}|)$ time overall.*

Proof It is easy to see that the upward path traced by Lemma 12.15 from a leaf v that corresponds to a position $h \in$ `target` is a (not necessarily proper) prefix of the upward path $P_v = u_1, u_2, \ldots, u_{p-1}, u_p$ defined as follows:

- $u_1 = v$;
- $u_x = \texttt{parent}(u_{x-1})$ for all $x \in [2..p]$;

https://doi.org/10.1017/CBO9781139940023.014 Published online by Cambridge University Press

- h is smaller than the position of all leaves in `target` that descend from u_x, for all $x \in [2..p-1]$;
- there is a leaf in `target` that descends from u_p whose position is smaller than h.

The subgraph of ST induced by all paths P_v such that v corresponds to a position $h \in$ `target` is $O(|\mathsf{ST}|) = O(|\texttt{source} \cup \texttt{target}|) = O(|\overrightarrow{b_k}|)$, since whenever paths meet at a node, only one of them continues upward (see Figure 12.2(c)). This subgraph contains the subgraph of ST which is effectively traced by Lemma 12.15, thus proving the claim. □

*12.2.2 Building the compact trie

The last ingredient in our construction consists in showing that ST can be built efficiently. For clarity, we assume that `source` and `target` intersect: Exercise 12.13 explores the case in which they are disjoint.

LEMMA 12.18 *Let $S \in [1..\sigma]^n$ be a string. Given an interval $[p..q] \subseteq [1..n]$, let* ST *be the compact trie of the suffixes of $S\#$ that start inside $[p..q]$, in which the children of every internal node are sorted lexicographically. Assume that we are given the suffix array $\mathsf{SA}_{S\#}$ of S, the longest common prefix array $\mathsf{LCP}_{S\#}$ of S, the range-minimum data structure described in Section 3.1 on array $\mathsf{LCP}_{S\#}$, and an empty array of σ integers. Then, we can build in $O((q-p)\log|S|)$ time and $O(q-p)$ space a compact trie ST' that is equivalent to* ST*, except possibly for a permutation of the children of every node.*

Proof First, we scan $S[p..q]$ exactly once, mapping all the distinct characters in this substring onto a contiguous interval of natural numbers that starts from one. The mapping is chronological, thus it does not preserve the lexicographic order of $[1..\sigma]$. We denote by S' a *conceptual* copy of S in which all the characters that occur in $S[p..q]$ have been globally replaced by their new code.

Then, we build the artificial string $W = W_1 W_2 \cdots W_{q-p+1}\#$, where each character W_h is a *pair* $W_h = (S'[p+h-1], r_h)$, and r_h is a flag defined as follows:

$$r_h = \begin{cases} -1 & \text{if } S'[p+h..|S|]\# < S'[q+1..|S|]\#, \\ 0 & \text{if } h = q, \\ 1 & \text{if } S'[p+h..|S|]\# > S'[q+1..|S|]\#. \end{cases}$$

The inequality signs in the definition refer to lexicographic order. In other words, we are storing at every position $p+h-1$ inside interval $[p..q]$ its character in S', along with the lexicographic relationship between the suffix of S' that starts at the *following* position $p+h$ and the suffix that starts at position $q+1$. Pairs are ordered by first component and by second component. It is easy to see that sorting the set of suffixes $S'[h..|S|]\#$ with $h \in [p..q]$ reduces to computing the suffix array *of the artificial string* W, and thus takes $O(q-p)$ time and space: see Exercise 12.12.

String W can be built by issuing $|W|$ range-minimum queries on $\mathsf{LCP}_{S\#}$ centered at position $q+1$, taking $O((q-p)\log|S|)$ time overall (see Section 3.1). Recall from

https://doi.org/10.1017/CBO9781139940023.014 Published online by Cambridge University Press

Exercise 8.14 that we can build the longest common prefix array of W in linear time from the suffix array of W. Given the sorted suffixes of S' in the range $[p..q]$, as well as the corresponding LCP array, we can build their compact trie in $O(q-p)$ time: see Section 8.3.2.

The resulting tree is topologically equivalent to the compact trie of the suffixes that start inside range $[p..q]$ in the original string S: the only difference is the *order* among the children of each node, caused by the transformation of the alphabet (see the example in Figure 12.2(c)). □

Recall that the order among the children of an internal node of ST is not used by the algorithm in Lemma 12.15, thus the output of Lemma 12.18 is sufficient to implement this algorithm. To complete the proof of Theorem 12.14, recall that the RMQ data structure on array $\mathsf{LCP}_{S\#}$ takes $O(|S|)$ space, and it can be constructed in $O(|S|)$ time (see Section 3.1).

12.3 Literature

The Lempel–Ziv parsing problem we described in this chapter is also called Lempel–Ziv 77 (Ziv & Lempel 1977). Given a string of length n and its Lempel–Ziv parse of size z, one can construct a context-free grammar of size $O(z\log(n/z))$ that generates S and only S, as shown by Rytter (2003) and Charikar *et al.* (2002). This is the best approximation algorithm for the *shortest grammar problem*, which is known to be NP-hard to approximate within a small constant factor (Lehman & Shelat 2002). Short context-free grammars have been used to detect compositional regularities in DNA sequences (Nevill-Manning 1996; Gallé 2011).

Space-efficient Lempel–Ziv parsing is considered in Belazzougui & Puglisi (2015). The version covered in the book is tailored for our minimal setup of data structures. The computation of the smallest number of bits in any Lempel–Ziv parse of a string described in Section 12.2 is taken from Ferragina *et al.* (2009) and Ferragina *et al.* (2013), where the complexity of Theorem 12.14 is improved to $O(|S|\cdot(\phi+\gamma))$ by using an RMQ data structure that can be built in $O(|S|)$ space and time, and that can answer queries in constant time. The same paper improves the $O(\sigma|S|)$ complexity of the greedy algorithm mentioned in Insight 12.1 to $O(|S|\cdot(1+\log\sigma/\log\log|S|))$. The non-greedy variants of LZ described in Insights 12.2 and 12.3 are from Kuruppu *et al.* (2011a) and references therein, where they are tested on large collections of similar genomes. Other solutions for large collections of genomes appear in Deorowicz & Grabowski (2011) and Kuruppu *et al.* (2011b), and applications to webpages are described in Hoobin *et al.* (2011).

Except for the few insights mentioned above, we did not extensively cover algorithms tailored to DNA compression: see Giancarlo *et al.* (2014) for an in-depth survey of such approaches.

https://doi.org/10.1017/CBO9781139940023.014 Published online by Cambridge University Press

Exercises

12.1 Recall from Exercise 11.7 that a σ-ary de Bruijn sequence of order k is a string of length $\sigma^k + k - 1$ over an alphabet of size σ that contains as substrings all the σ^k possible strings of length k. What is the number of phrases in the Lempel–Ziv parse of such a sequence? What does this tell you about the asymptotic worst-case complexity of a Lempel–Ziv parse? *Hint*. Combine your result with Lemma 12.2.

12.2 Consider the solution of Exercise 12.1. Is the size of the Lempel–Ziv parse of a string a good approximation of its Kolmogorov complexity, roughly defined in Section 11.2.5?

12.3 Show how to build in linear time the augmented suffix tree used in Section 12.1.1.

12.4 Show how to build the vector R described in Section 12.1.2 in $O(n \log \sigma)$ time and in $O(n \log \sigma)$ bits of space.

12.5 Prove Lemma 12.5 shown in Section 12.1.2.

12.6 Consider a generalization of Lempel–Ziv parsing where a phrase can point to its previous occurrence as a *reverse complement*. Show that the space-efficient parsing algorithms of Section 12.1 can be generalized to this setting. Can you modify also the bit-optimal variant to cope with reverse complements?

12.7 Prove the monotonicity of cost property for parse graphs described in Section 12.2.

12.8 Prove Lemma 12.12 by contradiction, working on the shortest path $P = v_{x_1} v_{x_2} \ldots v_{x_m}$ of the parse graph G such that its first nonmaximal arc $(v_{x_i}, v_{x_{i+1}})$ occurs as late as possible.

12.9 Prove Theorem 12.13, using the algorithm for computing the single-source shortest path in a DAG described in Section 4.1.2.

12.10 Consider all arcs in the parse graph G of Definition 12.10 that start from a given vertex, and let $b_{i_1}, b_{i_2}, \ldots, b_{i_k}$ be a partition of such arcs into equivalence classes according to the number of bits required to encode their copy distances using encoder f. Describe an algorithm that, given the length of the distance-maximal arc of class b_{i_p} and the length of the distance-maximal arc of the previous class $b_{i_{p-1}}$, where $p \in [2..k]$, returns all the length-maximal arcs inside class b_{i_p}, in constant time per length-maximal arc.

12.11 Recall from Section 12.2 the relationship between the length ℓ_e of a distance-maximal arc e and the LCP of the suffixes of S that start at those positions which are addressable with the number of bits used to encode the copy distance d_e of arc e. Prove this property, using the definition of a distance-maximal arc.

https://doi.org/10.1017/CBO9781139940023.014 Published online by Cambridge University Press

12.12 With reference to Lemma 12.18, show that sorting the set of suffixes $S'[h..|S|]\#$ with $h \in [p..q]$ coincides with computing the suffix array of string W.

12.13 Adapt Lemma 12.18 to the case in which `source` and `target` do not intersect. *Hint*. Sort the suffixes that start inside each of the two intervals separately, and merge the results using LCP queries.

https://doi.org/10.1017/CBO9781139940023.014 Published online by Cambridge University Press